Current CSP Models of Concurrent Eiffel

Phillip J. Brooke
School of Computing, University of Teesside
Middlesbrough, TS1 3BA, U.K.
pjb@scm.tees.ac.uk

December 10, 2007

current-model.tex,v 1.5 2007/12/10 08:29:30 pjb Exp

This document summarises the CSP model of specific concurrency mechanisms for Eiffel, presented in [1–3]. We present a model of SCOOP in the process algebra CSP [3]. An alternative variation [1] is given by modification. A modification of both the SCOOP and alternative variations to incorporate exceptions is also given [2].

This document reflects the current state of Brooke's CSPsim model (but does not yet include exceptions).

We refer readers to Hoare's text [4] (or later texts by Roscoe [7] or Schneider [8]) for more information about CSP.

1 Order of presentation of the model

We describe four layers in Section 1.1: systems, partitions, subsystems and objects, in order to provide some structure to a complex concept.

We then construct the CSP programs for systems (which are simple compositions of other CSP programs. These systems comprise a number of components, all composed in parallel.

Section 7 contains a listing of the events used, and the processes in which they appear.

1.1 Layers

We commence with several fundamental definitions.

A *system* comprises a number of *objects*; each *object* has its own notional thread of control.

Each *subsystem* is a collection of zero or more of the objects: each of the objects is *local* with respect to any other in its subsystem. Objects in different subsystems are *separate* and are referred to in programs via the **separate** keyword. These subsystems incorporate the notions of *processor* and *handler*.

Calls between objects in the same subsystem have sequential (synchronous) semantics; calls between objects in different subsystems have the SCOOP semantics, including asynchrony, lazy waiting and preconditions-as-wait.

The overall system then comprises one or more *partitions* or processing resources, which may be physical CPUs, POSIX processes, individual threads, or any other processing model. Each partition is responsible for zero or more subsystems (and the constituent handlers *a.k.a.* processors). Here, we are borrowing some elements and terms of the Ada 95 model of distributed processing [9].

Note that we do not need to model partitions directly in this abstraction, but when we wish to prove in future work that a distributed system works correctly, we will need to model the partition communication system.

Objects remain on the same subsystem throughout their life, and subsystems remain in the same partition throughout the execution.

1.2 Alternative model

Major changes:

- No subsystems.
- Each object is concurrent.
- Objects can migrate between partitions, but we give no mechanism.

2 System

A system is modelled by the alphabetised parallel composition of its components:

$$
\begin{aligned}
SYSTEM \triangleq\ & \|_{\alpha CALLCOUNT} CALLCOUNT \\
& \|_{\alpha CALLPARAMS} CALLPARAMS \\
& \|_{\alpha CALLSEPPARAMS} CALLSEPPARAMS \\
& \|_{\alpha OBJECTLOCALS} OBJECTLOCALS \\
& \|_{\alpha RESERVATIONS} RESERVATIONS \\
& \|_{\alpha BIGLOCK} BIGLOCK \\
& \|_{\alpha ALLOBJECTS} ALLOBJECTS \\
& \|_{\alpha ALLDOCALLS} ALLDOCALLS \\
& \|_{\alpha ALLCALLSDONE} ALLCALLSDONE \\
& \|_{\alpha ALLSCHEDULERS} ALLSCHEDULERS \\
& \|_{\alpha SEQOBJECTS} SEQOBJECTS \\
& \|_{\alpha SEQSUBSYSTEMS} SEQSUBSYSTEMS \\
& \|_{\alpha RESCHEDULER} RESCHEDULER
\end{aligned} \tag{1}
$$

2.1 Parameterisation

The model is parameterised by:

- $CLASSES$, a set of all possible classes, and $FEATURES$, the names of all possible features in the system.
- $maxCallCount$, the maximum number of calls the system will execute.
- $maxInstances$, the number of distinct objects within each class.
- $maxSubsystems$, the maximum number of subsystems (other than the initial subsystem).
- $maxParams$, the number of parameters each call may record.
- $maxLocals$, the number of local variables for each object.
- $WORK$, a process that simulates the body of the features — see Section 4.9.8.
- $rootClass$ and $rootFeature$ indicate which class and feature start the execution of the system.

From these parameters, three sets are defined:

$$CALLS \triangleq \{Call.c | c \in \{0, \ldots, maxCallCount\}\} \quad (2)$$

$$OBJECTS \triangleq \{Object.l.i | l \in CLASSES, i \leftarrow \{1, \ldots, maxInstances\}\} \quad (3)$$

$$SUBSYSTEMS \triangleq \{Subsystem.j | j \in \{0, \ldots, maxSubsystems\}\} \quad (4)$$

3 Major components

3.1 Component $ALLOBJECTS$

$ALLOBJECTS$ represents the creation of objects, records their handler (a subsystem) and after creation, allows locking.

Before an object can be reserved or carry out any work, it must be created.

Creation is effected via the Eiffel **create** keyword (or semantically equivalent methods). This causes the object to be allocated to a subsystem (handler), and if applicable, a creation routine is executed to initialize the object so that it is in a state satisfying its invariant clause (if any).

There are two cases:

- The object is created as a separate object. The new object is created in a new subsystem. The partition for this new subsystem is determined by the *concurrency control file* [5, p. 971] or by other implementation-dependent means.
- The object is created as a non-separate object: we can view this object as being 'local' with respect to its parent object. The new object is in the same subsystem as its parent.

Both cases follow the same CSP representation, although new separate objects trigger the creation of a new subsystem (see Section **??** for an example).

Creation of object i on subsystem j is represented by the event $createObject.i.j.c$. c is the call causing this object to be created. There is one exception: an object of the root class is presumed to exist initially: the system bootstrap must arrange for it to be created to begin the execution overall.

