Modelling Distributed Event-Based Systems Using the kell-m Calculus

Rolando Blanco and Paulo Alencar
David R. Cheriton School of Computer Science
University of Waterloo

Technical Report CS-2011-02

Abstract

In this report we use the kell-m process algebra to develop three models for Distributed Event-Based Systems (DEBSs). The first model is of the DEBS API standard proposed by Pietzuch et al. The second model is for the hierarchical structuring mechanism for components in the REBECA DEBS. The third model is for the internal structure of administrative components in the NaradaBrokering DEBS. These models support the specification of DEBS properties previously proposed in the area using other formalisms. We also show how new properties, based on the locality features provided by kell-m and the ability to passivate kells, can now be specified.

1 Introduction

Distributed Event-Based Systems (DEBSs) are middleware supporting the interaction of publisher and subscriber components via events. DEBSs are typically implemented as networks of distributed brokers. Brokers are administrative components in charge of the routing of events from publishers to interested subscribers.

In DEBSs, the subscribers to be notified when an event is announced are decided at run-time without requiring publisher components to know the name or locations of the subscribers, nor the subscribers to know the name or locations of the publishers. This weak coupling between components makes DEBSs suitable for applications with a large or unpredictable number of autonomous components.

In this report we use kell-m to model DEBSs. Kell-m is an an asynchronous higher-order process algebra with hierarchical localities [8]. Three models for DEBSs are presented. The first model is of the DEBS API standard proposed by Pietzuch et al. [35]. The main reason for the use of the standard API is that it can be supported by well-known DEBSs with little effort [35]. With this model we show: how event-related features previously characterized for DEBSs are supported by the model; how properties previously specified for DEBSs can still be specified; and how new properties not specifiable before can now be specified. The new properties expressible with our work impose requirements on the location of the actions occurring in a DEBS or its applications, and on how processing within locations adapts as the system evolves.

In the second model we represent a hierarchical structuring mechanism for components in the REBECA DEBS [19, 20, 18]. This model showcases the use of kell-m to represent a non-traditional feature proposed for DEBSs. Specifically, in REBECA publishers and subscribers can be grouped. A group is called a *scope*, and a component can belong to multiple scopes. A subscriber is only notified of events published by a component within the same scope as the subscriber.

In the third model, we show how kells, a locality feature in kell-m, can be used to model the internal structure of administrative components in NaradaBrokering [34]. NaradaBrokering is a DEBS used in diverse applications including earthquake modelling, environmental monitoring, and video conferencing and virtual environments [3]. This case study showcases the use of kell-m to represent non-event related features in a DEBS.

The models are specified using a sugared version of kell-m, called *hl-kell-m* for *high-level kell-m*. hl-kell-m provides basic control and modularization constructs and, in general, makes kell-m models more readable. We also present *hl-kμ*, a sugared version of kμ. kμ is the formalism used for the specification of properties of systems represented using kell-m [8].

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

Figure 1: Kell-m Syntax

We start with the presentation of hl-kell-m in Section 2, and hl-kμ in Section 3. In Section 4 we model the DEBS API standard of Pietzuch et al. In Section 5 we model scopes as implemented in REBECA. In Section 6 we show how kells can be used to model the hierarchical structure of NaradaBrokering. Related work is presented in Section 7, and we conclude the report in Section 8.

2 High-Level kell-m

In this section we show how basic control and modularization constructs are represented using kell-m, and introduce hl-kell-m. hl-kell-m is syntactic sugar that makes the modelling of systems using kell-m less laborious. Although not DEBS-specific, these constructs will help us in the specification of DEBS functionality and makes the DEBS models more readable.

We start with a quick introduction to kell-m. For detailed semantics of kell-m, refer to [8]. Processes in the kell-m calculus have the syntax specified in Figure 1. **0** is the *null* process (a process that does not perform any actions), and | is parallel composition of processes.

A write action on channel a is specified $@a(\widetilde{w})$, where $\widetilde{w}$ is the sequence of names and processes being written on channel a. A read action is specified as $a(\widetilde{c}) \triangleright P$, where a is the channel where the action takes place, $\widetilde{c}$ is a list of names to be instantiated with the values passed by the writing process, and P is the process to execute after the read.

An expression $a(r) \triangleright P$ is called a *trigger*. Upon communication, a new trigger is required if more read actions are wanted on channel a. Hence, a special kind of trigger, $a(r) \diamond P$, is used to avoid the need to specify the same trigger multiple times. Notice the use of the symbol $\diamond$ instead of $\triangleright$ in the trigger. A trigger where $\diamond$ is used is called a *recurrent trigger*. Upon communication on channel a, a process $@a(e) \mid a(r) \diamond P$, evolves to the process expression $P\{e/r\} \mid a(r) \diamond P$.

The construct **new** $a\ P$ is called *name restriction*, and it is used to specify a private name a, to be used in P. A private name is also called a *restricted name*. We write **new** *a,b,c* P to represent **new** a **new** b **new** $c\ P$, and **new** $\widetilde{c}\ P$ to represent **new** $c_1, c_2, \ \ldots, c_n\ P$ when $\widetilde{c} = c_1, c_2, \ldots, c_n$.

Processes can execute within localities called *kells*. Kells are identified by name. A process P executing within a kell K is specified as $K[P]$. Because a kell is itself a process, a kell can be within another kell: $K_1[K_2[P]]$. Sometimes we refer to the kells where a process executes as the *locations* of the process. Hence, P in the previous example is located within kells K_1 and K_2.

Another type of trigger, called a *passivation trigger*, is introduced to control the execution of kells. Passivation triggers have the form $K[x] \triangleright Q$, where K is the name of a kell, x is a variable, and Q is a process. The result of composing a passivation trigger with a kell having the same name as the kell specified in the trigger is $Q\{P/x\}$: $K[P] \mid K[x] \triangleright Q \rightarrow Q\{P/x\}$. In the previous example we say the kell $K[P]$ was *passivated* by the trigger. $Q\{P/x\}$ is the process resulting from replacing all occurrences of x in Q with P. Passivation triggers can also be specified using $\diamond$ instead of $\triangleright$, in which case the trigger is a recurrent passivation trigger.

2.1 Fresh Names

Although new restricted names can be created using the **new** construct, there is no corresponding construct to create new *unrestricted* names. Hence, we introduce the syntax:

$$\textbf{fresh}\ c_1, c_2, ..., c_n\ P$$

for,

$$\mathit{fresh}(c_1) \triangleright \mathit{fresh}(c_2) \triangleright \cdots \triangleright \mathit{fresh}(c_n) \triangleright P$$

and assume the existence of a process waiting on channel *fresh* for fresh name requests. Such a process returns a different name every time it is contacted. Assuming names d_1, d_2, d_3, ..., the fresh name generator is specified as follows:

$$\overline{fresh}(d_1) \mid \overline{fresh}(d_2) \mid \overline{fresh}(d_3) \mid \cdots$$

To guarantee the delivered name is fresh, names d_i must not occur in any other process. There must be as many writes $\overline{fresh}(d_i)$, as requests for fresh names are expected. If there are not enough writes, the process requesting a fresh name will be deadlocked. Intuitively, a process in kell-m is deadlocked if it is waiting for communication on a channel no other process will ever write to.

Fresh names can be used instead of restricted names when the newly created name should be visible by every process. Because it is not possible to predetermine the number of required fresh names, when modelling and verifying systems represented using hl-kell-m. This is done only as a convenience since, as previously shown, a provider of fresh names can be constructed natively in kell-m.

2.2 Variables and Procedural Abstractions

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

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

When composed in parallel with the previous process, $\overline{var}(v,\ a)$ makes channel v a variable with value a. A variable is then a channel that, when provided a read channel r and an update channel u, and based on which of these two channels is used, returns the value of the variable or instantiates the variable with a new value.

We introduce the following syntax for the declaration of variables; initialization values are optional:

$$\textbf{var}\ v_1 := val_i, ..., v_n := val_n$$

If no initialization value is provided, we assume the variable has a null value represented by the name *null*. The previous variable declaration is equivalent to process:

$$\overline{var}(v_1, val_1) \mid \cdots \mid \overline{var}(v_n, val_n)$$

Generic, *set* and *get* processes can be defined:

$$\begin{array}{l} \mathit{set}(\mathit{vname},\ \mathit{newval}) \diamond \textbf{new}\ r,u\ (\overline{vname}(r,u) \mid \overline{u}(newval)) \\ \mathit{get}(\mathit{vname},\ rc) \diamond \textbf{new}\ r,u\ (\overline{vname}(r,u) \mid \overline{r}(rc)) \end{array}$$

The following process illustrates how to retrieve, on channel *rc*, the value of a variable *vname*:

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

By convention, 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:

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

where $\widetilde{ps}$ represents zero or more parameters written to c, and $\widetilde{vs}$ are the values returned on channel *rc*. Parameters passed can be names an processes. For these invocations we write: $@c(\widetilde{ps})(\widetilde{vs}) \triangleright \cdots$. For example,

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

can be written:

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

The use of a fresh name instead of a restricted name makes the return channel visible and allows the specification of properties on actions performed on the channel.

Note *val* is bound in P and no return channel is specified. A name is to be *bound* in P in it is visible to P only (a formal definition of bound names is presented in [8]). 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:

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

In the previous process, 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, $\overline{*v}(\widetilde{w})$ is valid, as well as $*v(\widetilde{w}) \triangleright P$, $*v(\widetilde{w}) \diamond P$, and $\overline{a}(*v)$.

Sometimes it is desirable to know when the value of a variable has been changed before it is read. Assuming a variable v, one could try:

$$v := newval \mid *v(val) \triangleright P$$

However there is no guarantee the variable will be read after its value has been set to *newval*. Hence, we extend the definition of variables with a *synchronous update*. In a synchronous update, an extra channel *uc* is received along with the new value for the variable. When the update has been completed, a write is performed on channel *uc*:

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

set and *get* are redefined as:

$$\begin{array}{l} set(vname, newval) \diamond \textbf{new } r,u,s \ (\overline{vname}(r,u,s) \mid \overline{u}(newval)) \\ get(vname, rc) \diamond \textbf{new } r,u,s \ (\overline{vname}(r,u,s) \mid \overline{r}(rc)) \end{array}$$

And a synchronous *set* is introduced:

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

Then, in the process expression:

$$\textbf{new } uc \ (@syncset(v, newval, uc) \mid uc() \triangleright *v(val) \triangleright P)$$

val is instantiated with *newval*. Introducing a synchronous assignment $:=_s$, we write the previous process expression as:

$$(v :=_s newval) \triangleright *v(val) \triangleright P$$

A process P may need to wait for a variable to be created before continuing its execution, $\overline{var}(v, a) \mid *v(b) \triangleright P$. In such cases we write:

$$\textbf{var } v := a \textbf{ in } P$$

Finally, when defining a process, $pname(\widetilde{ps}) \stackrel{def}{=} P$, we write:

$$\textbf{process } pname(\widetilde{ps}) \ \{ \ P \ \}$$

2.3 Conditionals

Consider the following process:

$$\begin{array}{l} \mathit{casetf}(t, P_T, f, P_F) \diamond \textbf{new } T, F \ (\\ \quad T[t() \triangleright (P_T \mid \overline{\mathit{stop}}(F))] \mid \\ \quad F[f() \triangleright (P_F \mid \overline{\mathit{stop}}(T))] \\ ) \end{array}$$

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 not predictable if there are simultaneous writes on both t and f. A process wanting to execute P_T would be:

$$\textbf{new } t,f \ (\overline{\mathit{casetf}}(t, P_T, f, P_F) \mid \bar{t}())$$

Similarly, to execute P_F:

$$\textbf{new } t,f \ (\overline{\mathit{casetf}}(t, P_T, f, P_F) \mid \bar{f}())$$

Based on this process, we write:

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

to represent a process:

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

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

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

if elsif ... elsif else fi statements can be constructed using the basic **if then else fi** process.

Booleans are represented by processes at channels *true* and *false*. Each of the processes receives two channels t and f. The process waiting on *true* always writes on t; the process waiting on *false* always writes on f:

$$\begin{array}{l} \mathit{true}(rc) \diamond \textbf{fresh } t,f \ (\overline{rc}(t, f) \mid \bar{t}()) \\ \mathit{false}(rc) \diamond \textbf{fresh } t,f \ (\overline{rc}(t, f) \mid \bar{f}()) \end{array}$$

A process at channel *not* implements negation:

$$\mathit{not}(t, f, rc) \diamond \overline{rc}(f, t)$$

We write:

$$(\textbf{not } @\mathit{cond}(\widetilde{ps}))(t, f) \triangleright \cdots$$

for,

$$@\mathit{cond}(\widetilde{\mathrm{ps}})(t', f') \triangleright @\mathit{not}(t', f')(t, f) \triangleright \cdots$$

This allows us to write the following expression:

$$\textbf{if } (\textbf{not } @\mathit{cond}(\widetilde{ps})) \textbf{ then } P_T \textbf{ fi}$$

Boolean operators **or**, **and** are similarly defined.

