Cameo: an alternative model of concurrency for Eiffel

Phillip J. Brooke, Richard Paige

DOI 10.1007/s00165-008-0096-1
BCS © 2008
Formal Aspects of Computing (2009) 21: 363–391

Formal Aspects of Computing

Cameo: an alternative model of concurrency for Eiffel

Phillip J. Brooke[1] and Richard F. Paige[2]

[1] School of Computing, University of Teesside, Middlesbrough, TS1 3BA, UK. E-mail: pjb@scm.tees.ac.uk
[2] Department of Computer Science, University of York, Heslington, York, YO10 5DD, UK. E-mail: paige@cs.york.ac.uk

Abstract. We present a new concurrency model for the Eiffel programming language. The model is motivated by describing a number of semantic problems with the leading concurrency model for Eiffel, namely SCOOP. Our alternative model aims to preserve the existing behaviour of sequential programs and libraries wherever possible. Comparison with the SCOOP model is made. The concurrency aspects of the alternative model are presented in CSP along with a model of exceptions. The results show that while the new model provides increased parallelism, this comes with the price of increased overhead due to lock management.

Keywords: Concurrency; Formal methods; Programming languages; Eiffel; Alternative to SCOOP; Asynchronous exceptions

1. Introduction

Eiffel is a sequential object-oriented programming language [Mey97] to which concurrency has been introduced in two ways: by addition of a threading library (similar to Java), providing low-level mechanisms for synchronisation and mutual exclusion; and by the Simple Concurrent Object-Oriented Programming (or SCOOP) mechanism [AENV04]. SCOOP is a syntactically simple extension to Eiffel. It adds one new keyword, **separate**, to the language. This keyword can be applied to classes, arguments passed to methods and variables. **separate** is overloaded syntax. A class that has been tagged as **separate** will execute in its own conceptual thread. A variable or method argument tagged as **separate** specifies synchronisation points. Thus, SCOOP is interesting because it abstracts away from the need to deal with low-level concurrency and distribution mechanisms such as mutexes and semaphores (for the programmer). However, the interplay between **separate**, and Eiffel's other constructs—particularly its support for Design-by-Contract [Mey97]—is complex.

As part of our efforts in understanding the underlying complexity of SCOOP, and the interplay between SCOOP's conceptual notion of thread of control, synchronisation and other Eiffel constructs, we developed a CSP model of the concurrent aspects of SCOOP (i.e., omitting state change behaviour) [BPJ07], and analysed this model using both FDR2 and a new, bespoke tool, CSPsim [BP07b]. The results of the analysis helped us identify appropriate semantics for elements of SCOOP where there was a degree of choice (e.g., when locks on objects should be released). It also helped us identify a number of ambiguities and problems in SCOOP (which we discuss in more detail in Sect. 3). To resolve these ambiguities and problems, in this paper we propose an alternative concurrency model for Eiffel which, we believe, is conceptually simpler than SCOOP, while offering increased opportunity for parallelism; the trade-off is an increased overhead for managing locks on objects, which we discuss later in the paper.

Correspondence and offprint requests to: P. J. Brooke, E-mail: pjb@scm.tees.ac.uk

Additionally, sequential, synchronous exceptions are relatively easy to model, but asynchronous exceptions are more problematic due to broken chains of calls. To better understand the behaviour of systems in the presence of exceptions, we implement a model of exceptions, both synchronous and asynchronous.

1.1. Related work

An incomplete prototype of the SCOOP mechanism was implemented by Compton [Com00] by building upon the GNU SmartEiffel compiler and run-time system. A prototype preprocessor implementation was constructed by Fuks et al. for ISE Eiffel [FOP04].