So an object i is first created and can then be reserved. Additionally, the object will respond to queries regarding its handler.

$$OBJECT(i) \triangleq i = Object.rootClass.1 \Rightarrow OBJECT2(i, Subsystem.0)$$
$$i \neq Object.rootClass.1 \Rightarrow$$
$$createObject.i?j : SUBSYSTEMS?c : CALLS \rightarrow OBJECT2(i, j) \quad (5)$$

$$OBJECT2(i,j) \triangleq getHandler.i.j \rightarrow OBJECT2(i,j)$$
$$\Box\ reserve?c : CALLS.i \rightarrow OBJECT2(i,j)$$
$$\Box\ blocked?c : CALLS.i \rightarrow OBJECT2(i,j)$$
$$\Box\ free?c : CALLS.i \rightarrow OBJECT2(i,j)$$
$$\Box\ unreserved?c : CALLS.i \rightarrow OBJECT2(i,j) \quad (6)$$

$$ALLOBJECTS \triangleq |||i : OBJECTS \bullet OBJECT(i) \quad (7)$$

The rules governing reservations are in the process $RESERVATIONS$ and the actual execution of calls upon i is simulated by $ALLDOCALLS$.

3.1.1 Alternative

$ALLOBJECTS$ is modified a little more to remove references to $SUBSYSTEMS$ and $getHandler$:

$$OBJECT(i) \triangleq i = Object.rootClass.1 \Rightarrow OBJECT2(i, Subsystem.0)$$

$$
\begin{aligned}
& i \neq \mathit{Object.rootClass}.1 \Rightarrow \\
& \qquad \mathit{createObject}.i?c : \mathit{CALLS} \rightarrow \mathit{OBJECT2}(i) && (8) \\
\mathit{OBJECT2}(i) \triangleq\ & \mathit{reserve}?c : \mathit{CALLS}.i \rightarrow \mathit{OBJECT2}(i) \\
& \Box\ \mathit{blocked}?c : \mathit{CALLS}.i \rightarrow \mathit{OBJECT2}(i) \\
& \Box\ \mathit{free}?c : \mathit{CALLS}.i \rightarrow \mathit{OBJECT2}(i) \\
& \Box\ \mathit{unreserved}?c : \mathit{CALLS}.i \rightarrow \mathit{OBJECT2}(i) && (9)
\end{aligned}
$$

We could have merged the next component, $ALLSCHEDULERS$, into $ALLOBJECTS$, but to more easily compare with the existing SCOOP model, we retain the split between these processes.

3.2 Component *RESERVATIONS*

RESERVATIONS records which calls have locked which objects, and enables further locking (or offers a negative response).

We reason about objects being reserved by calls. These calls are executing routines, often on a different object.

In the model, each object is referred to simply by its name, say, i. Similarly, each call c embodies all the information about that call, such as the calling and called objects, arguments, and any other necessary book-keeping information. We assume that calls are unique in the model (a recursive routine should have different calls associated with each instance of the recursive routine).

There are two aspects to this component:

- recording the calls triggered by other calls — the *call chains*; and
- recording the calls that have reservations on each object.

3.2.1 Naive reading of reservations

Suppose that call c_1 reserves object i. c_1 now calls c_2, which also reserves object i. A simplistic view of the description in Meyer's text [5] suggests that deadlock will result: c_2 can never proceed, because it cannot reserve i; because c_2 can never proceed, nor will c_1; because c_1 never proceeds, it never releases its reservation on i. In our mechanical model (Section 5), this *no lock passing* variant is selected by the `-nopass` switch.

However, it is clear that the description above is unduly restrictive. We instead suggest that reservations be *handed-on* (or *passed-on*) to 'child' callers.

3.2.2 Call chains

As argued in subsection 3.2.1 above, our model requires information about call chains, *e.g.*, the information that call c_1 called c_2 which itself called c_3. We represent this with the function $isCaller(c, m)$, which is true if and only if c called m *either* directly *or* indirectly. Thus we have two further variations for our model, selected in the mechanical version via the `-direct` (direct lock passing) and `-indirect` (indirect lock passing) switches respectively.

For call c, C_c lists the parent calls of c.

3.2.3 Reservations-by-call

For each object i, s_i lists the calls holding a reservations on i, in the order that the reservations were made.

- A call c is in s_i if c has a reservation on i, even if it has handed-on the reservation.

- The last call in s_i is the *active reservation*: all calls with earlier reservations have handed-on the reservation to a subsequent call.

So a call c has exclusive access to object i if and only if c is the last entry in the sequence s_i.

Note that we allow c to remove itself from the list of reservations at any time (although later, we restrict this in one variant of the semantics — see Section 4.9.10): this represents the call indicating that it no longer has any interest in i (*e.g.*, it has completed). So in the case above, s_{i_1} would have the value $\langle c_1, c_2 \rangle$.

3.2.4 Available for reservation

We say that an object i is *available for reservation* if either

- the object is totally unreserved, *i.e.*, $s_i = \langle\rangle$; or
- the object is reserved by the (or a) caller higher up its call chain (modelled by $isCaller$).