2.4 Lists

Inspired by the implementation of lists in [26, 29], a list receives two channels e and c. If the list is empty, the list writes to e, without passing any values back on e. If the list is not empty, it writes to c the list's header s and tail ss:

$$\begin{array}{l} \mathit{empty}(rc) \diamond \textbf{fresh } l \ (\overline{rc}(l) \mid l(e,c) \diamond \bar{e}()) \\ \mathit{cons}(s,ss,rc) \diamond \textbf{fresh } l \ (\overline{rc}(l) \mid l(e,c) \diamond \bar{c}(s,ss)) \end{array}$$

An empty list l is obtained by executing:

$$@\mathit{empty}()(l)$$

And a list l, with element a, is constructed by:

$$@\mathit{cons}(a, @\mathit{empty}())(l)$$

car and *cdr* are defined with their usual meaning:

$$car(l,\ rc) \diamond \textbf{new}\ e,c\ (\bar{l}(e,c) \mid c(s,ss) \triangleright \overline{rc}(s))$$
$$cdr(l,\ rc) \diamond \textbf{new}\ e,c\ (\bar{l}(e,c) \mid c(s,ss) \triangleright \overline{rc}(ss))$$

Also, we introduce ::, and $[;\cdots;]$ as used in OCaml [2]:

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

The symbol :: is therefore a shorthand for *cons*. Commas can be used instead of semicolons when specifying lists:

$$[a,b,c] \quad \equiv \quad [a;b;c] \quad \equiv \quad @cons(a, @cons(b, @cons(c, @empty())))$$

As well, we introduce:

$$\begin{array}{l} \textbf{match}\ l\ \textbf{with} \\ \quad [\,] \triangleright P_{empty} \\ \quad \textbf{or}\ s :: ss \triangleright P_{cons} \end{array}$$

to represent:

$$\textbf{if}\ @isempty(l)\ \textbf{then}\ P_{empty}\ \textbf{else}\ @ht(l)(s,\ ss) \triangleright P_{cons}\ \textbf{fi}$$

where *isempty* is defined as:

$$\begin{array}{l} isempty(l,\ rc) \diamond \textbf{new}\ e,\ c,\ T,\ F\ \textbf{fresh}\ t,\ f\ (\\ \quad \overline{rc}(t,\ f) \mid \bar{l}(e,\ c) \mid T[e() \triangleright \bar{t}() \mid \overline{stop}(F)] \mid F[c(s,\ ss) \triangleright \bar{f}() \mid \overline{stop}(T)] \\ ) \end{array}$$

We also specify *ht* which returns, both, the head and tail of a list:

$$ht(l,\ rc) \diamond \textbf{new}\ e\ (\bar{l}(e,rc))$$

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

Other usual list functionality can be represented in kell-m as follows:

$$\begin{array}{l} foldr(p,\ v,\ l,\ rc)\ \diamond\ (\\ \quad \textbf{match}\ l\ \textbf{with} \\ \quad\quad [\,] \triangleright \overline{rc}(v) \\ \quad\quad \textbf{or}\ s :: ss \triangleright \overline{rc}(@p(s, @foldr(p,\ v,\ ss))) \\ ) \\ copy(l,\ rc) kp \diamond \overline{rc}(@foldr(cons, [\,], l)) \\ pos(l,\ n,\ rc)\ \diamond\ (\\ \quad \textbf{match}\ l\ \textbf{with} \\ \quad\quad [\,] \triangleright \mathbf{0} \\ \quad\quad \textbf{or}\ s :: ss \triangleright \textbf{if}\ (n = 1)\ \textbf{then}\ \overline{rc}(s)\ \textbf{else}\ \overline{pos}(ss,\ n-1,\ rc)\ \textbf{fi} \\ ) \end{array}$$

For simplicity, we assume support for numbers in kell-m. For ways to represent numbers in process algebras refer to [29].

The process at channel *del* in the following process expression deletes all occurrences of m in list l. We assume the existence of the $=$ operator, which is able to decide if two names are the same.

$$\begin{array}{l} append(l_1, l_2, rc)\ \diamond\ \overline{rc}(@foldr(cons, l_2, l_1)) \\ reverse(l,rc)\ \diamond\ (\\ \quad \textbf{match}\ l\ \textbf{with} \\ \quad\quad [\,] \triangleright \overline{rc}([\,]) \\ \quad\quad \textbf{or}\ s :: ss \triangleright \overline{rc}(@append(@reverse(ss), [s])) \\ ) \\ del(l,m,rc)\ \diamond\ \overline{rc}(@foldr(d, [\,], l)) \end{array}$$

where,

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

To represent sorted lists we require a channel *cmp*, that given two elements, decides if the first element should be before the second one in the sorted list:

$$\begin{array}{l}
cons_s(h,\ hs,\ cmp,\ rc)\ \diamond\ (\\
\quad \textbf{match}\ hs\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \overline{rc}(@cons(h, [\,]))\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ (\\
\quad\quad\quad \textbf{if}\ @cmp(h,s)\ \textbf{then}\ \overline{rc}(h :: hs)\ \textbf{else}\ \overline{rc}(@cons(s, @cons_s(h,\ ss)))\ \textbf{fi}\\
\quad\quad)\\
)
\end{array}$$

A parallel *map* iterator can be implemented by:

$$\begin{array}{l}
map(l,\ p)\ \diamond\ (\\
\quad \textbf{match}\ l\ \textbf{with}\\
\quad\quad [\,]\ \triangleright\ \textbf{0}\\
\quad\quad \textbf{or}\ s :: ss\ \triangleright\ (\\
\quad\quad\quad @p(s)\ |\ @map(ss,\ p)\\
\quad\quad)\\
)
\end{array}$$

We write:

$$\textbf{foreach}\ t\ \textbf{in}\ ts\ \ \textbf{do}\ P_f\ \textbf{done}$$

for,

$$\textbf{fresh}\ p\ (\ (p(t) \diamond P_f)\ |\ \overline{map}(l,\ p)\ )$$

A sequential version of the iterator can be implemented if process P_f writes to a *donec* channel when done:

$$\overline{donec}()\ |\ \textbf{foreach}\ t\ \textbf{in}\ ts\ \ \textbf{do}\ (donec() \triangleright P_f)\ \textbf{done}$$

2.5 Modules

We encapsulate variables and processes into *modules*, a construct similar to OO classes but without inheritance and other OO features. A module is declared with the following syntax:

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

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.

A module declaration corresponds to the following kell-m expression:

$$\begin{array}{l}
name(rc) \diamond \textbf{fresh}\ v_1, ..., v_m\ (\\
\quad \overline{var}(v_1, null)\ |\ \cdots\ |\ \overline{var}(v_m, null)\ |\ \overline{rc}([v_1; ...; v_m])\\
)\ |\\
name__vars(self, rc)\ \diamond\ (\\
\quad @pos(self,\ 1)(v_1) \triangleright @pos(self,\ 2)(v_2) \triangleright \cdots \triangleright @pos(self,\ n)(v_n) \triangleright \overline{rc}(v_1, ..., v_n)\\
)\ |\\
name__c_1(self, \widetilde{ps_1})\ \diamond\ (@name__vars(self)(v_1, ..., v_n)\ \triangleright\ P_1)\ |\\
name__c_2(self, \widetilde{ps_2})\ \diamond\ (@name__vars(self)(v_1, ..., v_n)\ \triangleright\ P_2)\ |\\
\cdots\ |\\
name__c_n(self, \widetilde{ps_n})\ \diamond\ (@name__vars(self)(v_1, ..., v_n)\ \triangleright\ P_n)
\end{array}$$

An instance of a method corresponds to an array of variables, one per variable in the method specification. A process waits on channel *name__*c_i for requests to execute method c_i. The first parameter passed is always a module instance. An extra method *name__vars* is defined for each module and returns the variables in the given array of variables. This extra method is invoked for every other module method causing references for module variables in P_i to be bounded to the variables of the given instance passed as parameter *self*.

An instance *inst* of module *name* can be created with:

$$@name()(inst) \triangleright \cdots$$

The $\triangleright \cdots$ indicates the creation of an instance is a read. Alternatively, we write:

$$inst{:}name \triangleright$$

The following process illustrates how to execute the method c_i in the newly created instance:

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

If method c_i does not return any values, the process to execute is:

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

As illustrated by the previous examples, when invoking a method, the instance must be cast with the name of the module. Variable indirections **v* are not allowed in place of module names.

A module can receive parameters to be used as initial values for the variables or within the methods in the module. The syntax for a module declaration with parameters used to initialize variables is:

$$\begin{array}{l}\textbf{module}\ name(val_1, val_2, ..., val_m)\ \{\\ \quad \textbf{var}\ v_1 := val_1, ..., v_m := val_m;\\ \quad c_1(\widetilde{ps_1}) \triangleright P_1;\\ \quad \cdots\\ \quad c_n(\widetilde{ps_n}) \triangleright P_n;\\ \}\end{array}$$

which corresponds to the kell-m process:

$$\begin{array}{l}name(val_1, val_2, ..., val_m, rc) \diamond \textbf{fresh}\ v_1, ..., v_m\ (\\ \quad \overline{var}(v_1, val_1) \mid \cdots \mid \overline{var}(v_m, val_n) \mid \overline{rc}([v_1; ...; v_m])\\ ) \mid\\ name\_\_vars(self, rc)\ \diamond\ (\\ \quad @pos(self,\ 1)(v_1) \triangleright @pos(self,\ 2)(v_2) \triangleright \cdots \triangleright @pos(self,\ n)(v_n) \triangleright \overline{rc}(v_1, ..., v_n)\\ ) \mid\\ name\_\_c_1(self, \widetilde{ps_1})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_1) \mid\\ name\_\_c_2(self, \widetilde{ps_2})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_2) \mid\\ \cdots \mid\\ name\_\_c_n(self, \widetilde{ps_n})\ \diamond\ (@name\_\_vars(self)(v_1, ..., v_n)\ \triangleright\ P_n)\end{array}$$

For example, the following module *temperature* is used to store the temperature for a given latitude and longitude location. Set and get methods are specified for all variables in the module.

$$\begin{array}{l}\textbf{module}\ temperature(t, lt, ln)\ \{\\ \quad \textbf{var}\ temp := t,\ lat := lt,\ lon := lt;\\ \quad gettemp(rc) \triangleright \overline{rc}(*temp);\\ \quad settemp(ntemp) \triangleright temp := ntemp;\\ \quad getlat(rc) \triangleright \overline{rc}(*lat);\\ \quad setlat(nlat) \triangleright lat := nlat;\\ \quad getlon(rc) \triangleright \overline{rc}(*lon);\\ \quad setlon(nlon) \triangleright temp := nlon;\\ \}\end{array}$$

$$
\begin{array}{lll}
\mathcal{F} & ::= & \texttt{tt} \mid \texttt{ff} \mid \neg\mathcal{F} \mid \mathcal{C}.\mathcal{F} \mid \mathcal{F} \wedge \mathcal{F} \mid \mathcal{F} \vee \mathcal{F} \mid \langle\eta\rangle.\mathcal{F} \mid [\eta].\mathcal{F} \mid N(\widetilde{p}) \\
\mathcal{C} & ::= & a \; op \; b \mid a \in \widetilde{w} \mid |\mathcal{K}| = n \mid a \in \mathcal{K} \mid \neg\mathcal{C} \mid \mathcal{C} \wedge \mathcal{C} \mid \mathcal{C} \vee \mathcal{C} \\
\eta & ::= & \varphi \mid -\varphi \mid S \mid -S \\
\varphi & ::= & (\alpha, \gamma) \mid (\alpha_\tau, \gamma, \gamma) \\
\gamma & ::= & * \mid \mathcal{K} \mid \supseteq \mathcal{K} \mid \not\subseteq \mathcal{K} \mid I \\
N(\widetilde{p}) & ::= & \mathcal{F}\{\widetilde{p}/\widetilde{x}\} \text{ with } N(\widetilde{x}) \stackrel{def}{=} \mathcal{F} \\
op & ::= & = \mid \neq \mid > \mid < \mid \geq \mid \leq
\end{array}
$$

Figure 2: kμ Syntax

We create a temperature instance t for a temperature of 22°C at location (43°, 80°) by invoking:

$$t\text{:}temperature(22, 43, 80) \triangleright \cdots$$

To avoid the need to specify set and get methods for variables in a module, these methods are always provided. Moreover, we write **(t::temperature.temp)* for:

$$@t\text{::}temperature.gettemp()(val)$$

And, *t::temperature.temp* := *newtemp* for:

$$@t\text{::}temperature.settemp(newtemp)$$

Since module variables are regular variables, only get methods are required. Once a variable method has been returned by the get method, its value can be set as a regular variable.

3 High-Level kμ

Properties in kμ are formulas $\mathcal{F}$ with the syntax specified in Figure 2. The syntax is inspired by the language implementation of the $\pi\mu$-calculus in the Mobility Model Checker [44]. Along with the term kμ *formula*, we also use the term kμ *condition* when referring to a kμ property.

tt represents true, ff represents false, $\neg\mathcal{F}$ is used to negate a condition specified by a formula. a and b represent names, n represents a number. $\mathcal{C}$ specifies comparison of values passed in communications, containment conditions on lists of names, containment conditions on kell sets, or checks on the size of a kell containment set.

In the extended labelled transition semantics of kell-m, as specified in [8], transitions are decorated with the channel or kells involved in a communication, as well as the location where the abstractions (reads) and concretions (write) actions are taking place. An actual communication is called a τ transition, and is labelled as $\overleftrightarrow{a}(\widetilde{w})$ or $\overleftrightarrow{K}[P]$. A τ transition labelled $\overleftrightarrow{a}(\widetilde{w})$ corresponds to a communication between a reader process and a writer process on channel a. A τ transition labelled $\overleftrightarrow{K}[P]$ corresponds to a kell passivation on kell K. In general a transition has one of the following two forms:

- $R \xrightarrow{\alpha,\kappa} Q$, with α having the form of an abstraction ($a(\widetilde{c})$ or $K[x]$) or a concretion ($\overline{a}(\widetilde{w})$ or $\overline{K}[P]$). Abstraction $a(\widetilde{c})$ indicates that process R is able to read on channel a. Concretion $\overline{a}(\widetilde{w})$ indicates that process R is able to write on channel a. Abstraction $K[x]$ indicates that process R is able to passivate kell K. Concretion $\overline{K}[P]$ indicates that kell $K[P]$ is executing in R. Q is the resulting process after the transition, and κ is the kell containment set for the process (the locations where the action is taking place).
- $R \xrightarrow{\alpha_\tau,\kappa_a,\kappa_c} Q$, with α_τ having the form $\overleftrightarrow{a}(\widetilde{w})$ or $\overleftrightarrow{K}[P]$. This transition is a τ transition and represents the matching of an abstraction and concretion. κ_a is the kell containment set for the process where the abstraction is executing, and κ_c is the kell containment set for the process where the concretion is executing.