More recently, Nienaltowski et al. [Nie05, Nie07] have produced the most complete implementation of SCOOP to date. This (in common with other prototypes) is a preprocessor that rewrites SCOOP-using classes. As a result of Nienaltowski's thesis [Nie07], improvements have been made to the SCOOP mechanism. In particular, Nienaltowski distinguishes between SCOOP_97, as described in [Mey97], and the current version of SCOOP, which is best presented in [Nie05, Nie07], and which is implemented in the current SCOOP prototype. We clearly distinguish the version of SCOOP we are referring to, when relevant. However our analysis of SCOOP's deficiencies and ambiguities is based on SCOOP_97; only some of these deficiencies have been dealt with in [Nie07]. The same work and others [NMO08] deal with the analysis of contracts in concurrent systems, and provide some initial steps towards proof rules in these systems.

Ostroff et al. [OTHS08] present a fair transition systems model of parts of SCOOP_97, and use it as the basis for a verification technique for checking a number of properties of SCOOP programs. In particular, they also explore testing techniques (via the SpecExplorer tool, based on Spec#) for SCOOP, and demonstrate that some properties cannot be automatically checked for SCOOP, particularly deadlocks.

More broadly, work that combines object-oriented specification, concurrency, and formal specification is predominantly focused on Circus [WC02] and TCOZ [MD02]. Circus integrates a guarded command language with Z and CSP and also focuses on refinement; refinement in an Eiffel context has only been considered in the context of sequential (and timed) programs [PO04]. Tools are beginning to become available for Circus, particularly to support translation to executable programming languages [FC06]. Our alternative concurrency model for Eiffel must not break existing, sequential Eiffel programs. Thus, work on Circus is related to our attempts to formalise concurrency models for Eiffel, but our need to integrate with existing tools and programming constructs implies additional constraints.

TCOZ is a formal specification language that combines Timed CSP and Object-Z; reasoning rules have been defined for TCOZ, but the language itself is not executable. Tool support is predominantly through the Community Z Tools [CZT] initiative.

1.2. Overview

In Sect. 2, we outline Eiffel and SCOOP. We summarise our critique of SCOOP in Sect. 3; a more detailed critique can be found in [BP06].

We present our alternative model in Sect. 4, an example in Sect. 5, along with a CSP sketch in Sect. 6 and a preliminary mechanical implementation in Sect. 7, and discuss our contribution in Sect. 8 before concluding in Sect. 9.

2. Eiffel

The Eiffel object-oriented programming language [Mey97, ECM05] has evolved over many years to provide a rich set of specification and implementation constructs for building systems. In particular, it supports Design by Contract [Mey97], a precise means for documenting classes and relationships between classes in such a way that the documentation is checked at run-time and thus provides additional guarantees as to the correctness of software.

Eiffel's syntax is similar to that of other object-oriented programming languages. A short example of an Eiffel class is shown in Fig. 1, taken from [Mey97]. The class BANK_ACCOUNT inherits from ACCOUNT (thus defining a subtyping relationship). It provides several attributes, e.g., balance and deposits_made; these features are publicly accessible to any client.

```
class BANK_ACCOUNT
inherit ACCOUNT
feature
  balance: INTEGER -- Current balance
  deposits_made: INTEGER is
    -- Number of deposits made since opening
  do
    ..
  end

feature
  deposit (sum: INTEGER)
    -- Add sum to account.
  require
    sum >= 0
  do
    balance := balance + sum
    deposits_made := deposits_made + 1
  ensure
    deposits_made = old deposits_made + 1
    balance = old balance + sum
  end

feature { NONE }
  my_deposits: LINKED_LIST[DEPOSIT]
  -- List of deposits since account's opening.

invariant
 (my_deposits /= Void) implies (balance = my_deposits.sum)
 (my_deposits = Void) implies (balance = 0)
end
```

Fig. 1. An Eiffel class

Attributes are by default of reference type; a reference attribute either points at an object on the heap, or is Void. The class provides a routine, deposit, which adds money to the bank account. These features may have preconditions (**require** clauses) and postconditions (**ensure** clauses). The former must be true when a routine is called (i.e., it is established by the caller) while the latter must be true when the routine's execution terminates. Finally, the class has an invariant, specifying properties that must be true of all objects of the class at stable points in time, i.e., before any valid client call on the object.