We can now write down a process representing all of the objects' reservation behaviour. It has a number of arms:

- calls succeeding or failing to reserve an object;
- calls freeing an object (even if they had previously failed to reserve it);
- listing the objects reserved for a particular call;
- updating the call chain; and
- allowing enqueueing of particular calls.

$$RESERVATIONS \triangleq RESERVATIONS2(\langle\langle\rangle, \ldots\rangle, \langle\langle\rangle, \ldots\rangle) \tag{10}$$

$$\begin{aligned}
&RESERVATIONS2(\mathbf{s}, \mathbf{C}) \\
&\quad \triangleq \quad \Box\, i : OBJECTS \bullet s_i = \langle\rangle \vee isCaller(last(s_i), c) \Rightarrow \\
&\qquad\qquad reserve.c.i \rightarrow RESERVATIONS2(\mathbf{s} \oplus s_i \leftarrow s_i{}^\frown\langle c\rangle, \mathbf{c}) \\
&\qquad \Box\, i : OBJECTS \bullet s_i \neq \langle\rangle \wedge \neg isCaller(last(s_i), c) \Rightarrow blocked.c.i \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C}) \\
&\qquad \Box\, i : OBJECTS \bullet free?c : \sigma(s_i).i \rightarrow RESERVATIONS2(\mathbf{s} \oplus s_i \leftarrow s_i \downarrow \{c\}, \mathbf{c}) \\
&\qquad \Box\, i : OBJECTS \bullet unreserved?c : (CALLS \setminus \sigma(s_i)).i \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C}) \\
&\qquad \Box\, c : CALLS, p : \mathcal{P}(OBJECTS) \bullet (\forall i : p \bullet last(s_i) = c) \Rightarrow \\
&\qquad\qquad reserved.c.p \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C}) \\
&\qquad \Box\, c : CALLS, p : \mathcal{P}(OBJECTS) \bullet (\exists i : p \bullet last(s_i) \neq c) \Rightarrow \\
&\qquad\qquad notReserved.c.p \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C}) \\
&\qquad \Box\, callCount?a?b \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C} \oplus C_b \leftarrow C_a{}^\frown\langle a\rangle) \\
&\qquad \Box\, i : OBJECTS, c : CALLS \bullet last(C_c) = last(s_i) \Rightarrow \\
&\qquad\qquad addCallRemote?j?j'?c.i?f \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C})
\end{aligned} \tag{11}$$

where $t \downarrow A$ means the sequence t with all occurrences of members of the set A removed, and $\sigma(t)$ returns the set of elements contained in the sequence t.

Taking the clauses in the equation above one-by-one, they say:

- Call c can reserve i if i is totally unreserved, *i.e.*, $s_i = \langle\rangle$, or if the active reservation on i was made by the caller of c.

- If c cannot reserve i, then the model only offers the $blocked$ event.
- c can free i (for itself) at any time, provided that c is currently reserving i.
- The fourth clause handles c attempting to free i when it did not have a reservation. (We need this for the CSP treatment of $RELEASING$, which we define in Section 4.9.4.)
- The next clause allows the event $reserved.c.p$ which indicates that the objects in the set p have been reserved for call c.
- Next, the contrary event, $notReserved.c.p$, is offered if $reserved.c.p$ is not available for that choice of c and p.
- The seventh clause updates the call chains when new calls are created.
- The final clause restricts the enqueuing of separate calls: a call c can only be enqueued on object i if the call c' that created call c currently holds the active reservation on object i. This follows from the current SCOOP model whereby a call c' running on object i' can enqueue call c on the separate object i only if c' holds the active reservation on i.

3.2.5 Alternative

The last clause of 11 is (for this model) written

$$\begin{array}{l} \vdots \\ \Box\, i : OBJECTS, c : CALLS \bullet last(C_c) = last(s_i) \Rightarrow \\ \qquad addCallRemote?i'?c.i?f \rightarrow RESERVATIONS2(\mathbf{s}, \mathbf{C}) \end{array} \tag{12}$$

again, following the removal of subsystems.

3.3 Component $ALLSCHEDULERS$

$ALLSCHEDULERS$ handles the enqueueing of calls against objects and causes $ALLDOCALLS$ to start executing when appropriate.

Each subsystem has an associated scheduler. One of these is the initial or bootstrap subsystem, and is identified as $Subsystem.0$. The remaining subsystems are not initialised until they are required. Finally, the system as a whole terminates when all the subsystems agree to terminate:

$$ALLSCHEDULERS \triangleq \|_{\{terminate\}} j : SUBSYSTEMS \bullet SCHEDULER(j) \tag{13}$$

There are two aspects to the state stored in the scheduler model.

- Q, the queue of calls enqueued by separate objects. This is a sequence of triples, where each triple gives the call, the object and the feature.
- C, the chain of local calls currently being processed. This is used when a currently executing feature calls another feature on the same object or a relatively local object. This captures the sequential nature of non-separate calls.

A scheduler is either the bootstrap subsystem (and has the bootstrap call in its queue) or is initially quiescent:

$$SCHEDULER(Subsystem.0) \triangleq SCHEDULER2(\langle (Call.0, Object.rootClass.1, rootFeature)\rangle, \langle\rangle) \tag{14}$$

$$\begin{array}{rll} SCHEDULER(j) & \triangleq & \Box\, c' : CALLS \bullet newSubsystem.j.c' \rightarrow SCHEDULER2(\langle\rangle, \langle\rangle) \\ \text{where } j \neq Subsystem.0 & & \Box\, terminate \rightarrow Stop \end{array} \tag{15}$$

Next, the definition of $SCHEDULER2$:

$$\begin{aligned} SCHEDULER2(Q, C) \triangleq\ & addCallRemote?j'.j?c?i?f \rightarrow SCHEDULER2(Q^\frown\langle(c, i, f)\rangle, C) \\ & \Box\ \#Q > 0 \wedge \#C = 0 \Rightarrow schedule.j.head_1(Q).head_2(Q).head_3(Q) \rightarrow \\ & \quad SCHEDULER2(tail(Q), \langle head_1(Q)\rangle) \\ & \Box\ createObject?i.j?c \rightarrow SCHEDULER2(Q, C) \\ & \Box\ \#Q = 0 \wedge \#C = 0 \Rightarrow terminate \rightarrow Stop \\ & \Box\ addCallLocal.j?c'?i?f \rightarrow schedule.j.c'.i.f \rightarrow SCHEDULER2(Q, C^\frown\langle c'\rangle) \\ & \Box\ unschedule.j.last(C) \rightarrow SCHEDULER2(Q, front(C)) \end{aligned} \quad (16)$$