Formulas in kμ have the form $\langle\varphi\rangle.\mathcal{F}$, $[\varphi].\mathcal{F}$, $\langle-\varphi\rangle.\mathcal{F}$, $[-\varphi].\mathcal{F}$, $\langle-\{\varphi_1, \varphi_2, ...\}\rangle.\mathcal{F}$, and $[-\{\varphi_1, \varphi_2, ...\}].\mathcal{F}$, with $\mathcal{F}$ a formula, and φ a condition on a transition. Conditions on transitions specify the action being executed by the process and, optionally, a kell containment condition restricting the location of the process executing the action. When φ is negated, $-\varphi$, the kμ formula applies to transitions where the transition condition φ does not hold.

When the condition φ on a transition is within a diamond modality, $\langle\varphi\rangle.\mathcal{F}$, the formula holds if there is at least one transition from the current process such that the transition meets the conditions imposed by φ, and formula $\mathcal{F}$ holds for the resulting process. When within a box modality, $[\varphi].\mathcal{F}$, $\mathcal{F}$ must hold for every process for which a transition from the current process meets the conditions imposed by φ.

For sets of transition conditions, $\langle\{\varphi_1, \varphi_2, ...\}\rangle.\mathcal{F}$ is equivalent to $\langle\varphi_1\rangle.\mathcal{F} \vee \langle\varphi_2\rangle.\mathcal{F} \vee ...$, and $[\{\varphi_1, \varphi_2, ...\}].\mathcal{F}$ is equivalent to $[\varphi_1].\mathcal{F} \wedge [\varphi_2].\mathcal{F} \wedge ....$

φ is a condition on a concretion or abstraction transition when φ is a pair (α, γ), and α represents an abstraction (i.e., $a(\widetilde{c})$, $K[x]$), or a concretion (i.e., $\overline{a}(\widetilde{w})$, $\overline{K}[P]$). γ represents a kell containment condition on the action α. Communication parameters in α (i.e., $\widetilde{c}$, $\widetilde{w}$, x, and P) can be variables and names. If they are names, they must match the names of the parameters in the transitions; if variables, they are instantiated with the corresponding parameter.

φ is a condition on a τ transition when φ is a triple $(\alpha_\tau, \gamma, \gamma)$, and α_τ represents τ actions (i.e., $\overleftrightarrow{a}(\widetilde{w})$, $\overleftrightarrow{K}[P]$).

Kell containment conditions γ are *any* for $*$, when no condition is imposed on the location of the action; *exactly* for $\mathcal{K}$, when the action must occur within and only within $\mathcal{K}$; *at least* for $\supseteq \mathcal{K}$, when the action must occur in $\mathcal{K}$ and optionally in other locations as well; *except* for $\not\subseteq \mathcal{K}$, when the action cannot occur in $\mathcal{K}$; and *instantiation* for I, where I is a variable, when the location condition is later specified in a $\mathcal{C}$ expression.

We introduce a few sugared constructs for kμ. The intention is to improve the readability of kμ properties. As usual, implication $\mathcal{F} \Rightarrow \mathcal{F}$ is defined from $\neg\mathcal{F}$ and $\mathcal{F} \vee \mathcal{F}$. We use *inert* to specify [-].`ff`. *inert* holds when, in the LTS, there are no more transitions from the current state.

We also introduce the following process definitions:

$$\begin{aligned}
\mathit{future}(N) &\stackrel{def}{=} [-].(N \vee (\neg\mathit{inert} \wedge \mathit{future}(N))) \\
\mathit{future}_e(N) &\stackrel{def}{=} \langle-\rangle.(N \vee \mathit{future}_e(N)) \\
\mathit{Eventually}(N) &\stackrel{def}{=} N \vee \mathit{future}(N) \\
\mathit{Eventually}_e(N) &\stackrel{def}{=} N \vee \mathit{future}_e(N)
\end{aligned}$$

We use $\mathbf{F}(N), \mathbf{F}_e(N), \mathbf{E}(N), \mathbf{E}_e(N)$ as shorthand notations for $\mathit{future}(N), \mathit{future}_e(N), \mathit{Eventually}(N)$, and $\mathit{Eventually}_e(N)$.

When no kell containment condition is specified, $*$ (*any*) is assumed. For example, we write $\langle a(c)\rangle.\mathcal{F}$ for $\langle a(c), *\rangle.\mathcal{F}$. Moreover, if no formula is specified after a $\langle\ .\ \rangle$ and $[\ .\ ]$ modalities, `tt` is assumed. Therefore condition $\langle a(c), *\rangle$.`tt` can be written as $\langle a(c)\rangle$.

Kell containment conditions for τ transitions are also optional, and $*$ is assumed for both abstraction and concretion. If only one kell containment condition is specified for a τ transition, it is assumed the condition applies to the abstraction, and $*$ any is assumed for the concretion. Therefore,

$$\begin{aligned}
\langle\overleftrightarrow{a}(c)\rangle &\equiv \langle\overleftrightarrow{a}(c), *, *\rangle.\mathtt{tt} \\
\langle\overleftrightarrow{a}(c), \gamma\rangle &\equiv \langle\overleftrightarrow{a}(c), \gamma, *\rangle.\mathtt{tt} \\
\langle\overleftrightarrow{a}(c), \gamma_a, \gamma_c\rangle &\equiv \langle\overleftrightarrow{a}(c), \gamma_a, \gamma_c\rangle.\mathtt{tt}
\end{aligned}$$

When specifying properties for methods in modules, communication channels can be specified as *module.method* and *inst::module.method*. Recall method modules are represented on channels named *module__method*, and receive as first parameter a method instance corresponding to a list of module variables (cf. Section 2.5). For example, we write $\langle\mathit{module.method}(\widetilde{ps})\rangle$ for $\langle\mathit{module\_\_method}(\widetilde{ps})\rangle$, and $\langle\mathit{inst::module.method}(\widetilde{ps})\rangle$ for $\langle\mathit{module\_\_method}(\mathit{inst}, \widetilde{ps})\rangle$.

Finally, property naming $\mathit{PropName}(\widetilde{ps}) \stackrel{def}{=} \mathcal{F}$, can be written as:

property $\mathit{PropName}(\widetilde{ps})$ { $\mathcal{F}$ }

4 Common API

Although there is no standard DEBS API supported by each DEBS, a *Common API* has been proposed by Pietzuch et al. [35]. This Common API consists of two other APIs: a *Core API*, and an *Optional API*. DEBSs following the Core API are referred to as *simple DEBSs* [19, 20, 35]. The Core API contains the basic calls required in the subscription and announcement of events. The Optional API extends the Core API by providing calls for DEBSs requiring the advertisement of the events to be published.

The APIs only list the calls, parameters, and returned values. No data types or implementation details are specified. How the calls are implemented in a specific DEBS determines the event model provided by the system. As shown by

```
process coreapi_debs(subscribe, unsubscribe, publish, deliver) {
    fresh sem (
        sem() |
        var subsc := [ ] in (
            subscribe(filter, callback, ttl, rc) ⋄ (
                fresh s (
                    subscription(s, filter, callback, ttl)
                    |
                    sem() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem())
                )
            )
            |
            unsubscribe(s) ⋄ (
                sem() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem()
            )
            |
            publish(e) ⋄ (
                foreach s in *subsc do @s(deliver, e) done
            )
        )
    )
}
```

Figure 3: Core API Specification

Pietzuch et al., the APIs can be supported by well-known DEBSs with little effort. This is the main reason for our use of the APIs: in this report we model DEBSs as systems providing a DEBS API and behaving according to a DEBS event model.

The calls in the Core API are:

```
subscribe(filter, callback, ttl) → subscription
unsubscribe(subscription)
publish(event)
```

`filter` is an expression determining the events of interest to a component. When an event of interest has been published, it is communicated to the interested component via a `callback` routine. `ttl` is a time-to-live or expiration date determining for how long a subscription should be kept in the system. A `subscribe` call returns a `subscription` which, depending on the system, could be an object or handle.

The process *coreapi_debs* in Figure 3 represents generic functionality for DEBSs supporting the Core API. The process takes as arguments channels *subscribe*, *unsubscribe*, *publish*, and *deliver*. With the exception of the *deliver* channel, the other channels correspond to the calls in the API.

The list *subsc* in *coreapi_debs* stores the active subscriptions managed by the DEBS. *sem* is a binary semaphore (implemented as a channel), used to guarantee exclusive access when the list of subscribers is being modified.

Assuming a *subscription* process, specified in Figure 4, when a subscription request is received a subscription s is created, returned, and added to *subsc*. The synchronous variable assignment ($:=_s$, cf. Section 2.2) is used to be able to release the semaphore after the list has been updated. An unsubscribe request removes the given subscription from *subsc*.

Features that typically vary among different systems are parameterized in the *coreapi_debs* model. For example, implementation details with regards to the routing of events between brokers in the DEBS middleware. Because of our process-based approach to the specification of DEBSs, the parameterization of features is done by assuming processes at predetermined channels model specific details for a system of interest. In *coreapi_debs*, delivery of events is parameterized via a *delivery* channel. The specification of the process at the *delivery* channel depends on the actual system being modelled. A trivial delivery process is:

$$deliver(callback, event) \diamond @callback(event)$$

```
process subscription(notify, filter, callback, ttl) {
    notify(deliver, event) ⋄ (
        if (@filter(event) and @ttl()) then
            @deliver(callback, event)
        fi
    )
}
```

Figure 4: Subscription Process for Core API

Other possible delivery process specifications are discussed in [6].

When an event is published, each subscription must decide if the event is to be delivered to the subscriber. Given our process-based approach, *filter* is a channel where a process, capable of identifying if a published event should be delivered to a component, is waiting for filtering requests. Similarly, *callback* is a channel where a subscriber process is waiting for event notifications. *ttl* is a channel representing a time-to-live or expiry time for the subscription. *event* is a channel where a process representing an event can be accessed.

A model for a specific system can then be produced by composing the specification of a DEBS API compatible with the event model of the DEBS of interest, and processes specifying the functionality parameterized in the DEBS API model. These processes model functionality specific to the DEBS of interest:

$$coreapi\_debs \mid SystemSpecificFeature_1 \mid SystemSpecificFeature_2 \mid \cdots$$

4.1 Safety and Liveness Properties

We show how basic safety and liveness properties, previously proposed for DEBSs [31, 20], can be specified when the *coreapi_debs* process is used as the model of a DEBS. We specify two safety properties: events are not delivered to uninterested subscribers, and delivery of an event to a subscriber never occurs prior to the subscription and publication. We also specify a liveness property requiring published events to be notified of all subscribed components.

We start by specifying a property holding when a time-to-live for a subscription is still active:

$$\textbf{property } c\_active(Ttl) \ \{ \ \mathbf{E}_e(\langle \overleftrightarrow{Ttl}(Rc) \rangle.returns\_true(Rc)) \ \}$$

We use uppercase for variables. $\mathbf{E}_e$, existential eventually, was defined in Section 3, and holds if the formula received as its only parameter holds in at least one transition from the current process, or in the future as the process evolves.

For a subscription, *c_active* holds if the process at *Ttl* returns *true*. Recall from Section 2.3, booleans in kell-m are represented by two channels. If communication occurs on the first channel, *true* is assumed, and if communication occurs on the second channel, *false* is assumed instead. In *returns_true*, the channel corresponding to *true* is represented with variable T:

$$\textbf{property } returns\_true(Rc) \ \{ \ \mathbf{E}_e(\langle \overleftrightarrow{Rc}(T,F) \rangle.\mathbf{E}_e(\langle \overleftrightarrow{T}() \rangle)) \ \}$$

Properties *c_active* and *returns_true* require actual communications to occur in the process. Actual communications correspond to τ transitions in the LTS representing the evolution of the process. Potential for communication corresponds to LTS transitions for abstractions and concretions. The following property uses potential for communication to specify the condition under which a time-to-live is active:

$$\textbf{property } l\_active(Ttl) \ \{ \ \mathbf{E}_e(\langle \overline{Ttl}(Rc) \rangle.l\_returns\_true(Rc)) \ \}$$

where *l_returns_true* is defined as:

$$\textbf{property } l\_returns\_true(Rc) \ \{ \ \mathbf{E}_e(\langle \overline{Rc}(T, F) \rangle.\mathbf{E}_e(\langle \overline{T}() \rangle)) \ \}$$

We now specify a property *c_interested*, holding when a given event *Event* is of interest to a subscription filter *Filter*:

$$\textbf{property } c\_interested(Filter, Event) \ \{ \ \mathbf{E}_e(\langle \overleftrightarrow{Filter}(Event, Rc) \rangle.returns\_true(Rc)) \ \}$$

The following safety property holds if the system cannot evolve such that an event is delivered to an active subscription without interest in the event:

$$
\begin{array}{l}
\textbf{property } \textit{of\_interest\_only}() \ \{ \\
\quad \neg \mathbf{E}_e(\langle \overleftrightarrow{\textit{subscribe}}(\textit{Filter, Callback, Ttl, Rc}) \rangle . \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{Filter}}(\textit{Event, Rc}) \rangle . \\
\qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{Rc}}(\textit{T, F}) \rangle . \\
\qquad\qquad \mathbf{E}_e(\overleftrightarrow{F}().\mathbf{E}_e(\overleftrightarrow{\textit{deliver}}(\textit{Callback, Event}))) \\
\qquad\quad) \\
\qquad) \\
\quad)\}
\end{array}
$$

We assume the callback and filter channels are unique to each subscription.