For more details on the language, see [Mey97] or the more recent [ECM05]. Of note from [ECM05], references types are, by default, *attached* and may not be Void. Only *detachable* types are permitted to be Void. However, this has no direct relevance to the work presented here.

2.1. SCOOP

Simple Concurrent Object-Oriented Programming (SCOOP) provides an abstract concurrency model for Eiffel [Mey97]; it was designed to minimally disrupt existing code. Indeed, existing (sequential) Eiffel programs are not broken by inclusion in a SCOOP program, though additional care may need to be taken when using sequential libraries in a concurrent setting (as is normally the case). SCOOP introduces concurrency to Eiffel by addition of the keyword **separate**. **separate** may be applied to the declaration of a class, the declaration of an entity or a formal routine argument. Examples of these three types of applications are:

```
separate class ROOT
x: separate PROCESS
f(y:separate PROCESS)
```

The normal way in which Eiffel programs operate is that objects are created, and then routines are called on these objects to carry out computations. This model of operation applies to SCOOP programs as well. The fundamental change is that when an object is created from a **separate** class, it has its own conceptual thread of control (although this is complicated shortly when we discuss 'processors', which are implementation constructs).

Access to a **separate** object indicates different semantics to the usual sequential Eiffel model. In the sequential model, feature calls on an object cause execution to switch to the called object, whereupon the feature executes, and (perhaps after storing a result), execution continues at the next instruction. In SCOOP, routine calls to x or y (in the examples above) are asynchronous. The called object can queue multiple calls, allowing callers to continue concurrent execution.

Function calls and reference to attributes are synchronous, but these may be subject to lazy evaluation [Car93]. Additionally, races are prevented by the convention that a **separate** formal argument causes the object to be exclusively locked during that feature call. This locking is also known as *reservation* in SCOOP. When locks should be *released* is an important issue in maximising parallelism in SCOOP.

SCOOP introduces the notion of a *processor*. When a **separate** object is created, a new processor is also created to handle its processing. This processor is called the object's *handler*. Thus, a processor is an autonomous thread of control capable of supporting *sequential* instruction execution [Mey97]. A system in general may have many processors associated with it.

Compton [Com00] introduces the notion of a *subsystem*—a model of a processor and the set of objects it operates on—to distinguish the execution of sequential and concurrent programs. In his terminology, a **separate** object is any object that is in a different subsystem. In this paper, we will refer to subsystems rather than processors (to avoid possible confusion with real CPUs).

2.2. Exceptions in sequential Eiffel

Exceptions in sequential Eiffel can occur when an assertion is violated (e.g., a routine is called with precondition **false**), a called routine fails, an interrupt is sent by the operating system, or an operation fails (e.g., creation of a new object fails, arithmetic overflow occurs).

Exceptions are handled by so-called **rescue** clauses. A rescue clause is attached to a routine (after the postcondition) and describes a sequence of instructions that should be executed when the routine is to recover from an undesirable run-time event. One new instruction that can be included in rescue clauses is **retry**, which is a directive to re-start execution of the routine body from the beginning.

Exceptions propagate upwards through the call chain until either a rescue clause executes **retry**, or the root feature fails and the whole system stops.

2.3. Preconditions and waiting

Eiffel uses **require** and **ensure** clauses for specifying the pre- and postcondition of features. If a precondition evaluates to false, an exception is raised and must be processed by the client of the call; a postcondition failure indicates an error in the feature itself, and the feature can define an exception handler to deal with such failures.

In SCOOP, a **require** clause on a feature belonging to a **separate** object specifies a *wait* condition: if a feature's **require** clause evaluates to false, the processor associated with that object waits until the precondition is true before proceeding with feature execution. Thus, SCOOP uses the precondition as a *guard*. The intent of this mechanism is that another object may call routines on x causing the wait condition to subsequently evaluate to true. This also admits that the entire system may become deadlocked: the run-time system has a duty to detect such circumstances.