Taking equation **??** clause-by-clause:

- $addCallRemote$ events are calls enqueued from other subsystems, although the event is actually in the enqueueing call's $DOCALL$ process.
- If the subsystem is idle (*i.e.*, $\#C = 0$ meaning that no local calls are executing) and there are enqueued calls waiting ($\#Q > 0$) then the first waiting call is enqueued.
- New objects can be created on the subsystem.
- If the subsystem is idle and has no calls waiting, then it is prepared to terminate.
- If a local call enqueues a new call, then the former is immediately interrupted to process the new call.
- A call completing on the subsystem removes itself from the list of currently executing calls, C.

3.3.1 Alternative

Whereas the SCOOP model's version of $ALLSCHEDULERS$ comprises a parallel instance of $SCHEDULER$ for each subsystem, this model has an instance for each object.

$$ALLSCHEDULERS \triangleq \|_{\{terminate\}} i : OBJECTS \bullet SCHEDULER(i) \quad (17)$$

$$SCHEDULER(Subsystem.0) \triangleq SCHEDULER2(\langle(Call.0, Object.rootClass.1, rootFeature)\rangle, \langle\rangle) \quad (18)$$

$$\begin{aligned} SCHEDULER(i) \triangleq\ & createObject.i?c \rightarrow SCHEDULER2(\langle\rangle, \langle\rangle) \\ \text{where } i \neq Subsystem.0 \quad & \Box\ terminate \rightarrow Stop \end{aligned} \quad (19)$$

$$\begin{aligned} SCHEDULER2(Q, C) \triangleq\ & addCallRemote?i'?c.i?f \rightarrow SCHEDULER2(Q^\frown\langle(c, i, f)\rangle, C) \\ & \Box\ \#Q > 0 \wedge \#C = 0 \Rightarrow schedule.head_1(Q).head_2(Q).head_3(Q) \rightarrow \\ & \quad SCHEDULER2(tail(Q), \langle head_1(Q)\rangle) \\ & \Box\ \#Q = 0 \wedge \#C = 0 \Rightarrow terminate \rightarrow Stop \\ & \Box\ addCallLocal?c'.i?f \rightarrow schedule.c'.i.f \rightarrow SCHEDULER2(Q, C^\frown\langle c'\rangle) \\ & \Box\ unschedule.i.last(C) \rightarrow SCHEDULER2(Q, front(C)) \end{aligned} \quad (20)$$

3.4 The $RESCHEDULER$ process

Suppose call c is unable to reserve object j. Then it will engage in the event $blocked.c.j$. Because of this, we know that call c will soon suspend, because the only reason to obtain a lock is to start a call (or a call via a lazy lock). For each object j, $RESCHEDULER$ maintains a queue recording which calls are blocked on the object. When another call, c', frees j the head of the queue is allowed to be

rescheduled. An additional event, $reschedule.c$ was added to the model to allow $RESCHEDULER$ to prevent rescheduling by refusing this event for a call c.

The CSP program for this process is a generalised parallel of a process for each object:

$$RESCHEDULER \quad \triangleq \quad \|_{\{reschedule.c|c:CALLS\}} i : OBJECTS \bullet RESCHEDULER2(i, \langle\rangle)$$

The alphabet in the $\|$ operator ensures that a rescheduling for a call will not occur unless all the components agree.

For an individual object i with queue of blocked calls Q,

$$\begin{aligned} RESCHEDULER2(i, Q) \quad \triangleq \quad & c \notin \sigma(Q) \Rightarrow reschedule.c \rightarrow RESCHEDULER2(i, Q) \\ & \Box\ blocked?c.i \rightarrow RESCHEDULER2(i, Q^\frown\langle\rangle) \\ & \Box\ Q = \langle\rangle \Rightarrow free?c.i \rightarrow RESCHEDULER2(i, Q) \\ & \Box\ Q \neq \rangle \Rightarrow free?c.i \rightarrow RESCHEDULER2(i, tail(Q)) \end{aligned}$$

Clause-by-clause, we see that

- the rescheduling of call c is allowed unless c is in the queue;
- a call c blocking on object i is added to the queue;
- object i can always be freed even if there is nothing in the queue; and
- if there is something in the queue when object i is freed, then the head is removed from the queue, thus allowing its reschedule (at least from this object's perspective).

The CSP programs from the original model need to be slightly modified from

$$\ldots suspend.c \rightarrow \ldots$$

to

$$\ldots suspend.c \rightarrow reschedule.c \rightarrow \ldots$$

This is satisfactory for calls making a single reservation. It is less satisfactory for multiple reservations, but they do not arise in the examples we are currently examining. A general algorithm for deciding which call should proceed when multiple calls are attempting to reserve multiple partially overlapping sets of objects is beyond the scope of this paper.

Note that this does not guarantee that a rescheduled call will make progress: it might still block because the wrong call has been freed by interaction with lock-passing. In this case, further calls still need to free the object concerned, so that rescheduled call might find itself blocked immediately and then put back to the end of the queue.

4 Minor components

4.1 Component $BIGLOCK$

$BIGLOCK$ is a simple mutex to prevent races while locking objects.

While constructing our model, it became clear that a call c that needs to reserve more than one separate argument s is at risk of races with other calls making reservations. Ideally, each call would like to have an atomic test-and-set routine of the form: if all s are available, then reserve all s for c. Because the s could be on different partitions (*i.e.*, physically distant), we introduce a 'reservation lock' *for the entire system* (*i.e.*, all partitions).