The following property should not hold on processes representing DEBSs. It specifies an event delivery occurring before a subscription to the event:

$$
\begin{array}{l}
\textbf{property } \textit{c\_delivery\_before\_subsc}() \ \{ \\
\quad \langle \overleftrightarrow{\textit{deliver}}(\textit{Callback, Event}) \rangle \ \vee \\
\quad \langle - \overleftrightarrow{\textit{subscribe}}(\textit{Filter, Callback, Ttl, Rc}) \rangle . \textit{c\_delivery\_before\_subsc}() \\
\}
\end{array}
$$

A modality $\langle - \overleftrightarrow{\textit{subscribe}}(\textit{Filter, Callback, Ttl, Rc}) \rangle . \mathcal{F}$ holds for a process P if $\exists Q : P \xrightarrow{\alpha,\kappa} Q, \alpha \neq \overleftrightarrow{\textit{subscribe}}(\textit{Filter, Callback, Ttl, Rc})$, and $\mathcal{F}$ holds for Q.

Similarly, a delivery of an event cannot happen before the publication of the event:

$$
\begin{array}{l}
\textbf{property } \textit{c\_delivery\_before\_pub}() \ \{ \\
\quad \langle \overleftrightarrow{\textit{deliver}}(\textit{Callback, Event}) \rangle \ \vee \\
\quad \langle - \overleftrightarrow{\textit{publish}}(\textit{Event}) \rangle . \textit{c\_delivery\_before\_pub}() \\
\}
\end{array}
$$

When an event is published, DEBSs attempt to notify all subscribed components. The following safety property holds if the publication of an event and the interest of a subscription on the event imply the event is delivered to the subscriber, unless the subscriber unsubscribes or the time-to-live for the subscriber is exceeded:

$$
\begin{array}{l}
\textbf{property } \textit{all\_notified}() \ \{ \\
\quad \neg \mathbf{E}_e(\langle \overleftrightarrow{\textit{subscribe}}(\textit{Filter, Callback, Ttl, SubRc}) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{\textit{SubRc}}(S) \rangle . \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(\textit{Event}) \rangle . \\
\qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{Filter}}(\textit{Event, Rc}) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{\textit{Rc}}(\textit{T, F}) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{T}() \rangle . \neg (\\
\qquad\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{deliver}}(\textit{Callback, Event}) \rangle) \ \vee \\
\qquad\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{unsubscribe}}(S) \rangle) \ \vee \\
\qquad\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{Ttl}}(\textit{TtlRc}) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{\textit{TtlRc}}(\textit{T', F'}) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{F'}() \rangle)))))))))))) \\
\}
\end{array}
$$

4.2 Unordered Delivery

Few DEBSs guarantee events are notified in the same order they are delivered [7]. The following property specifies events may not be delivered in the order they are published:

$$
\begin{array}{l}
\textbf{property } \textit{c\_unordered\_notification}() \ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(E_1) \rangle . \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\textit{publish}}(E_2) \rangle . \\
\qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{\textit{deliver}}(C_2, E_2) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{\textit{deliver}}(C_1, E_1) \rangle))) \\
\qquad) \\
\quad) \\
\}
\end{array}
$$

A DEBS does not provide ordering guarantees if the publication of E_1 followed by the publication of E_2 may be followed by the delivery of E_2 prior to the delivery of E_1.

4.3 Kell Containment Properties

It is possible to use kells and kell containment conditions when formalizing features of interest in a system. For example, one may be interested in guaranteeing events of a certain kind are only published by components executing on a certain location. Using kells to model locations, such a property can be specified as:

property *c_from_site_only*(*Site, Event*) {
$[\overleftarrow{\textit{publish}}(\textit{Event}), K_r, K_w].(\textit{Site} \in K_w) \ \wedge \ [-].\textit{c\_from\_site\_only}(\textit{Site, Event})$
}

Or,

property *c_from_site_only*(*Site, Event*) { $\neg \mathbf{E}_e(\langle \overleftarrow{\textit{publish}}(Event), K_r, \not\subseteq \{Site\}\rangle)$ }

For example, *c_from_site_only*(*waterloo, event*) holds if all events *event* are only published by components within kell *waterloo*. In *c_from_site_only*, instead of specifying:

$$[\overleftarrow{\textit{publish}}(\textit{Event}), K_r, K_w].(\textit{Site} \in K_w)$$

one may be tempted to write:

$$[\overleftarrow{\textit{publish}}(\textit{Event}), *, \supseteq \{\textit{Site}\}]$$

which is equivalent to:

$$[\overleftarrow{\textit{publish}}(\textit{Event}), *, \supseteq \{\textit{Site}\}].\mathtt{tt}$$

But such a condition just establishes that `tt` must hold for every event published at *Site*, not that every event *Event* should be published at *Site*.

In general, when it is required that every process performing an action α within a kell K must meet a condition $\mathcal{F}$ the formula to specify is:

$$[\alpha, *, \supseteq \{K\}].\mathcal{F}$$

When every action α must occur within a kell K, the condition to specify is:

$$[\alpha, *, k_w].(K \in K_w).$$

A site may be forbidden from publishing events. Such a property would be specified as:

property *c_not_at_site*(*Site*) {
$[\overleftarrow{\textit{publish}}(\textit{Event}), K_r, K_w].(\textit{Site} \notin K_w) \ \wedge \ [-].\textit{c\_not\_at\_site}(\textit{Site})$
}

Publishing components not at *Site* may need to meet extra condition $\mathcal{F}$:

property *c_extra_cond_if_not_site*(*Site*) {
$[\overleftarrow{\textit{publish}}(\textit{Event}), K_r, \not\subseteq \{Site\}].\mathcal{F} \ \wedge \ [-].\textit{c\_extra\_cond\_if\_not\_site}(\textit{Site})$
}

Event dependency conditions can also be specified based on location. For example, consider the case when publications of E_1 at *Site* are always followed by a publication of event E_2:

property *chained_events*(E_1, E_2, *Site*) {
$\neg \mathbf{E}_e(\langle \overleftarrow{\textit{publish}}(E_1), K_r, \supseteq \{Site\}\rangle.\neg \mathbf{E}(\langle \overleftarrow{\textit{publish}}(E_2)\rangle))$
}

When events are structured, sometimes it is necessary to include kell-m expressions exposing the data in the events. This allows properties such as *c_from_site_only* above to identify events of interest. For example, a kell-m process exposing the data for temperature events represented as instances of a *temperature* module (cf. Section 2.5) is:

process *expose_temp*(*T, Rc*) {
  $\overline{Rc}$(**T::temperature.temp*, **T::temperature.lat*, **T::temperature.lon*)
}

Such a process needs to be invoked for every temperature event published. In general, one can modify the *publish* call to receive the event and a channel able to expose the event's data. Alternatively, a unique process able to expose the data for all events in the system can be assumed. This generic process could be invoked by the process at channel *publish*. The actual approach will depend on the system being specified and the properties to verify.

Once the model has been modified to expose event information, a property *temp_at_location* can identify, given an event, locality requirements for the publishing process:

property *temp_at_location*(*T, Lt, Ln*) {
  $\mathbf{E}_e(\langle \overleftrightarrow{expose\_temp}(T, Rc)\rangle.\mathbf{E}_e(\langle \overleftrightarrow{Rc}(Temp, Lat, Lon)\rangle.(Lat = Lt \wedge Lon = Ln)))$
}

A more accurate check for location can be done calculating distances based on the given latitude and longitude coordinates. For simplicity, in this example we check for exact latitude and longitude coordinates.

We then modify property *c_from_site_only* to receive as parameter the coordinates of interest:

property *c_from_site_only*(*Kell, Event, Lat, Lon*) {
  $[\overleftrightarrow{publish}(Event), K_r, K_w].(temp\_at\_location(Event, Lat, Lon) \Rightarrow Kell \in K_w) \wedge$
  $[-].c\_from\_site\_only(Kell, Event, Lat, Lon)$
}

Similar properties can be specified to restrict the location of subscribed components.

4.4 Kell Passivation Properties

Besides the ability to specify properties imposing conditions on the locations of the actions, another novel aspect of our work is the ability to specify properties on the passivation of kells. For example, one may be interested in specifying that after a given event is received by a subscribed component, the subscribed component is passivated and moved to another location. Such a property is useful when specifying services that migrate based on the availability of resources. A service may receive an event indicating that the resources at its current location (for example a computer) are running low. The reaction of the subscribed event is then to migrate to another computer where resources are available.

Consider the following property:

property *service_migration*(*Callback, Service, Event*) {
  $[\overleftrightarrow{Callback}(Event), \supseteq \{Service, Computer_1\}, *].($
    $\mathbf{E}(\langle \overleftrightarrow{Service}[X], \supseteq \{Computer_2\}, *\rangle.(Computer_1 \neq Computer_2))$
  $) \wedge [-].service\_migration(Callback, Service, Event)$
}

where *Service* is the kell for the migrating service and *Callback* is the callback registered by the service for the notification of events representing low resources. For simplicity we assume the value provided in the property parameter *Event* matches such events.

The property *service_migration* holds for systems where the service *Service* is notified of a low resource event *Event* while executing at computer $Computer_1$ and its reaction is to move to a computer $Computer_2$. Notice the action $\overleftrightarrow{Service}[X]$ in the property corresponds to the passivation of the service *Service*. The passivation is performed at computer $Computer_2$.

In the previous example the services themselves decide when to migrate. Some systems may perform forced migration where a monitoring component migrates the services as reaction to events received. The event themselves may include information of which service to migrate.

Consider the following property,

$$\begin{array}{l}\textbf{property}\ \mathit{forced\_migration}(\mathit{Callback})\ \{\\ \quad [\overleftrightarrow{\mathit{Callback}}(\mathit{System})].(\mathbf{E}(\langle\overleftrightarrow{\mathit{Service}}[X], \{\mathit{Computer}_1\}, \{\mathit{Computer}_2\}\rangle))\ \wedge\\ \quad [-].\mathit{forced\_migration}(\mathit{Callback})\\ \}\end{array}$$

Callback is the channel at which the process performing the migrations receives the events. For simplicity, we assume the system to be migrated is received as the event itself. Property *forced_migration* holds when systems are migrated from one computer to another one after the event triggering the migration is received. $\mathit{Computer}_1$ is the computer where service *Service* is running, $\mathit{Computer}_2$ is the computer to which the service is migrated.

In some systems a number of servers may be dynamically adjusted according to the number of client requests. For example a web server may spawn web server processes when the number of http requests increases. When the number of http requests decreases, a number of servers may be killed. A property specification in this case is:

$$\begin{array}{l}\textbf{property}\ \mathit{reduce\_servers}(\mathit{Callback})\ \{\\ \quad [\overleftrightarrow{\mathit{Callback}}(\mathit{WebServer})].(\mathbf{E}(\langle\overleftrightarrow{\mathit{stop}}(\mathit{WebServer})\rangle.\mathbf{E}(\langle\overleftrightarrow{\mathit{WebServer}}[X]\rangle)))\ \wedge\\ \quad [-].\mathit{reduce\_servers}(\mathit{Callback})\\ \}\end{array}$$

Callback is the channel where the process in charge of killing web servers is waiting for the events. Again, for simplicity we assume the event itself is the web server to kill.

Passivation is not restricted to stopping or changing the location of a process. Passivation can also be used to alter the process executing within a kell. For example, consider the following process specification where the features provided by a process in a kell K are adjusted after an event e is received:

$$\textbf{process}\ \mathit{adjust\_features}()\ \{\ \mathit{callback}(e)\ \triangleright\ (K[X] \triangleright K[P])\ \}$$

In the example, the process for kell K is changed to P when an event e is received in the *callback* channel. Such an specification may be useful for applications executing in handheld devices. When running low on battery, one may be interested in having the applications in the handheld adapt by providing a reduced set of features. Assuming a *low-battery* event, P in the previous specification corresponds to the reduced features of application K available when the device is running at low battery. A kμ property can be specified requiring the passivation after such events are received:

$$\begin{array}{l}\textbf{property}\ \mathit{change\_features}()\ \{\\ \quad ([\overleftrightarrow{\mathit{callback}}(\mathit{Event})].\mathbf{E}(\langle\overleftrightarrow{K}[X]\rangle))\ \wedge\ [-].\mathit{change\_features}()\\ \}\end{array}$$

Event in the property corresponds to the *low-battery* event.

4.5 Optional API

Knowing the kinds of events that will be published allows DEBSs to optimize the routing of events to subscribers. Hence, several DEBSs require publishers to advertise events prior to publication (e.g., Hermes [36] and Siena [12]; details in [8]). For these systems, the core API is extended with event advertisement and the ability to update the time-to-live of both subscriptions and advertisements. Specifically, the Optional API extends the Core API with the following calls:

```
advertise(filter,ttl) → advertisement
unadvertise(advertisement)
renew_sub(subscription,ttl)
renew_adv(advertisement,ttl)
```

A `filter` in an advertisement is an expression determining the events being advertised; `ttl` determines how long should the advertisement be kept in the system.