3. Difficulties with SCOOP

As it is currently understood [Mey97], SCOOP suffers from under-specification and a number of complications or ambiguities [BP06]:

1. Chains of calls could cause deadlock unless reservations can be 'passed-on'. For example, if $a.f$, i.e., feature f of the object attached to entity a, wishes to call feature $b.g$, then the current model of SCOOP requires that
   (a) a must have a reservation on b before it can call g in b.
   (b) b needs to obtain reservations on each (separate) argument in the call before it can execute g.

Now suppose that $b.g$ is a function call and that $a.f$ requires the result of that call before it can proceed past a certain point. Moreover, a also holds the reservation on a further separate object s and s is one of the arguments to $b.g$. In this case, deadlock will result:

- $a.f$ cannot make progress because it requires $b.g$ to return its result; but
- $b.g$ cannot start because $a.f$ has reserved s; and
- $a.f$ will not release s until it has finished.

This is clearly undesirable, and is a serious problem since object-oriented programming often includes chains of calls passing on the same argument. [BPJ07] proposed that this problem is solved by allowing features to 'pass on' their reservations to calls they themselves make; while they have passed on their reservation, they cannot make progress that requires it. This mechanism is 'lock-passing'. Nienaltowski investigates this and implemented a variant of the mechanism in the SCOOPLI pre-processor [Nie07].

2. It is unclear when reservations should be released. Should the release happen as soon as the last reference to the reserved object has been processed, as soon as the end of the reserving feature is reached, or only when all calls made by the feature have also finished? [BPJ07] argues that the last choice worsens progress and increases the risk of deadlock, but makes no view on the first two choices.
3. Although SCOOP offers high parallelism, implicit reservations and the use of subsystems to group objects reduces parallelism. Each object could be viewed as parallel thread in its own right, yet the current formulation restricts progress.
4. It is unclear when blocked preconditions should be rescheduled, particularly when there is contention for a single resource. Meyer's book [Mey97] suggests a default first-in-first-out policy for identifying the order in which calls proceed, with an option for the use of library mechanisms to override this default. However, priorities and issues of liveness require a general mechanism.
5. Priorities and call queues interact badly. The call queues are strict FIFOs to prevent races. There is no obvious way for an 'urgent' call to be handled ahead without breaking this mutual exclusion mechanism.
6. The formulation of separate-ness adds complication: the layers of objects, handlers (subsystems) and systems; the necessity for rules to prevent traitors;[1] and the overloading of formal arguments with implicit reservation make it difficult to pass separate parameters without locking.
7. Some areas are omitted:
   (a) (asynchronous) exceptions, which are needed for duels;
   (b) real-time; and
   (c) interrupts (from external processes and devices).
8. Finally, there are a number of implementation matters that need to be addressed for a practical solution:
   (a) failure of the underlying communication system;
   (b) races on reservations;
   (c) termination detection;
   (d) deadlock detection; and
   (e) scalability.

The overall complexity of SCOOP makes it particularly challenging to both implement and model. Some issues described above are addressed in the latest version of SCOOP [Nie07], but the majority are not.

4. Cameo: an alternative model

We now present our alternative concurrency model for Eiffel.[2] We start from (sequential) Eiffel, rather than from SCOOP, and first present the model informally, supplemented by an example. We next express the concurrency aspects of the model in CSP, and then summarise results from mechanical analysis.

[1] A *traitor* is a separate object that becomes attached to a local entity: thus the semantics are confused.
[2] Cameo: a partial anagram of the leading letters of 'explicit asynchrony model of concurrent eiffel objects'.

We want existing (sequential) programs to have the same behaviour as now *without changing the text of the program*. Additionally, we want to preserve as much use as possible of existing libraries even when used in a concurrent context. Finally, our model is clearly inspired by SCOOP, but aims to mitigate some of the issues identified so far.