This is clearly not good for performance: however, this is the semantics that the overall system should exhibit, and may be weakened in a practical implementation provided that the implementation preserves those semantics.

A CSP program for this system-wide lock is

$$BIGLOCK \triangleq biglock?c \rightarrow bigfree.c \rightarrow BIGLOCK \quad (21)$$

Then equation 36 (Section 4.9.2) is modified to read

$$RESERVING(c,s) \triangleq biglock.c \rightarrow |||i : s \bullet (reserve.c.i \rightarrow Skip \,\Box\, blocked.c.i \rightarrow Skip)\,;\, bigfree(22)$$

(We suspect that it may be appropriate to apply this lock to the release of reservations, but we have not yet investigated this point.) The alternative is a 'spin-lock' where each object repeatedly attempts to collect all the reservations it needs: this could result in livelock. However, this is primarily a low-level detail for SCOOP implementors. An obvious optimisation in this model is to add queues indicating which objects would like to reserve which other objects, and then providing a higher-level 'scheduler' of reservations. This has implications for deadlock detection and avoidance. However, we do not develop this idea further for now, instead, concentrating on offering a clear model, rather than a model of a high-performance implementation.

4.2 Component $CALLCOUNT$

$CALLCOUNT$ assigns a unique number to each call made between objects.

This is a simple labelling process that allows a call c to obtain a new call number i:

$$CALLCOUNT \triangleq CALLCOUNT2(1) \quad (23)$$

$$CALLCOUNT2 \triangleq i \leq maxCallCount \Rightarrow callCount.Call?c : \{0,\ldots,i-1\}.Call.i \rightarrow CALLCOUNT2(i+1)$$
$$i > maxCallCount \Rightarrow tooManyCalls \rightarrow Stop \quad (24)$$

4.3 Component $ALLCALLSDONE$

$ALLCALLSDONE$ records that a call has been completed.

This process simply records when a particular call has completed (via the $unschedule$ event, then offers the $doneCall$ event.

$$ALLCALLSDONE \triangleq |||c : CALLS \bullet CALLDONE(c) \quad (25)$$

$$CALLDONE(c) \triangleq unschedule?j.c \rightarrow CALLDONE2(c) \quad (26)$$

$$CALLDONE2(c) \triangleq doneCall.c?c' : (CALLS \backslash \{c\}) \rightarrow CALLDONE2(c) \quad (27)$$

4.3.1 Alternative

This requires a trivial modification to replace the reference to a subsystem in $unschedule$ with a reference to an object.

4.4 Component $CALLSEPPARAMS$

$CALLSEPPARAMS$ records the objects that should be locked for the call to proceed.

$DOCALL$ requires the identities of the separate parameters for a particular call. This book-keeping process allows the use of $setSepParam$ and $getSepParam$, with a default of the empty set.

$$CALLSEPPARAMS \triangleq |||c : CALLS \bullet CALLSEPPARAMS2(c,\{\}) \quad (28)$$

$$CALLSEPPARAMS2(c,s) \triangleq getSepParam.c.s \rightarrow CALLSEPPARAMS2(c,s) \,\Box\, setSepParam.c?s : \mathcal{P}(OBJECTS) \rightarrow CALLSEPPARAMS2(c,s) \quad (29)$$

4.5 Component $CALLPARAMS$

$CALLPARAMS$ records the actual arguments for each call.

This process allows each call to be associated with up to $maxParams$ individual parameters.

$$CALLPARAMS \triangleq |||c : CALLS, i : \{1, \ldots, maxParams\} \bullet setParam.c.i?j : OBJECTS \rightarrow CALLPARAMS2(c, i, j) \quad (30)$$

$$CALLPARAMS2(c, i, j) \triangleq getParam.c.i.j \rightarrow CALLPARAMS2(c, i, j) \Box setParam.c.i?j : OBJECTS \rightarrow CALLPARAMS2(c, i, j) \quad (31)$$

4.6 Component $OBJECTLOCALS$

$OBJECTLOCALS$ records the state of variables within an object.

$OBJECTLOCALS$ performs the same function as $CALLPARAMS$, but for the local attributes of a particular object.

$$OBJECTLOCALS \triangleq |||c : CALLS, i : \{1, \ldots, maxLocals\} \bullet setLocal.c.i?v : OBJECTS \rightarrow OBJECTLOCALS2(c, i, v) \quad (32)$$

$$OBJECTLOCALS2(c, i, v) \triangleq getLocal.c.i.v \rightarrow OBJECTLOCALS2(c, i, v) \Box setLocal.c.i?v : OBJECTS \rightarrow OBJECTLOCALS2(c, i, v) \quad (33)$$

4.7 Component $SEQOBJECTS$

$SEQOBJECTS$ is a new process (that can also be added to [3]) that causes objects of a class to be created in numerical order. This reduces the creation of spurious new states (*i.e.*, should we create $Object.2$ or $Object.3$ if we already have $Object.1$ but not $Object.2$? this process prevents $Object.3$ being created).

4.8 Component $SEQSUBSYSTEMS$

$SEQSUBSYSTEMS$ is a new process (that can also be added to [3]) that causes subsystems to be created in numerical order.

4.9 Component $ALLDOCALLS$

$ALLDOCALLS$ represents the execution of each possible call, *e.g.*, attempting the required locks, checking preconditions, starting work (which may involve the creation of further objects and subsystems, or arranging for further calls to be executed).

A particular call may also refer to the processes $ADDCALL$ (Section 4.9.9) or $ENDFEATURECALLS$ (Section 4.9.10).

We model all calls made, including calls *within* an object, *e.g.*, object i is executing routine r_1 which calls routine r_2 in the same object. Such calls execute immediately as in the sequential model of Eiffel. Similarly, calls between *relatively local* objects, *i.e.*, those on the same subsystem, also execute immediately as in the sequential model. We additionally handle calls between separate objects.