Based on the specification for the Optional API [35], when an event is published there is no way to guarantee the announcer of the event is the same component which advertised the event. In practice, DEBSs identify the component either explicitly because a parameter is passed to the call identifying the component ([e.g., 12, 36]), or implicitly

```
process subscription(notify, renew_subsc, filter, callback, ttl) {
    var vttl := ttl in (
        notify(deliver, event) ◇ (
            if (@filter(event) and @ttl()) then
                @deliver(callback, event)
            fi
        )
        |
        renew_subsc(nttl) ◇ (
            vttl := nttl
        )
    )
}
```

Figure 5: Subscription Process for Optional API

because the API implementation attaches identifying information to the requests when communicating with the system ([e.g., 31]). In any case, to simplify the representation of advertisements we alter the `publish` call in the API by adding as parameter the advertisement. Notice without this extra parameter we would need to keep a list of advertisements and, upon event publication, search in the list for a matching advertisement.

Compared to the Core API, support for advertisements in the Optional API is the most relevant feature addition. Therefore, we further simplify our representation of the Optional API by excluding time-to-live renewal calls. The calls require the ability to update time-to-live values. For subscriptions, a possible specification is shown in Figure 5.

A kell-m model representing generic functionality of a DEBS supporting the Optional API with our simplifications is shown in Figure 6. The *subscription* process is the same as the one for *coreapi_debs* (Figure 4). The `advertise` call in the API is modelled as channel *advertise* receiving the filter, time-to-live, and a return channel. The return channel is used to return a newly created advertisement a.

When an event is published and before notifying subscribers, a process at a checks if the event being published is accepted by the advertised filter. The actual notification is specified in the advertisement process depicted in Figure 7.

Safety and liveness properties specified for the Core API in Section 4.1 apply to the Optional API as well. Safety properties *c_of_interest_only* and *c_delivery_before_subsc* can be used without modification. The property *c_delivery_before_pub* needs to be modified to include the advertisement when events are published:

$$\begin{array}{l}\textbf{property } o\_delivery\_before\_pub() \ \{\\ \quad \langle \overleftrightarrow{deliver}(Callback,\ Event) \rangle \ \vee \\ \quad \langle - \{ \overleftrightarrow{publish}(Adv,\ Event), \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ Rc) \} \rangle . o\_delivery\_before\_pub() \\ \}\end{array}$$

The following safety property, exclusive to the Optional API, requires published events to match their advertisements:

$$\begin{array}{l}\textbf{property } o\_matches\_adv() \ \{\\ \quad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ E) \rangle) \ \Rightarrow \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(Filter,\ Ttl,\ Rc) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{Rc}(Adv) \rangle . c\_interested(Filter,\ E))) \\ \}\end{array}$$

Liveness property *c_all_notified* needs to be modified to take into consideration the possibility of unadvertisements. First we specify an auxiliary property holding when an advertisement is created and an event of interest to a subscriber is published using the advertisement:

$$\begin{array}{l}\textbf{property } o\_adv\_sub\_pub(FilterA,\ TtlA,\ Filter,\ Callback,\ Ttl) \ \{\\ \quad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(FilterA,\ TtlA,\ RcA) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv) \rangle)) \ \wedge \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Filter,\ Callback,\ Ttl,\ SubRc) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{SubRc}(S) \rangle)) \ \wedge \\ \quad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event) \rangle . (c\_interested(Filter,\ Event) \wedge c\_interested(FilterA,\ Event))) \\ \}\end{array}$$

```
process optapi_debs(subscribe, unsubscribe, advertise, unadvertise, publish, deliver) {
    fresh sem (
        sem() |
        var subsc := [ ] in (
            subscribe(filter, callback, ttl, rc) ◇ (
                fresh s(
                    subscription(s, filter, callback, ttl)
                    |
                    sem() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem())
                )
            )
            |
            advertise(filter, ttl, rc) ◇ (
                fresh a(
                    rc(a) | advertisement(a, filter, ttl)
                )
            )
            |
            publish(a, e) ◇ (
                @a(subsc, deliver, e)
            )
            |
            unsubscribe(s) ◇ (
                sem() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem()
            )
            |
            unadvertise(a) ◇ 0
        )
    )
}
```

Figure 6: Optional API Specification

```
process advertisement(advnotify, filter, ttl) {
    advnotify(subsc, deliver, event) ◇ (
        if (@filter(event) and @ttl()) then
            foreach s in *subsc do @s(deliver, event) done
        fi
    )
}
```

Figure 7: Advertisement Process for Optional API

FilterA and *TtlA* are the advertisement's filter and time-to-live. *Filter*, *Callback*, and *Ttl* are the parameters of the subscription. The liveness property can now be specified as:

$$
\begin{aligned}
&\textbf{property } o\_all\_notified() \ \{ \\
&\quad o\_adv\_sub\_pub(\mathit{FilterA, TtlA, Filter, Callback, Ttl}) \Rightarrow \\
&\quad (\mathbf{E}_e(\langle \overleftrightarrow{deliver}(\mathit{Callback, Event})\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{unsubscribe}(S)\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{unadvertise}(\mathit{Adv})\rangle) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{Ttl}(\mathit{TtlRc})\rangle.\neg \mathit{returns\_true}(\mathit{TtlRc})) \ \vee \\
&\quad\ \mathbf{E}_e(\langle \overleftrightarrow{TtlA}(\mathit{TtlRA})\rangle.\neg \mathit{returns\_true}(\mathit{TtlRcA})))
\end{aligned}
$$

The kell containment and kell passivation properties specified for the Core API also apply to the Optional API. Properties where actions on the *publish* channel are specified (e.g., *c_from_site_only*, *c_not_at_site*) need to be adjusted to include the advertisement.

5 REBECA

Most DEBSs readily support the Common API, or can be easily modified to support it [35]. An example is REBECA, also known as the Rebeca Event-Based Electronic Commerce Architecture. Developed by Gero Mühl [31], REBECA is a DEBS originally proposed for the study of event routing algorithms [30, 32, 31, 33]. REBECA provides content-based event subscriptions with event replaying capabilities.

The calls in the API provided by REBECA are:

```
void publish(Event e)
void subscribe(Subscription s,EventProcessor proc)
void unsubscribe(Subscription s)
void advertise(Advertisement a)
void unadvertise(Advertisement a)
```

Subscriptions and advertisements are instances of classes `Subscription` and `Advertisement`. These classes are subclasses of a `Filter` class. Each filter instance implements a `match` method, able to identify if a given event is of interest. Parameter `proc` in the `subscribe` call is an instance of `EventProcessor`, and provides a callback `process` method to be invoked when an event is notified.

Comparing REBECA's API with the Optional API presented in the previous section, subscriptions and advertisements in REBECA correspond to filters in process *optapi_debs*, and event processors correspond to callbacks. Since REBECA does not have time-to-live support, REBECA's subscription call can be modelled with *optapi_debs*'s *subscribe* call, specifying *true* as time-to-live. Recall from Section 2.3, a process at *true*, always returns two other channels, t and f, and writes on the first one.

Later releases of REBECA support scopes, a hierarchical structuring mechanism for DEBSs [19, 20, 18]. The specification of scopes in this section showcases the use of kell-m to represent advanced features proposed for DEBSs.

A scope is a group of publishers, subscribers, and other scopes. Publishers and subscribers are called *simple components*, whilst scopes are called *complex components*. Formally, with $\mathcal{C}$ the set of simple components, $\mathcal{S}$ the set of scope (complex) components, and E a binary relation over $C = \mathcal{C} \cup \mathcal{S}$, the scope hierarchy is modelled by a directed acyclic graph $G = (C, E)$.

To support scopes, REBECA's API is extended with the following calls:

```
void joinScope(Component c,Scope s)
void leaveScope(Component c,Scope s)
```

Class `Scope` is a subclass of `Component`. `joinScope` and `leaveScope` allow a component to join and leave scopes.

Key to the concept of scopes, is the fact that visibility of events is limited to the components enclosed within a scope. Formally, visibility v is a reflexive and symmetric relation over C. Having $\mathit{super}(X) = X' | (X, X') \in E$

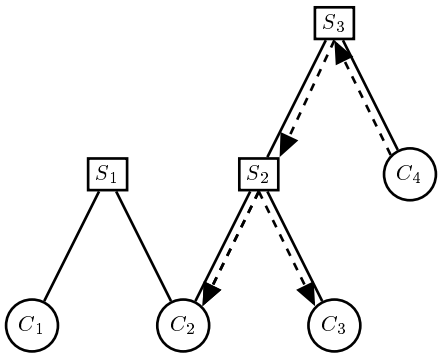

Figure 8: Sample Scope Hierarchy

denote the set of parents of X in the scope hierarchy, two components X and Y are visible to each other, $v(X, Y)$, if they both are part of a common superscope:

$$v(X,Y) \Leftrightarrow X = Y \vee v(Y,X) \vee v(X',Y) \text{ with } X' \in \mathit{super}(X)$$

When a component publishes an event, the event is delivered to all subscribed components visible to the publisher component.

For example, consider the scope hierarchy depicted in Figure 8. Boxes are used to represent scopes, and circles to represent simple components. Dashed arrows represent the propagation of an event published by C_4. Propagation occurs only to visible components: S_3, S_2, C_2 and C_3. If interested, these are the only components notified of such an event. An event published by C_2 is visible to all components, and an event notified by C_1 is only visible to S_1 and C_2.

5.1 Specification of Scopes and Event Visibility

Because a component in REBECA can belong to more than one scope, kell containment cannot be used to model scope-containment relationships. Hence, we model the scope hierarchy as a list of pairs $[S, C]$, indicating that component C is in scope S. The representation of the scope hierarchy for Figure 8 is $[[S_1, C_1], [S_1, C_2], [S_2, C_2], [S_2, C_3], [S_3, S_2], [S_3, C_4]]$.

Given a component and the scope hierarchy *shchy*, the following process at channel *super* returns the list of parent superscopes for the component:

```
super(c, shchy, accum, rc) ⋄ (
    match shchy with
            [ ] ▷ rc̄(accum)
      or s :: ss ▷ if (@member(c, @pos(s, 2))) then
                       super̄(c, ss, @cons(@pos(s, 1), accum), rc)
                   else
                       super̄(c, ss, accum, rc)
                   fi
)
```

Partial results are stored in parameter *accum*. For example, when $\overline{\mathit{super}}(C_2, \mathit{shchy}, [\,], \mathit{rc})$, the list returned in channel *rc* is $[S_1, S_2]$.

The visibility v, between two components c_1 and c_2, is determined by the following process at channel v:

```
v(c1, c2, shchy, rec, rc) ⋄ (
    if (c1 = c2 or (rec = 0 and @v(c2, c1, shchy, 1, rc))) then
        fresh t, f (rc̄(t, f) | t̄())
    else
        vsuper̄(@super(c1, shchy, [ ]), c2, shchy, rc)
    fi
)
```

Notice the process at channel v closely follows the definition of visibility $v(X, Y)$ previously presented. Parameter *rec* is used to flag the invocation where components c_1 and c_2 have been swapped. Otherwise, the recursion may never end. At channel *vsuper*, the following process checks for visibility between a component c and a list of scopes *supscopes*. Parameter *shchy* represents the scope hierarchy:

```
vsuper(supscopes, c, shchy, rc) ⋄ (
    match supscopes with
                [ ] ▷ fresh t, f (rc(t,f) | f())
        or s :: ss ▷ if (@v(s, c, shchy, 0)) then
                        fresh t, f (rc(t,f) | i())
                     else
                        vsuper(ss, c, shchy, rc)
                     fi
)
```

Using the previous processes, the specification for a DEBS with support for scopes is depicted in Figure 9. Because of the need to know the components involved in the interactions, most of the methods are extended with a parameter representing the component. Since REBECA does not support time-to-live, this feature is excluded from the specification to simplify the model.

Besides *subsc*, the list of subscribers used in the to previous models presented in this section, a list *shchy* storing the scope hierarchy is now used. Two semaphores are used as well, *sem_s* is used to guarantee exclusive access to *subsc*, and *sem_h* for *shchy*. The scope hierarchy is maintained by the processes at channels *joinscope* and *leavescope*.

The actual check for visibility is modelled in the *subscription* process as shown in Figure 10. Notice the extra parameter passed to the *notify* channel in the *subscription* process. It corresponds to the publishing component. An event is delivered if it is of interest and if the subscriber and publisher components are in a common scope.

The advertisement module, as shown in Figure 11, receives the component for which the advertisement was created, and passes that information as a parameter when communicating with subscription processes.