4.1. Objects and concurrency

We give each object a notional thread of control. Semantically, each object should be able to make independent progress (although implementations would clearly have to multiprogram). This includes expanded objects as well as reference objects. Note that we do not expect every object to have an operating system-level thread. We simply would not be able to commit a thread for every **INTEGER** object, for example. The compiler will need to handle this issue carefully.

Each object has an associated queue of feature calls made by itself or other objects. These calls are processed in FIFO order to implement mutual exclusion to prevent races (as in SCOOP). At most one feature call can be executing on an object at any one time. If there are no calls in the queue, then the object is idle until a call is made. If all objects are idle, then the system terminates.

4.2. Making calls and asynchrony

All calls are synchronous unless the text of the program indicates otherwise using the **async** keyword to qualify the call, e.g., **async** `a.f`. Thus concurrency is introduced explicitly in the program text, ensuring that existing programs do not inadvertently introduce concurrency. We speculate that real programs (rather than illustrations) will generally use more sequential calls than asynchronous calls, so the programmer burden is lower.

An object a making a call $b.f$ (i.e., feature f of object b) places this call f on b's queue. As an exception, if the call is unqualified (i.e., f or **Current**.f) or $a = b$ and the call is synchronous then the call executes immediately. This is to ensure that trivial deadlocks do not occur due to aliasing and matches the expected behaviour in a sequential program.

Asynchronous calls, i.e., those marked **async**, are always enqueued. **async** is allowed to decorate an unqualified call: this does not create an additional thread in the object, but instead enqueues another call on **Current**.

This carries a clear risk of deadlock: an asynchronous unqualified function call cannot return, since the function call will not start until the first call has completed—but this will not complete until the function call does. Aliasing carries this risk in general, and is a good reason to require proofs of either lack of aliasing or at least correct behaviour even in the presence of aliasing.

A compiler can be optimistic and treat synchronous calls as asynchronous if it can guarantee semantically equivalent behaviour.

Function calls result in *two* entries being enqueued: one entry to indicate the feature called (and its parameters, etc.) and a second to indicate the synchronisation point where it requires the result of that function. This could be later in the feature body, using the wait-by-necessity mechanism, or even omitted entirely if the result is not actually needed (although the developer might wish to question the purpose of such a call; excessive optimisation might even remove such a call when it is necessary).

Procedure calls only require one entry to be enqueued. Reference to attributes should be treated as for function calls (due to Eiffel's 'uniform access principle'). Creation procedures can also be decorated with **async**.

4.3. Locking, lazy locks and lock-passing

We require that feature calls can only be enqueued on locked objects, i.e., **Current** holds the 'active lock' on the callee. Each object holds a list of objects that have reserved it; the last entry is the 'active reservation' or 'active lock'. This is part of the lock-passing mechanism from [BPJ07]. So for a calling $b.f$, a must hold the active lock on b.

All objects given as formal arguments are locked unless explicitly excluded using the **unlocked** keyword in the declaration of the argument list, `h (a: unlocked C)`. Locking is the default to reduce potential races when existing sequential libraries are re-used in a concurrent context. Note that **unlocked** is associated with the declaration of

the feature, not a call to a feature, as the need to reserve objects is best viewed as part of the contract offered by the feature.

'Lazy locks' are automatically made on unlocked objects that are the subject of a feature call. This lock only exists for the duration of that call. This applies to the body of features and also to assertion evaluation. Lazy locks can apply to both synchronous and asynchronous calls; access to an asynchronous result requires another lazy lock against the object.

The main purpose of lazy locks is to allow existing sequential code to work unchanged, even though an 'active lock' is required to enqueue feature calls on other objects. Additionally, it allows blocks of code that do not require mutual exclusion on a object to still use it (when no other object is reserving it). This is at the expense of potentially greater overheads.

A lazy lock is semantically equivalent to a (synchronous) call to a wrapper in the current object. This wrapper has the same preconditions as the called feature.[3]

Locks are passed on through call chains, as described in [BPJ07]. Thus an object may be locked if

- it is not currently locked; or
- a parent caller (whether direct or indirect) holds the lock. While a child call holds the lock, the parent temporarily loses the active lock.