The component $ALLDOCALLS$ models every potential call in the system:

$$ALLDOCALLS \triangleq |||c : CALLS \bullet DOCALL(c) \quad (34)$$

In each case, a call follows a clear process, starting with the scheduling of a call.

4.9.1 A call is scheduled

$$\begin{aligned} DOCALL(c) \quad \triangleq \quad & \Box\, j : SUBSYSTEMS, i : OBJECTS, f : FEATURES \bullet \\ & schedule.j.c.i.f \rightarrow \\ & \quad \Box\, s : \mathcal{P}(OBJECTS) \bullet getSepParam.c.s \\ & \quad \rightarrow DOCALL1(c, j, i, f, s) \end{aligned} \tag{35}$$

where c is a member of $CALLS$. The event $getSepParam.c.s$ obtains (from a book-keeping process) the separate parameters, s, that are to be reserved for this call. Thus c is the call being executed, with i the object, f the feature and j the handling subsystem.

4.9.2 Reserving separate call arguments for a call

Before executing call c, the system must reserve each separate arguments of c. We construct a process that collects the reservations (or notes the block) for each separate argument in the set s:

$$RESERVING(c, s) \quad \triangleq \quad |||i : s \bullet (reserve.c.i \rightarrow Skip \,\Box\, blocked.c.i \rightarrow Skip) \tag{36}$$

Note that we allow the reservations to proceed in any order. If s is empty, then $RESERVING$ is defined simply as $Skip$.

The next part of call execution continues by behaving as $RESERVE$, then checking if all the required reservations have been obtained:

$$\begin{aligned} DOCALL1(c, j, i, f, s) \quad \triangleq \quad & RESERVE(c, s); \\ & \big(reserved.c.s \rightarrow DOCALL2(c, j, i, f, s) \\ & \;\Box\, notReserved.c.s \rightarrow DOSUSPEND(c, j, i, f, s)\big) \end{aligned} \tag{37}$$

4.9.3 Suspending a call

If all the reservations required cannot be made or the preconditions are not satisfied (see Section 4.9.5), then the call releases all the reservations it made, then suspends to reattempt the call later.

A simple CSP program represents suspension of a call c:

$$DOSUSPEND(c, j, i, f, s) \quad \triangleq \quad RELEASING(c, s); suspendCall.c \rightarrow DOCALL1(c, j, i, f, s) \tag{38}$$

where $RELEASING$ is described immediately below; $DOCALL$ is described in equation 35; and the event $suspendCall.c$ simply marks the fact that the call is waiting for a time.

4.9.4 Releasing reservations

We construct a counterpart process to equation 36 that frees all reservations collected:

$$RELEASING(c, s) \quad \triangleq \quad |||i : s \bullet (free.c.i \rightarrow Skip \,\Box\, unreserved.c.i \rightarrow Skip) \tag{39}$$

We need do nothing more when releasing reservations: either we have the reservation, in which case we can release it, or we never had it, so we skip over it (the $unreserved$ event). Like $RESERVING$, if s is empty, then $RELEASING$ is defined as $Skip$.

4.9.5 Preconditions of a call

A separate call does much more than make and release reservations. Once it has collected all its reservations (which might have taken several attempts) it checks the preconditions on the call (provided in a **require** clause). If the preconditions are false, then the call suspends (*i.e.*, releases all its reservations and tries again later).

Since we do not model the detailed semantics of sequential Eiffel code, we instead offer two events: $preconditionsOkay.c$ representing that the preconditions were satisfied (*i.e.*, the call can proceed); and $preconditionsFail.c$ representing the failure of the preconditions.

$$\begin{aligned} DOCALL2(c, j, i, f, s) \quad \triangleq \quad & preconditionsOkay.c \rightarrow DOCALL3(c, j, i, f, s) \\ & \Box\ preconditionsFail.c \rightarrow DOSUSPEND(c, j, i, f, s) \end{aligned} \tag{40}$$

4.9.6 Completing a call

Once the preconditions are established, the call c can continue. It does this by behaving as $WORK$, which models individual feature bodies, then subsequently releases its reservations and indicates to the scheduler that it has completed. As each individual call c only occurs once, the process becomes $Stop$.

$$DOCALL3(c, j, i, f, s) \quad \triangleq \quad WORK(c, i, f); RELEASE(c, s); unschedule.j.c \rightarrow Stop \tag{41}$$

4.9.7 Alternative

We can modify $ALLDOCALLS$ to remove the subsystems.

$$ALLDOCALLS \quad \triangleq \quad |||c : CALLS \bullet DOCALL(c) \tag{42}$$

$$\begin{aligned} DOCALL(c) \quad \triangleq \quad & \Box\ i : OBJECTS, f : FEATURES \bullet \\ & \quad schedule.c.i.f \rightarrow \\ & \quad \Box\ s : \mathcal{P}(OBJECTS) \bullet getSepParam.c.s \\ & \quad \rightarrow DOCALL1(c, i, f, s) \end{aligned} \tag{43}$$

$$\begin{aligned} DOCALL1(c, i, f, s) \quad \triangleq \quad & RESERVE(c, s); \\ & \big(reserved.c.s \rightarrow DOCALL2(c, i, f, s) \\ & \ \Box\ notReserved.c.s \rightarrow DOSUSPEND(c, i, f, s)\big) \end{aligned} \tag{44}$$

$$DOSUSPEND(c, i, f, s) \quad \triangleq \quad RELEASING(c, s); suspendCall.c \rightarrow DOCALL1(c, i, f, s) \tag{45}$$

$$\begin{aligned} DOCALL2(c, i, f, s) \quad \triangleq \quad & preconditionsOkay.c \rightarrow DOCALL3(c, i, f, s) \\ & \Box\ preconditionsFail.c \rightarrow DOSUSPEND(c, i, f, s) \end{aligned} \tag{46}$$

$$DOCALL3(c, i, f, s) \quad \triangleq \quad WORK(c, i, f); RELEASE(c, s); unschedule.i.c \rightarrow Stop \tag{47}$$