5.2 Safety and Liveness Properties for Scoped DEBSs

Safety properties specified for simple DEBSs specified in Section 4.1 also apply to scoped DEBSs but require modification to include the new parameters for component and to exclude time-to-live conditions:

```
property s_of_interest_only() {
    ¬E_e (⟨subscribe(Component, Filter, Callback, Rc)⟩.
        (E_e (⟨deliver(Callback, Event)⟩) ∧ ¬c_interested(Filter, Event))
    )
}

property s_delivery_before_subsc() {
    ⟨deliver(Callback, Event)⟩ ∨
    ⟨−subscribe(Component, Filter, Callback, Rc)⟩.s_delivery_before_subsc()
}
```

```
process scoped_debs(subscribe, unsubscribe, advertise, unadvertise, publish,
                    joinscope, leavescope, deliver) {
  fresh sem_s, sem_h (
    sem_s() | sem_h() |
    var subsc := [ ], shchy := [ ] in (
      subscribe(component, filter, callback, rc) ◇ (
        fresh s(
          subscription(s, component, filter, callback)
          |
          sem_s() ▷ (subsc :=_s @cons(s, *subsc)) ▷ (rc(s) | sem_s())
        )
      )
      |
      advertise(component, filter, rc) ◇ (
        fresh a(
          rc(a) | advertisement(a, component, filter)
        )
      )
      |
      publish(a, e) ◇ (
        @a(subsc, shchy, deliver, e)
      )
      |
      unsubscribe(s) ◇ (
        sem_s() ▷ (subsc :=_s @del(*subsc, s)) ▷ sem_s()
      )
      |
      unadvertise(a) ◇ 0
      |
      joinscope(component, scope) ◇ (
        sem_h() ▷ (shchy :=_s @cons([scope, component], *shchy)) ▷ sem_h()
      )
      |
      leavescope(component, scope) ◇ (
        sem_h() ▷ (shchy :=_s @del(*shchy, [scope, component])) ▷ sem_h()
      )
    )
  )
}
```

Figure 9: Scoped DEBS Specification

```
process subscription(notify, component, filter, callback) {
  notify(deliver, pubcomponent, event) ◇ (
    if (@filter(event) and @v(component, pubcomponent)) then
      @deliver(callback, event)
    fi
  )
}
```

Figure 10: Subscription Process for Scoped DEBSs

```
process advertisement(advnotify, pubcomponent, filter) {
    advnotify(subsc, shchy, deliver, event) ⋄ (
        if (@filter(event)) then
            foreach s in *subsc do @s(deliver, pubcomponent, event) done
        fi
    )
}
```

Figure 11: Advertisement Process for Scoped DEBSs

An extra property, specific to scoped DEBSs, requires delivery of events to visible components only:

$$
\begin{array}{l}
\textbf{property}\ s\_scoped\_delivery() \ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event) \rangle) \ \Rightarrow \ (\\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ Rc) \rangle) \ \wedge \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(PubComponent,\ FilterA,\ RcA) \rangle . \\
\qquad\quad \mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv) \rangle . \\
\qquad\qquad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event) \rangle . s\_visible(Component,\ PubComponent)) \\
\qquad\quad) \\
\qquad) \\
\quad) \\
\}
\end{array}
$$

Property *s_visible* is specified as:

$$\textbf{property}\ s\_visible(C_1, C_2) \ \{ \ \langle \overleftrightarrow{v} (C_1, C_2, Shchy, N, Rc) \rangle . returns\_true(Rc) \ \}$$

The liveness property for scoped DEBSs requires notification of all visible and interested components. We start by specifying an auxiliary property holding when an advertisement is created and a subscribed and interested component is visible:

$$
\begin{array}{l}
\textbf{property}\ s\_adv\_sub\_pub\_visible(Component,\ PubComponent,\ FilterA,\ Filter,\ Callback) \ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{advertise}(PubComponent,\ FilterA,\ RcA) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{RcA}(Adv) \rangle)) \ \wedge \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{subscribe}(Component,\ Filter,\ Callback,\ SubRc) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{SubRc}(S) \rangle)) \ \wedge \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv,\ Event) \rangle . (c\_interested(Filter,\ Event) \wedge c\_interested(FilterA,\ Event) \wedge \\
\qquad\qquad\qquad\qquad s\_visible(Component,\ PubComponent))) \\
\}
\end{array}
$$

An event should be delivered unless the component unsubscribes, the publisher unadvertises, or the publisher and subscriber no longer see each other:

$$
\begin{array}{l}
\textbf{property}\ s\_all\_notified() \ \{ \\
\quad s\_adv\_sub\_pub\_visible(Component,\ PubComponent,\ FilterA,\ Filter,\ Callback) \Rightarrow \\
\quad (\mathbf{E}_e(\langle \overleftrightarrow{deliver}(Callback,\ Event) \rangle) \ \vee \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{unsubscribe}(S) \rangle) \ \vee \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{unadvertise}(Adv) \rangle) \ \vee \\
\quad \neg s\_visible(Component,\ PubComponent)) \\
\}
\end{array}
$$

Kell containment and kell passivation properties specified in Section 4.3 apply to scoped DEBSs and do not require adjusting. The unordered delivery property just needs to be modified to include scope-specific parameters:

$$
\begin{array}{l}
\textbf{property}\ s\_ordered\_delivery() \ \{ \\
\quad \neg(\mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv_1, E_1) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{publish}(Adv_2, E_2) \rangle)) \Rightarrow \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{deliver}(C_1, E_1) \rangle . \mathbf{E}_e(\langle \overleftrightarrow{deliver}(C_2, E_2) \rangle))) \\
\}
\end{array}
$$

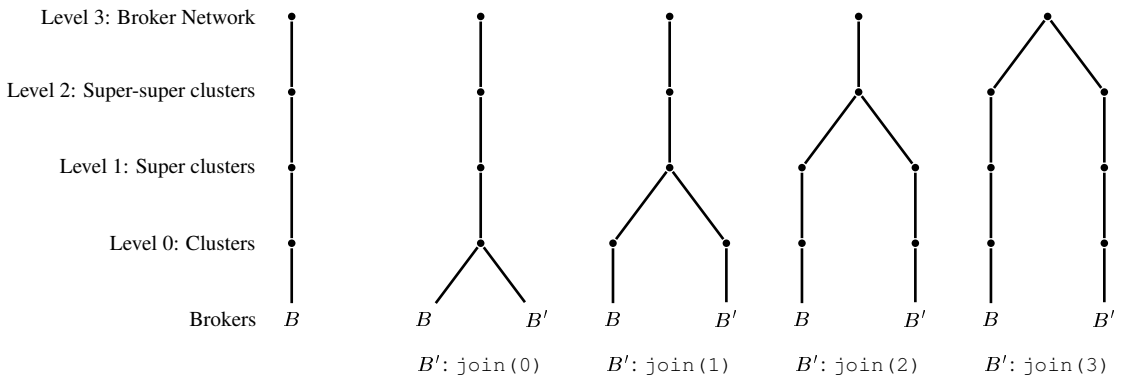

Figure 12: NaradaBrokering Broker Network Creation

6 NaradaBrokering

With the exception of properties imposing locality constraints via kell containment and kell passivation conditions (cf. Section 4.3), the use of kells in the models presented so far is concealed in the sugared constructs of hl-kell-m used in the specifications (e.g., variables, control structures, list support). In this section we show how kells can be explicitly used to model structural aspects of DEBSs. In particular we look at NaradaBrokering, a DEBS where components are organized in a hierarchical structure [34].

Besides publishers and subscribers, in NaradaBrokering there is a special group of components called *brokers*. Brokers are in charge of routing the events across the system from publishers to subscribers. Brokers are grouped in clusters, clusters are grouped in super-clusters, and super-clusters are grouped in super-super-clusters. By default the cluster containment hierarchy has four levels. Using NaradaBrokering's terminology, brokers are at leaf level, clusters are at level 0, super-clusters at level 1, and so on. A broker can only be in one cluster. The term *broker network* is used to refer to the complete broker hierarchy.

At least one broker is required when NaradaBrokering is started. Brokers in the network provide a `join(level)` operation that can be invoked by other brokers wanting to join the network. As shown in Figure 12, when `level` is 0, the requesting broker (B') joins the same cluster where the contacted broker (B) is located. When `level` is 1, a new cluster is created and the requesting broker becomes the only member of the cluster. The new cluster is located in the same super-cluster as the cluster where the contacted broker is located. If `level` is 2, a cluster and super-cluster are created. Assuming the default four levels in the cluster containment hierarchy, the maximum allowed value for `level` is 3.

Brokers running on a computer register a network port number at the computer. Once a broker has registered, other brokers can obtain a communication link with the broker by knowing the computer where the broker runs and the port where the broker is waiting for requests. This setup is typical in applications using network sockets for communication. We model this operation with the *host* process specified in Figure 13.

A list of brokers is kept at each host. Elements in the list are pairs (p, j) with p a port number and j a join channel. Join channels are the channels at which brokers wait for requests from other brokers wanting to join the broker network. As previously mentioned, ports correspond to a destination network port on the host being modelled. When the host receives a connection request on a particular port, it looks for the join channel of the broker associated to the port.

6.1 Representation of the Broker Network

We model NaradaBrokering broker networks using kells. Specifically, brokers, clusters, super-clusters and so on execute within kells that match the structure of the broker network. A broker network is setup by a process at channel *brokernw* in the process *narada* depicted in Figure 14. A process at channel *cluster* is used to represent the functionality of clusters and all groupings higher up in the hierarchy.

Given a number of clustering levels, and a channel for communication with the parent cluster for coordination of join operations, the process at channel *brokernw* returns a process with as many nested kells as one less than the number of levels specified. Inside each kell there is a cluster process as returned by the process at *cluster*. Notice

```
process host(addbroker, connect) {
    fresh sem (
    sem() |
        var brokers := [ ] in (
            addbroker(joinc, port) ⋄ (
                sem() ▷ (brokers :=_s @cons([port, joinc], *brokers)) ▷ sem()
            )
            |
            connect(port, rc) ⋄ (
                match *brokers with
                            [ ] ▷ rc(null)
                    or p :: ps ▷ if (@pos(p, 1) = port) then
                                     rc(@pos(p, 2))
                                 else
                                     @connect(ps, rc)
                                 fi
            )
        )
    )
}
```

Figure 13: Host Representation for NaradaBrokering

two return channels are used at *brokernw*. *rj* is used to return a communication channel where the cluster at level 0 can be contacted for join requests; *rc* is used to return the actual process modelling the broker network. As done in NaradaBrokering, a join request is first received by a broker which in turn forwards the request to its cluster which, if necessary (i.e., $level > 0$), forwards the request up the broker network until it is received by the process in charge of the requested join level.

At each level, a process (as returned by the process at channel *cluster*) waits for join requests at channel *join*. The first argument in the communication at *join* is the name of the kell where the broker wishing to join is executing. If the request is for level 0, the broker is transported to the cluster using kell passivation: $K[X] \triangleright K[X]$. When a broker joins the broker network it receives as return value the join channel of the cluster where it has been placed, $\overline{rc}(join)$.

If the requested level is higher than the level of the cluster, the request is handed to the parent of the cluster in the cluster containment hierarchy. Notice the parent's own join operation is available at the *pjoin* channel. If the requested level is not 0 and matches the cluster's own level, new cluster groupings are created as previously illustrated in Figure 12, and the join request is handed to the new cluster at level 0 in the newly created branch.

A broker is modelled using the following process:

```
process broker(bjoin, cjoin) {
    bjoin(kell, level, rc) ⋄ cjoin(kell, level, rc)
}
```

The process receives the join channel where the broker will wait for join requests, and the join channel for the cluster where the broker has been placed.

```
process narada(brokernw, cluster) {
    brokernw(levels, pjoin, rj, rc) ◇ (
        if (levels >= 0) then
            fresh K, join (
                @cluster(join, levels, pjoin)(pcluster) ▷ (
                    @brokernw(levels − 1, join, rj)(sclusters) ▷ rc̄(K[pcluster|sclusters])
                )
            )
        else
            rc̄(0) | rj̄(pjoin)
        fi
    )
    |
    cluster(join, level, pjoin, crc) ◇ (
        crc̄(
            join(K, blevel, rc) ◇ (
                if (blevel = 0) then
                    K[X] ▷ K[X] |
                    rc̄(join)
                elsif (blevel > level) then
                    pjoin̄(K, blevel, rc)
                else
                    fresh SK, njoin, rj (
                        @brokernw(level − 1, njoin, rj)(sclusters) ▷ (
                            SK[sclusters] | @rj(sjoin) ▷ sjoin̄(K, 0, rc)
                        )
                    )
                fi
            )
        )
    )
}
```

Figure 14: Model for NaradaBrokering Broker Network

Initially, a broker network of four levels and a single broker is setup by the following process:

```
narada(brokernw, cluster) | host(addbroker, connect) |
fresh rj (
    @brokernw(3, null, rj)() ▷ (
        rj(cjoin) ▷ (
            fresh bjoin, B1 (
                B1[@cjoin(B1, 0)(cj) ▷ (broker(bjoin, cjoin) | addbroker̄(bjoin, 1025) · · · )]
            )
        )
    )
)
```

The join channel for the only cluster at level 0 is *cjoin*. A new channel *bjoin* and kell B_1 are created. Within kell B_1 the only broker sets its join operation using $broker(bjoin, cjoin)$ and registers itself as a broker on its host at port 1025.

If a second broker knows there is a broker on the host at port 1025, it can join the broker network as follows:

$$
\begin{array}{l}
\textbf{fresh } \mathit{bjoin}, B_2(\\
\quad B_2[@\mathit{connect}(1025)(\mathit{joinc}) \triangleright @\mathit{joinc}(B_2, 0)(\mathit{cjoin}) \triangleright (\\
\qquad \mathit{broker}(\mathit{bjoin}, \mathit{cjoin}) | \overline{\mathit{addbroker}}(\mathit{bjoin}, 7070)) \\
\quad] \\
)
\end{array}
$$

In the example the new broker registers itself at port 7070. Because the broker requested a level 0 join, no new clusters are created.

6.2 Safety Property for Broker Network

Since the structure of the broker network is specified in model *narada* using kell containment, when a broker joins a cluster, the process modelling the cluster must be within *levels* of other kells, where *levels* is the number of levels in the clustering containment hierarchy (four for the broker networks depicted in Figure 12):

$$
\begin{array}{l}
\textbf{property } \mathit{cluster\_nesting}(\mathit{levels}) \ \{ \\
\quad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{cluster}}(\mathit{join},\ 0,\ \mathit{pjoin},\ \mathit{crc}) \rangle . \\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{join}}(\mathit{broker\_kell}, 0, \mathit{rc}) \rangle . \\
\qquad\quad \mathbf{E}(\langle \overleftrightarrow{\mathit{broker\_kell}}[X], \mathit{Kr}, \mathit{Kw} \rangle . (|\mathit{Kr}| = \mathit{levels})))) \\
\}
\end{array}
$$

Property *cluster_nesting* specifies a communication representing the creation of a cluster process at level zero (the cluster level), followed by a broker join request, always followed by the passivation of the broker's kell. The passivation of the broker's kell occurs when the broker is included into the broker network. The property requires the broker's passivation to be within *levels* kells.