4.4. Waiting, preconditions and suspension

Feature calls have wait conditions as for SCOOP.

Wait conditions apply to all calls, both synchronous and asynchronous. Calls that cannot obtain all locks needed to proceed, or which fail their wait conditions are suspended and retry later.

Suspended calls are queued and should be re-evaluated whenever the system detects a change such that the wait conditions may now be true. To ensure liveness, suspended calls should be re-attempted in FIFO order of suspension, although a given call might still be blocked on other locks or wait conditions, or there may be other features active on that object.

If the compiler or run-time system can determine that all wait conditions can never be true, then an error or exception should result. In particular, if at the time the call is enqueued, it can be determined that the wait conditions will never be true, then a synchronous (precondition violation) exception should be raised. However, we need to determine if this is feasible, and when the compiler should guarantee that a wait condition will be thus converted. Otherwise, an expectation that an exception will be raised in some circumstances might not be fulfilled.

Future work (dealing with real-time programming) may impose or suggest further scheduling policies. This could be based on Welling's work on RECOOP [Wel06].

4.5. Asynchronous exceptions

The developer should choose whether (groups of) asynchronous exceptions cause an object to either

- cause the whole system to halt; or
- cause the object to die without attempting to process any other feature calls from its queue. Future attempts to access the object cause the caller to receive the (synchronous) exception separate_object_failure.

This mechanism is described in more detail in [BP07a] (along with discarded alternatives) . We model it in CSP in Sect. 6.

Objects are assigned to processing resources (e.g., dedicated CPUs, POSIX threads) that we call *partitions*. This assignment is via a policy set by the engineer at compile time. There is no intrinsic reason why objects should not be able to migrate (or indeed, be replicated for fault-tolerance) between partitions. However, we do not describe that mechanism here. A problem in the underlying partition communication system that prevents communication with an object results in the exception partition_communication_failure (again, as described in [BP07a]).

[3] This fixes a problem of trivial deadlock in the buffer example identified by Nienaltowski when an early version of this work was presented at CORDIE'06.

Interrupts from outside the Eiffel system (e.g., operating system or device interrupts) are placed in a queue that can be waited on, or interrogated by other objects.

5. Example program: buffer-consumer

5.1. SCOOP version

We first present a standard SCOOP version of the buffer-consumer example, directly derived from http://www.cs.yorku.ca/~jonathan/students/FuksSlides.ppt and [AENV04] (with some renaming and modifications following mechanical analysis— see Sect. 5.3):

```
class ROOT is

    create make

feature

  make
    local
      b     : separate BUFFER
      p     : separate PRODUCER
      c     : separate CONSUMER
    do
      create b
      create p.make(b)
      create c.make(b)
       start_all (p, c)
    end

  start_all (p : separate PRODUCER; c : separate CONSUMER)
    do
      p.loop_forever
      c.loop_forever
    end

end -- ROOT


class PRODUCER

  create make

feature {NONE}

  local_b : separate BUFFER

  put (b : separate BUFFER, m : MESSAGE)
    require
      not b.is_full
    do
      b.put(m)
    end

feature

  make (b : separate BUFFER)
    do
      local_b := b
    end

  loop_forever
    local
      m : MESSAGE
    do
      loop
```

```
            m := produced_data
            put(local_b, m)
        end loop
    end

end -- PRODUCER


class CONSUMER

    create make

feature {NONE}

    local_b : BUFFER

    get (b : separate BUFFER) : MESSAGE
        require
            not b.is_empty
        do
            Result := b.get
        end

feature

    make (b : separate BUFFER)
        do
            local_b := b
        end

    loop_forever
        local
            m : MESSAGE
        do
            loop
                m := get(local_b)
                ... do stuff with m ...
                munged_data
            end loop
        end

end -- CONSUMER
```