4.9.8 $WORK$

If the preconditions are true, then the work of the call can go ahead, represented by the CSP program $WORK(c, i, f)$ for call c of feature f on object i:

$$WORK(c, i, f) \quad \triangleq \quad startWork.c.i.f \rightarrow \ldots; endWork.c.i.f \rightarrow Skip \tag{48}$$

where the events $startWork$ and $endWork$ represent time passing during the work. The ellipsis represents the body of a particular feature. See the examples in Section **??** for the means of translation from Eiffel.

4.9.9 $ADDCALL$

As part of $WORK$, the call may make other local and separate calls.

We represent this by the CSP program $ADDCALL(c, c', i, j, f)$, where c is the call enqueuing call c'; the object making the call is i and the target object is j and the desired feature is f:

$$\begin{aligned} ADDCALL(c, c', i, j, f) \quad \triangleq \quad & getHandler.i?h_i : SUBSYSTEMS \rightarrow \\ & getHandler.j?h_j : SUBSYSTEMS \rightarrow \\ & \Big(h_i = h_j \Rightarrow \big(addCallLocal.h_i.c'.j.f \rightarrow doneCall.c'.c \rightarrow Skip \big) \\ & \Box\ h_i \neq h_j \Rightarrow \big(addCallRemote.h_i.h_j.c'.j.f \rightarrow Skip \big) \Big) \end{aligned} \tag{49}$$

In the first case, both the source and target objects are handled by the same subsystem, so the call is local. Call c waits for the newly enqueued call c', and $ALLSCHEDULERS$ arranges for c' to start immediately.

In the second case, the call is separate. If $WORK$ for c needs the result synchronously (*i.e.*, c′ represents a function call) then a synchronisation on $doneCall.c'.c$ should be included. Note that separate calls can only be enqueued if the target is locked by the current call: this is enforced in $RESERVATIONS$.

4.9.10 $ENDFEATURECALLS$

Separate calls do not immediately execute: they are queued on the called object's handler. We now give two variants that capture different semantics. The difference relates to when reservations may be released when compared with the execution of calls made with that reservation.

1. We argue that a reservation should be released as soon as possible (see Section **??**).
2. Nienaltowski [6] argues that a reservation should not be released until all work queued by the caller on the callee has been completed.

Therefore, as part of $WORK$, the call should collect the various calls it makes, say, $c_1, c_2, \ldots$ and collect them into a set C. The final action of $WORK$ before $endWork.c.i.f$ should be to behave as $ENDFEATURECALLS(c, C)$.

We give the two variants of $ENDFEATURECALLS$:

1. The authors' preferred variant imposes no restriction on when the call c may terminate in respect to its child calls given in the set C:

$$ENDFEATURECALLS(c, C) \quad \triangleq \quad Skip \tag{50}$$

2. An encoding of Nienaltowski's desired semantics simply waits to engage in $doneCall.c'.c$ events before allowing completion:

$$ENDFEATURECALLS(c, C) \quad \triangleq \quad |||c' : C \bullet doneCall.c'.c \rightarrow Skip \tag{51}$$

   If C is empty, then this variant is simply $Skip$.

The two variants are selected in the mechanical implementation with the `-nowait` and `-wait` switches respectively.

4.9.11 Alternative: Adding calls and lazy locks

We must modify $ADDCALL(c, c', i, j, f)$, which represents call c on object i enqueuing a call c' of feature f on object j. The SCOOP version distinguishes between local calls (*i.e.*, calls to the Current or or an object on the same subsystem) and remote (calls to an object on another subsystem) whereas this model distinguishes between calls to Current or other objects. Aliasing is implicitly handled in both cases since we deal directly with the objects, not the Eiffel entities (*i.e.*, variables).

$$\begin{aligned} ADDCALL(c, c', i, j, f) \quad \triangleq \quad & i = j \Rightarrow \big(addCallLocal.c'.j.f \rightarrow doneCall.c'.c \rightarrow Skip\big) \\ & \Box\ i \neq j \Rightarrow \big(addCallRemote.i.c'.j.f \rightarrow Skip\big) \end{aligned}$$

Both $WORK$ and $ENDFEATURECALLS$ are unmodified from the SCOOP version.

The alternative model in this paper allows for lazy locks. This is really a syntactic mechanism rather than semantic. This means that the implicit wrapper needs to be explicitly included in the CSP model of a particular program. Thus there is no other special modification to the model to accommodate lazy locks.

5 Mechanical implementation

See work on CSPsim elsewhere.

6 Availability of code and examples

CSPsim is available from
http://www.scm.tees.ac.uk/p.j.brooke/cspsim/ for particular versions of the GNAT compiler.
Some example source code, state output, and dot files are available at http://www.scm.tees.ac.uk/p.j.brooke/ce1/.

7 Glossary of events

We list the events and the processes that engage in those events.

7.1 Alternative

Compared to the listings below, the alternative has the following changes:

- $addCallLocal$,$createObject$ and $schedule$ events have the reference to $SUBSYSTEMS$ removed.
- $addCallRemote$ also has references to $SUBSYSTEMS$ removed, but we add a component i' indicating which object enqueued the call.
- The $getHandler$ and $newSubsystem$ events are no longer needed.
- $unschedule$ refers to the object running the call rather than a subsystem.