6.3 Adaptation of Broker Network

Although not currently supported in NaradaBrokering, the brokering network could adapt by adding brokers as the load of the network increases and by reducing the number of brokers as the load decreases. In Figure 15 we show a model based on the *narada* process representing the addition of a broker every time a *"high-load"* event is received.

A variable port is used to keep track of the ports where brokers have already being assigned. A semaphore *sem* is used to guarantee exclusive access to the *port* variable. Once the broker is created (*brokernw*) and the first broker has been setup, a subscription to events *"high-load"* is obtained. We assume a DEBS with no time-to-live support for subscriptions.

Channel *loadcb* is registered as the callback in the event subscription. Upon reception of an event the process at channel *loadcb* creates a new broker, adds the broker to the broker network, and registers the broker at the next available port.

The following property specifies the creation of a broker after each *"high-load"* event is notified:

$$
\begin{array}{l}
\textbf{property } \mathit{add\_broker}() \ \{ \\
\quad \neg \mathbf{E}_e(\langle \overleftrightarrow{\mathit{cluster}}(\mathit{Join},\ 0,\ \mathit{Pjoin},\ \mathit{Crc}) \rangle . (\\
\qquad \mathbf{E}_e(\langle \overleftrightarrow{\mathit{loadcb}}(E) \rangle . \neg \mathbf{E}(\langle \overleftrightarrow{\mathit{Join}}(\mathit{NewBroker},\ 0,\ \mathit{Rc}) \rangle)))))) \\
\}
\end{array}
$$

After a cluster is created, every time an event is received in channel *loadcb*, the *join* channel of the cluster is used to create a new broker at level 0.

For simplicity we assume in this example all brokers are added on the current server, and all broker additions occur at level 0. In a real-world application, a process such as the one described would be running on each server hosting brokers. Also, as the load decreases, brokers should be removed from the network.

```
process narada_adapt {
    narada(brokernw, cluster)
    |
    host(addbroker, connect)
    |
    var port :=_s 1024 in (
        fresh rj @brokernw(3, null, rj)() ▷ (
            sem()
            |
            rj(cjoin) ▷ (
                fresh bjoin, B (
                    B[@cjoin(B, 0)(cj) ▷ (broker(bjoin, cjoin) | addbroker(bjoin, 1024) ···)]
                )
                |
                @subscribe(loadfilter, loadcb)(s) ▷ (
                    loadfilter(e, rc) ◇ (
                        fresh t, f (rc(t,f) | if (e = "high-load") then t() else f() fi)
                    )
                    |
                    loadcb(e) ◇ (
                        sem() ▷ (
                            port :=_s *port + 1 ▷ (
                                var p := *port in (
                                    sem() | fresh nbj, NB (
                                        NB[@cjoin(NB, 0)(cj) ▷ (
                                            broker(nbj, cjoin) | addbroker(nbj, *p)
                                        )]
                                    )
                                )
                            )
                        )
                    )
                )
            )
        )
    )
}
```

Figure 15: Adaptation of Broker Network

6.4 Adaptation of Broker Behaviour

We now illustrate how the behaviour of brokers can be adapted when they join the broker network. Specifically, we assume functionality related to the services that all brokers provide within the broker network is injected into each broker upon joining the network.

In Figure 14 a broker being added into the broker network is represented by the action $K[X] \triangleright K[X]$, where K is the name of the kell corresponding to the broker being added. Assuming P is the functionality that is added to each joining broker, the only required change in the *narada* model is to replace the previous action with $K[X] \triangleright K[X|P]$. This new action indicates not only a broker being moved into the network via a kell passivation, but also the adaptation of the process within the broker.

Further, if there is at least one action in P that is observable, we can specify a property requiring all brokers to exhibit the observable action from P once they join the broker network. For simplicity we will assume the observable action is a communication on channel a where the abstraction part of the communication is executed by the broker:

property *adapt_broker*() {
  $\mathbf{E}_e(\langle \overleftrightarrow{cluster}(join, 0, pjoin, crc)\rangle.$
    $\mathbf{E}_e(\langle \overleftrightarrow{join}(broker\_kell, 0, rc)\rangle.$
      $\mathbf{E}(\langle \overleftrightarrow{broker\_kell}[X]\rangle.\mathbf{E}(\langle \overleftrightarrow{a}(), \supseteq \{broker\_kell\}, *\rangle))$
)
)
}

The previous property specifies that, once a cluster at level 0 is created, join requests for the cluster are followed by the observable action. The abstraction part of the action on a must be located within kell *broker_kell* indicating that the abstraction must have executed within the broker. Similar properties can be specified for other observable actions in P.

7 Related Work

In this section we first describe representative models proposed for general implicit invocation systems (IISs) and discuss their applicability to DEBSs [16, 17]. We then describe models proposed specifically for DEBSs [31, 4, 10].

Besides the specification of the DEBS event model and the behaviour exhibited by the DEBS middleware, kμ can be used to specify the behaviour of components as it pertains to the reaction and generation of events (i.e., application behaviour). At the end of this section we review formalisms proposed for the specification of component behaviour.

7.1 Model for Synchronous Implicit Invocation

Dingel et al. propose a formal model for the compositional verification of synchronous implicit invocation systems [16]. In their model, a system S consist of a set of methods $M = \{m_1, m_2, ..., m_n\}$ with one of the methods in M being a distinguished dispatcher method *disp*. The dispatcher method *disp* stores and delivers events e from a set of events E. A binding set $B \subseteq E \times M$, associates events with the methods to be triggered when an event is announced. A method m_i is a UNITY program [13], and it is represented as a 4-tuple $m_i = (V_i, E_i, P_i, S_i)$ where V_i is the set of variables m_i accesses; E_i is the set of events m_i announces; P_i is a boolean expression over V_i, describing the initial states of m_i; and S_i is a set of guarded statements of the form $g \xrightarrow{a} x := exp$. Guard g is a boolean expression over V_i. If g holds in the current state, then action a is executed, and the value of the variable $x \in V_i$ is set to the expression *exp* over V_i. Assuming event e and predicate p, an action a can be any UNITY action, plus the following communication actions for sending and receiving messages:

- $\langle e, p\rangle!$, send event e to dispatcher if p holds
- $\langle v, p\rangle?$, an event can be received on variable v if p holds
- $\langle e, p\rangle?$, event e can be received if p holds

Communication is achieved by matching announcing ($\langle e, p\rangle!$) and consuming actions ($\langle v, p'\rangle?$ or $\langle e, p'\rangle?$). On communication a silent action τ occurs and, if the input action was $\langle v, p'\rangle?$, event e is assigned to variable v. The dispatcher

decides which methods should receive the event by looking at the binding $B \subseteq E \times M$. It is not clear from [16], if it is possible to alter binding B while the system is in operation, as it is required by the DEBS event model.

An environment method m_E, non-deterministically chooses and executes an action from a set of communication actions $\{a_1, ..., a_n\}$. First-order linear-time temporal logic is used to specify the ongoing behaviour of the system.

In order to define the semantics on an event, the environment is constrained to be a method that just announces the event. The focus is not on issues related to delivery policies and event distribution guarantees. It is not clear if by considering each event in isolation, the behaviour exhibited by the system can be fully specified. This is related to the fact that in the work of Dingel et al. an event cannot cause the announcement of other events. Another issue is the assumption of synchronicity by the subscriber: a subscriber is blocked until an event is published and sent to it.

7.2 Implicit Invocation Language

Although not proposed specifically for applications that follow the DEBS event model, in [45], Zhang et al. develop an Implicit Invocation Language IIL, and a set of source transformation tools for generating applications verifiable in the Cadence SMV model checker [27]. Properties to verify are expressed using linear temporal logic. Event bindings in IIL are static and explicitly specified. An event binding determines the components that need to be notified of the occurrence of an event. Since DEBSs support dynamic event binding (bindings between events and components that react to the events can be established or terminated dynamically), representation of DEBS-specific functionality using IIL is not supported. Specifically, the ability in DEBSs for components to introduce new types of events at run time, dynamically subscribe to and unsubscribe from events, and dynamically advertise and unadvertise events.

7.3 Logic of Event Consumption and Publication

Fenkam et al. provide operational semantics for an event based system using what the authors call LECAP: Logic of Event Consumption And Publication [17]. In this model, functional components interested in events are invoked by the system and execute only when an event of interest is published. A component P is specified by using the following syntax:

$$
\begin{aligned}
P ::= & x := val \mid P_1 \ ; \ P_2 \mid \textbf{if } cond \textbf{ then } P_1 \textbf{ else } P_2 \textbf{ fi} \mid \\
& \textbf{while } cond \textbf{ do } P_1 \textbf{ od} \mid P_1 \parallel P_2 \mid \textbf{announce } (e) \mid \\
& \textbf{skip} \mid \textbf{await } cond \textbf{ do } P_1 \textbf{ od}
\end{aligned}
$$

Where P, P_1, and P_2 are components. The functional components triggered by the publication of an event of interest are part of the component that published the event: the transitions of the triggered components are internal transitions of the publisher component. This is the main limitation of this model, with respect to its applicability in DEBSs. There is no way to specify *always running* reactive components: execution of the reactive components occur only while reacting to an event, and within the context of the publishing component.

7.4 Traces

Mühl proposes a DEBS model based on execution traces [31]. A trace of a system is a sequence $\alpha = s_1, op_1, s_2, op_2, s_3, ...$ where s_i is the state of the system in step i, and op_i is the operation taken in step i which results in the system going to state s_{i+1}. A specification $\mathcal{S}$ is a set of traces. A system is correct with respect to $\mathcal{S}$, if the system exhibits only traces that are in $\mathcal{S}$.

The sets of traces that a DEBS must exhibit are defined by using predicates for the DEBS operations, quantifiers $\exists$, $\forall$, logical operators $\wedge$ $\vee$ $\Rightarrow$ $\neg$, and temporal operators $\Box$ (always), $\Diamond$ (eventually), and $\circ$ (next).

The model proposed by Mühl assumes instantaneous notification of events, no communication failures, and requires components to be active (connected to the system) in order to be notified of events.

Another work based on traces is by Baldoni et al. [4]. In this work, and in contrast to [31], notifications are not assumed to run instantaneously. Instead of a global trace, a local trace is kept for each component in the system. Each operation in a local trace is tagged with the time, from a global clock, at which the operation is executed. The global trace H for the DEBS is defined as the collection of local traces $\langle\ h_1, h_2, h_3, \ldots\ \rangle$. A subscription interval $I(X, F)$, for a component X and a filter F, is defined as the sequence of all time-tagged operations $\langle Op(X, Y), t\rangle$ in trace h_X executed between subscription $\langle \mathit{Subs}(X, F), s\rangle$ and unsubscription $\langle \mathit{Unsub}(X, F), u\rangle$, with $s \leq t \leq u$. Hence, the time between s and u represents, for the subscriber, the time the subscription was active.

Works based on traces allow the specification of properties based on the order in which the operations should appear in the traces. With kell-m and kμ it is possible to specify conditions on the location of the actions as well as the passivation of kells. Such conditions cannot be expressed using only traces.

7.5 Other Models

In more recent work [10], Dingel et al. model the behaviour of SIENA [12], a particular DEBS. The purpose of the model is to verify *already developed* DEBS applications. The model itself is specified in BIR, the input language for the model checker Bogor [38]. Two data structures are used in the model: a FIFO communication channel between clients and the DEBS, and an event set to store the events before they are delivered. The event set is used, instead of an event queue, to model the fact that in SIENA there are no order guarantees in the order of delivered events. Only subscribe, unsubscribe and publish operations are supported. No attempts are made in this work to generalize the event model of DEBSs, nor to present a formal specification of the SIENA event model itself.

In contrast, we specify formal models that support, not only the verification of ordering guarantees but also the specification and verification of liveness and safety properties previously identified in the area, the specification and verification of kell containment and kell passivation properties and, in general, application-level properties expressible in terms of communication actions occurring in the modelled systems.

7.6 Application Behaviour

Besides the formalisms reviewed above for modelling DEBS (the middleware), other formalisms have also been proposed to model the application level behaviour exhibited by the publisher and subscriber components. In particular we look at event-condition-action rules and automata-based formalisms.

Event-Condition-Action Rules

As proposed in Rapide [37, 25, 24], and widely used in active databases [43], reactive behaviour can be modelled by ECA rules. An ECA rule specifies input events, possible composite, that must occur for the rule to be triggered. When triggered, a condition on local variables, also part of the rule is then evaluated. Based on the result of the evaluation, an action may be executed. When modelling behaviour using ECA rules, languages must be provided for specifying the input events that trigger the rule, the rule condition, and its actions. The specification of the input events is typically based on event algebras [22, 11], whilst some process calculus formalism may be used to specify the rule actions. In the case of Rapide, specification of temporal conditions in the event part of the rule is supported. Also, it is possible to specify if the events generated by the actions in a rule are independent of each other or not. To be able to decide if two or more components can be composed into a complex component, or to coordinate the interaction of several components, it is necessary to verify that the ECA rules describing the behaviour of each component are composable, and that their actions obtain the target behaviour of the composition.

Interface Automata

Interface automata, proposed by Alfaro and Henzinger [14], have been used to describe the behaviour of reactive systems [40, 41]. Interface automata are specified in an automata-based language. This language is used to capture both, input assumptions about the order in which a component reacts to events, and output guarantees about the order in which the component generates events. Interface compatibility is decided based on an *optimistic* approach. In a traditional, pessimistic approach, two components are compatible if they can be used together in all systems. In the optimistic approach proposed with Interface automata, a helpful environment is assumed: two components are compatible if they can be used together in at least one design. The advantage of the optimistic approach is a simpler model. Interface automata interact through the synchronization of input and output events. Internally, actions of concurrent automata are interleaved asynchronously.