7.2 Visible events

The visible events are those that are likely to be interesting for the analysis of SCOOP systems.

Event(s)	Types	Components involved	Meaning

$addCallLocal.j.c.i.f$	$j : SUBSYSTEMS$ $c : CALLS$ $i : OBJECTS$ $f : FEATURES$	$ALLDOCALLS$ $ALLSCHEDULERS$	A call to feature f on object i is enqueued on subsystem j as call c
$addCallRemote.j.j'.c.i.f$	$j : SUBSYSTEMS$ $j' : SUBSYSTEMS$ $c : CALLS$ $i : OBJECTS$ $f : FEATURES$	$ALLDOCALLS$ $ALLSCHEDULERS$ $RESERVATIONS$	A call to feature f on object i is enqueued on subsystem j' as call c by a call running on subsystem j
$biglock.c$ $bigfree.c$	$c : CALLS$	$BIGLOCK$ $ALLDOCALLS$	System-wide reservations for call c
$createObject.i.j.c$	$i : OBJECTS$ $c : CALLS$ $j : SUBSYSTEMS$	$ALLOBJECTS$ $ALLDOCALLS$ $ALLSCHEDULERS$	Call c causes subsystem j to create object i
$preconditionsOkay.c$ $preconditionsFail.c$	$c : CALLS$	$ALLDOCALLS$	Simulates the evaluation of preconditions for call c
$reserve.c.j$ $blocked.c.j$ $free.c.j$ $unreserved.c.j$	$c : CALLS$ $j : OBJECTS$	$RESERVATIONS$ $ALLOBJECTS$ $ALLDOCALLS$	Lock/free/*etc.* object j for call c
$schedule.j.c.i.f$	$j : SUBSYSTEMS$ $c : CALLS$ $i : OBJECTS$ $f : FEATURES$	$ALLDOCALLS$ $ALLSCHEDULERS$	Call c (feature f on object i) is to start executing on subsystem j
$startWork.c.i.f$ $endWork.c.i.f$	$c : CALLS$ $i : OBJECTS$ $f : FEATURES$	$ALLDOCALLS$	Start and end of call c, where the call is feature f operating on object i
$terminate$		$ALLSCHEDULERS$	All subsystems are prepared to terminate
$tooManyCalls$		$CALLCOUNT$	Too many calls for this instance

7.3 Hidden events

We hide a number of events in $SYSTEM$, mostly those dealing with book-keeping, rather than those of particular interest.

Event(s)	Types	Components involved	Meaning
$callCount.c.i$	$c : CALLS$ $i : CALLS$	$CALLCOUNT$ $RESERVATIONS$, $ALLDOCALLS$	Call c gets assigned a new call i
$doneCall.a.b$	$a : CALLS$ $b : CALLS$	$ALLDOCALLS$ $ALLCALLSDONE$	Call b is told that call a has completed
$getHandler.i.j$	$i : OBJECTS$ $j : SUBSYSTEMS$	$ALLOBJECTS$ $ALLDOCALLS$	Reports that object i is handled by subsystem j
$newSubsystem.j.c$	$j : SUBSYSTEMS$ $c : CALLS$	$ALLDOCALLS$ $ALLSCHEDULERS$	Call c causes subsystem j to be initialised
$reserved.c.s$ $notReserved.c.s$	$c : CALLS$ $s : \mathcal{P}(OBJECTS)$	$RESERVATIONS$ $ALLDOCALLS$	Has call c reserved each object in the set s?

$setLocal.c.i.j$ $getLocal.c.i.j$	$c : CALLS$ $i : \{1, \ldots, maxLocals\}$ $j : OBJECTS$	$OBJECTLOCALS$ $ALLDOCALLS$	Set/get ith local for call c with value j
$setParam.c.i.j$ $getParam.c.i.j$	$c : CALLS$ $i : \{1, \ldots, maxParams\}$ $j : OBJECTS$	$CALLPARAMS$ $ALLDOCALLS$	Set/get ith parameter for call c with value j
$setSepParam.c.s$ $getSepParam.c.s$	$c : CALLS$ $s : \mathcal{P}(OBJECTS)$	$CALLSEPPARAMS$ $ALLDOCALLS$	Set/get s, the set of locks to be obtained for call c
$suspend.c$	$c : CALLS$	$ALLDOCALLS$	Call c is suspending
$unschedule.j.c$	$j : SUBSYSTEMS$ $c : CALLS$	$ALLDOCALLS$ $ALLCALLSDONE$ $ALLSCHEDULERS$	Call c on subsystem j has finished

References

[1] Phillip J. Brooke and Richard F. Paige. An alternative model of concurrency for Eiffel. In Richard F. Paige and Phillip J. Brooke, editors, *Proc. First International Symposium on Concurrency, Real-Time, and Distribution in Eiffel-like Languages (CORDIE)*, number YCS-TR-405. University of York, July 2006.

[2] Phillip J. Brooke and Richard F. Paige. Exceptions in concurrent Eiffel. *Journal of Object Technology*, November/December 2007. To appear.

[3] Phillip J. Brooke, Richard F. Paige, and Jeremy L. Jacob. A CSP model of Eiffel's SCOOP. *Formal Aspects of Computing*, 2007. To appear.

[4] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International UK, 1985.

[5] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 2nd edition, 1997.

[6] P. Nienaltowski, editor. *Proceedings of First SCOOP Workshop*, February 2005.

[7] A.W. Roscoe. *The Theory and Practice of Concurrency*. Series in Computer Science. Prentice Hall, 1998.

[8] Steve Schneider. *Concurrent and Real-time Systems*. Wiley, 2000.

[9] T. Taft and R. A. Duff, editors. *Ada 95 Reference Manual*. Number 1246 in Lectures Notes in Computer Science. Springer-Verlag, 1997.