Events are not queued in interface automata: the arrival of a message while in an state not prepared to handle the message, indicates an incompatibility between the environment and the automaton. This is in contrast to DEBSs where events are typically queued until the receiving component is in a state ready to handle the message. Similarly to the DEBS behaviour, in kell-m a write (i.e., concretion) on a channel is only matched to a read (i.e., abstraction) when both actions are ready for the communication.

Finite State Machines

In [9], Bultan et al. analyze component composition by looking at the conversations between the components. A conversation is the concatenation of all the events exchanged by the components being composed. The behaviour of the components themselves is represented by Mealy machines [28]: finite state machines where output actions can be specified in the transitions. A component is then viewed as a Mealy machine that decides, based on the received events and the events already sent, if a new event should be sent. In contrast to interface automata, Mealy machines interact asynchronously. But in order to perform the analysis of the compositions, it is required to have a global watcher that keeps track of all events as they occur. The authors start by trying to deduce global behaviour by analyzing the behaviour of the components. They find this bottom-up approach flawed and propose to perform a top-down approach instead. Their argument is that given a conversation, it is not possible to find a regular language (*global behaviour*) as its core. Bultan et al. argue this is because of the asynchronous nature of the interactions.

In the top down approach, on the other hand, they start with conversations that represent the intended *global* behaviour of the system, and construct Mealy machines that realize that conversation. Similar to the research we propose, the authors final goal is to understand component composition in distributed systems. Our approach diverges from theirs since our focus is in the specification of functionality in DEBSs, instead of deducing a global (local) behaviour based on a local (global) behaviour.

Statecharts

First proposed by Harel [21], a statechart is a graphical representation used to model reactive systems. Since Harel's original work, multiple variations, both in semantics and structure, have been proposed. UML statecharts [1] is currently the most used variation. Statecharts are, fundamentally, Mealy machines with state entry and exit reactions, hierarchical states, and parallelism. States represent processing stages of the artifact being modelled. Transitions between states are triggered by the reception of events and the evaluation of an optional condition. Actions can be executed as part of the transition. States can be represented by a substatechart or two or more substatecharts operating in parallel.

Harel statecharts assume instantaneous event processing: the reaction to an event occurs in zero time, upon notification of the event. This is not the case when dealing with DEBSs, where events take time to reach subscribed components. Another issue is the assumption that only one event may happen at a time. UML statecharts do not have this restriction, providing instead a queue of events. Both Harel and UML statecharts assume broadcasting of events, where events are globally visible to all components in the system. In contrast, in DEBSs events are only notified to subscribed components. These differences between the DEBS event model and the statechart event model make the use of statecharts to model behaviour in DEBSs and DEBS applications inadequate. Specifically, event interactions in DEBS cannot be directly specified using statechart event interactions since they are semantically different. This problem is not unique to DEBSs and arises when dealing with any IIS that has an event model incompatible with the statechart event model. The effect of these differences between event models has been the proposal of multiple, different, and sometimes incompatible, statechart variations when using them to model complex IISs ([e.g., 42, 23, 15, 5, 39]).

8 Conclusions

DEBSs provide an Application Programming Interface (API) for the interaction with components. Publisher and subscriber components use the calls in the API to announce events and to indicate to the system the events of interest. Although there is no standard DEBS API supported by most DEBSs, there is a proposal for a Common API. The Common API itself is composed of two other APIs: a Core API and an Optional API. The Core API specifies calls for the publication and subscription of events. The Optimal API extends the Core API with calls for the advertisement of events and for the renewal of subscriptions and advertisements.

Since most existing DEBSs can be easily modified to support this Common API, in this report we developed a model for generic DEBSs that follow these APIs. The model parametrizes features typically varying among different DEBS implementations such as filtering and event delivery capabilities. We showed how properties previously identified for DEBSs can be specified in the model. We also showed how the model support the specification of properties beyond the ones previously identified. In particular, we provided examples on how kells can be used in the model to specify

application-level properties that deal with locality of publishers and subscribers, as well as kell passivation. These new properties were not expressible using the formalisms previously proposed in this area.

We modelled REBECA, a particular research DEBS extended with scopes. Scopes are used to structure components in DEBSs. This model showcases the use of kell-m to model an extension to the basic DEBS features.

The previous models represent behaviour exhibited by DEBSs as it pertains to the publication, subscription, and notification of events. To showcase the use of kell-m to model other, possibly implementation-specific, features of interest that may not be exposed to publishers and subscribers, we modelled the structure of administrative components in NaradaBrokering, a particular DEBS. In NaradaBrokering specialized components, called Brokers, are in charge of the communication within the system. Brokers are grouped in hierarchical clusters. In the model, kells are used to represent such a hierarchy.

References

[1] UML 2.1.1 Superstructure Specification, Chapter 15: State Machines. `http://www.omg.org/technology/documents/formal/uml.htm`, February 2007. Object Management Group OMG. 32

[2] Objective Caml Version 3.11. `http://caml.inria.fr`, December 2008. Institut National de Recherche en Informatique et Automatique. 6

[3] Projects Utilizing NaradaBrokering. `http://www.naradabrokering.org/deployments/index.html`, November 2009. Pervasive Technology Labs at Indiana University. 1

[4] R. Baldoni, R. Beraldi, S. T. Piergiovanni, and A. Virgillito. On the Modelling of Publish/Subscribe Communication Systems. *Concurrency and Computation: Practice and Experience*, 17(12):1471–1495, 2005. 29, 30

[5] F. Barbier and N. Belloir. Component Behavior Prediction and Monitoring Through Built-in Test. In *ECBS '03: Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 17–22, Huntsville, AL, USA, April 2003. IEEE Computer Society. 32

[6] R. Blanco and P. Alencar. Distributed Event-Based System Features: Representation and Reasoning. In *IEEE International Conference on Software: Science, Technology and Engineering (SwSTE '10)*, Herzlia, Israel, June 2010. IEEE Computer Society. 12

[7] R. Blanco and P. Alencar. Event Models in Distributed Event Based Systems. In A. Hinze and A. Buchmann, editors, *Principle and Applications of Distributed Event-based Systems*. IGI Global, 2010. 13

[8] R. Blanco and P. Alencar. Semantics and Encoding of the kell-m Calculus. Technical Report CS-2010-XX, University of Waterloo, 2010. 1, 2, 4, 9, 16

[9] T. Bultan, X. Fu, R. Hull, and J. Su. Conversation Specification: A New Approach to Design and Analysis of E-Service Composition. In *WWW '03: Proceedings of the 12th International Conference on World Wide Web*, pages 403–410, New York, NY, USA, 2003. ACM Press. 32

[10] L. R. Cai, J. S. Bradbury, and J. Dingel. Verifying Distributed, Event-Based Middleware Applications Using Domain-Specific Software Model Checking. In M. M. Bonsangue and E. B. Johnsen, editors, *FMOODS*, volume 4468 of *Lecture Notes in Computer Science*, pages 44–58. Springer-Verlag, June 2007. 29, 31

[11] J. Carlson and B. Lisper. An Event Detection Algebra for Reactive Systems. In *EMSOFT '04: Proceedings of the 4th ACM International Conference on Embedded Software*, pages 147–154, New York, NY, USA, 2004. ACM Press. 31

[12] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and Evaluation of a Wide-Area Event Notification Service. *ACM Transactions on Computer Systems*, 19(3):332–383, 2001. 16, 31

[13] K. M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988. 29

[14] L. de Alfaro and T. A. Henzinger. Interface Automata. *ACM SIGSOFT Software Engineering Notes*, 26(5):109–120, 2001. 31

[15] M. S. Dias and M. E. R. Vieira. Software Architecture Analysis Based on Statechart Semantics. In *IWSSD '00: Proceedings of the 10th International Workshop on Software Specification and Design*, Washington, DC, USA, 2000. IEEE Computer Society. 32

[16] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning About Implicit Invocation. In *SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 209–221, New York, NY, USA, 1998. ACM Press. 29, 30

[17] P. Fenkam, H. Gall, and M. Jazayeri. A Systematic Approach to the Development of Event Based Applications. In *22nd Symposium on Reliable Distributed Systems (SRDS 2003)*, page 199. IEEE Computer Society, October 2003. 29, 30

[18] L. Fiege. *Visibility in Event-Based Systems*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, Darmstadt, Germany, Apr 2005. 1, 19

[19] L. Fiege, M. Mezini, G. Mühl, and A. P. Buchmann. Engineering Event-Based Systems with Scopes. In B. Magnusson, editor, *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP02)*, volume 2374 of *Lecture Notes in Computer Science*, pages 309–333. Springer-Verlag, June 2002. 1, 10, 19

[20] L. Fiege, G. Mühl, and F. C. Gärtner. Modular Event-Based Systems. *The Knowledge Engineering Review*, 17(4):359–388, 2002. 1, 10, 12, 19

[21] D. Harel. Statecharts: a Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, June 1987. 32

[22] P. Konana, G. Liu, C.-G. Lee, and H. Woo. Specifying Timing Constraints and Composite Events: An Application in the Design of Electronic Brokerages. *IEEE Transactions on Software Engineering*, 30(12):841–858, 2004. Member-Mok, Aloysius K. 31

[23] N. G. Leveson, M. P. E. Heimdahl, H. Hildreth, and J. D. Reese. Requirements Specification for Process-Control Systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994. 32

[24] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 31

[25] D. C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995. 31

[26] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying Distributed Software Architectures. In *Proceedings of the 5th European Software Engineering Conference*, pages 137–153, London, UK, 1995. Springer-Verlag. 5

[27] K. McMillan. Getting started with smv. `http://www.kenmcmil.com/tutorial.ps`, 1999. Cadence Berkeley Laboratories. 30

[28] G. H. Mealy. A Method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 35(5), 1955. 32

[29] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, Computer Laboratory, University of Cambridge, 1999. 5, 6

[30] G. Mühl. Generic Constraints for Content-Based Publish/Subscribe Systems. In C. Batini, F. Giunchiglia, P. Giorgini, and M. Mecella, editors, *Proceedings of the 9th International Conference on Cooperative Information Systems (CoopIS '01)*, volume 2172 of *Lecture Notes in Computer Science*, pages 211–225. Springer-Verlag, September 2001. 19

[31] G. Mühl. *Large-Scale Content-Based Publish /Subscribe Systems*. PhD thesis, Technische Universität Darmstadt, Darmstadt, Germany, September 2002. 12, 17, 19, 29, 30

[32] G. Mühl and L. Fiege. Supporting Covering and Merging in Content-Based Publish/Subscribe systems: Beyond Name/Value Pairs. *IEEE Distributed Systems Online (DSOnline)*, 2(7), 2001. 19

[33] G. Mühl, L. Fiege, and A. P. Buchmann. Filter Similarities in Content-Based Publish/Subscribe Systems. In *ARCS '02: Proceedings of the International Conference on Architecture of Computing Systems*, pages 224–240, London, UK, 2002. Springer-Verlag. 19

[34] S. Pallickara and G. Fox. NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids. In *Proceedings of ACM/IFIP/USENIX 2003 International Middleware Conference Middleware*, volume 2672 of *Lecture Notes in Computer Science*, pages 41–61, Berlin, Germany, 2003. Lecture Notes In Computer Science, Springer-Verlag. 1, 24

[35] P. Pietzuch, D. Eyers, S. Kounev, and B. Shand. Towards a Common API for Publish/Subscribe. In H.-A. Jacobsen, G. Mühl, and M. A. Jaeger, editors, *Proceedings of the Inaugural Conference on Distributed Event-Based Systems*, pages 152–157, New York, NY, USA, June 2007. ACM Press. 1, 10, 16, 19

[36] P. R. Pietzuch. *Hermes: A Scalable Event-Based Middleware*. PhD thesis, University of Cambridge, Queens' College, February 2004. 16

[37] Rapide Design Team. DRAFT Rapide 1.0 Pattern Language Reference Manual. `http://pavg.stanford.edu/rapide/lrms/patterns.ps`, July 1997. Program Analysis and Verification Group, Stanford University. 31

[38] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An Extensible and Highly-Modular Software Model Checking Framework. In *ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, New York, NY, USA, 2003. ACM. 31

[39] M. Ryu, J. Kim, and J. C. Maeng. Reentrant Statecharts for Concurrent Real-Time Systems. In H. R. Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications & Conference on Real-Time Computing Systems and Applications*, pages 1007–1013. CSREA Press, June 2006. 32

[40] M. Veanes, C. Campbell, W. Schulte, and P. Kohli. On-The-Fly Testing of Reactive Systems. Technical Report MSR-TR-2005-05, Microsoft Research, January 2005. 31

[41] P. Völgyesi, M. Maróti, S. Dóra, E. Osses, and Ákos Lédeczi. Software Composition and Verification for Sensor Networks. *Science of Computing Programming*, 56(1-2):191–210, 2005. 31

[42] M. von der Beeck. A Comparison of Statecharts Variants. In *ProCoS: Proceedings of the Third International Symposium Organized Jointly with the Working Group Provably Correct Systems on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 128–148, London, UK, 1994. Springer-Verlag. 32

[43] J. Widom and S. Ceri, editors. *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann, 1996. 31

[44] P. Yang, C. R. Ramakrishnan, and S. A. Smolka. A Logical Encoding of the Pi-Calculus: Model Checking Mobile Processes Using Tabled Resolution. In *VMCAI 2003: Proceedings of the 4th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 116–131, London, UK, 2003. Springer-Verlag. 9

[45] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Using Source Transformation to Test and Model Check Implicit-Invocation Systems. *Science of Computer Programming*, 62(3):209–227, 2006. 30