We assume that we have a standard buffer:

```
class BUFFER[G] creation
    make

feature

    put (x:G) is
        require
          not is_full
        ensure
          count = old count + 1
        end

    item : G is
        require
          not is_empty
        end

    remove is
        require
          not is_empty
        ensure
          count = old count - 1
        end
```

```
    is_full  : BOOLEAN is ...
  is_empty : BOOLEAN is ...
end -- BUFFER
```

5.2. Cameo version

The code using this model is very similar, using an identical buffer. As in the SCOOP version, this illustrates the typical model of OO code re-use: neither model needs to change the definition of the buffer to make it safe in a concurrent context in this example.

```
class ROOT is

      create make

feature

  make
      local
        b      : BUFFER
        p      : PRODUCER
        c      : CONSUMER
      do
        create b
        async create p.make(b)
        async create c.make(b)
      end

end -- ROOT


class PRODUCER

  create make

feature {NONE}

  local_b : BUFFER

feature

  make (b : unlocked BUFFER)
      do
        local_b := b
        loop_forever
      end

  loop_forever
      local
        m : MESSAGE
      do
        loop
          m := produced_data
          local_b.put(m)
        end loop
      end

end -- PRODUCER


class CONSUMER

  create make
```

```
feature {NONE}

    local_b : BUFFER

    get (b : BUFFER) : MESSAGE
        require
            not b.is_empty
        do
            Result := b.get
        end

feature

    make (b : unlocked BUFFER)
        do
            local_b := b
            loop_forever
        end

    loop_forever
        local
            m : MESSAGE
        do
            loop
                m := get( local_b )
                ... do stuff with m ...
                munged_data
            end loop
        end

end -- CONSUMER
```

5.3. Differences

In such a small program, the differences arising between the original SCOOP model, and the alternative model, are few; in this case they are:

- **separate** is removed, since all the objects are notionally concurrent.
- The argument (the buffer) to each of the creation procedures for the producer and consumer is annotated **unlocked**. The SCOOP version serialises these two creations due to the implicit reservation on the buffer.
- In ROOT.make, the creation of the buffer is unadorned, but the creation of the producer and consumer objects are annotated with **async** to explicitly indicate that the calls to their creation procedures should be asynchronous calls. The creation of the producer and consumer objects also starts the two loop_forever procedures (but not in the SCOOP version).
- start_all is not required in the alternative version, but is required for SCOOP: this starts the two loop_forever procedures without holding onto the buffer lock. If we were to attempt to use the Cameo versions of PRODUCER.make and CONSUMER.make in the SCOOP version, the subsequent calls to loop_forever would prevent the buffer lock being released during ROOT.make.
- PRODUCER.put is not required in the alternative version, since it merely exists to obtain a lock on the buffer and call put on the buffer. Thus a lazy lock suffices here (and implicitly calls a wrapper similar to PRODUCER.put).
- A lazy lock is not sufficient to replace CONSUMER.get: the two lines

```
Result := b.item
b.remove
```

  must not be separated or a race could result if other consumers called CONSUMER.get at the same time.

This example is usually generalised to the case of producing and consuming several items at once. In this case, more explicit mutual exclusion is required and an explicit, multi-item PRODUCER.put would be required.

6. CSP sketch of the alternative mechanism

Using the process algebra CSP [Hoa85], we sketch a model of our alternative formulation of concurrency for Eiffel. This is an incremental variation of the SCOOP model [BPJ07]. Briefly,

- *Stop* denotes the process that does nothing;
- $a \rightarrow P$ performs event a then behaves as process P;
- $P \,\Box\, Q$ allows the environment to choose between processes P and Q via their first events. If the first events of P and Q are identical, then the choice of P and Q is nondeterministic.
- *Skip* is the process representing successful completion.
- $P;\ Q$ behaves as P. After P is successful, the process behaves as Q.
- $P \,|||\, Q$ interleaves the two processes P and Q.
- $P \|_A Q$ denotes the two processes P and Q agreeing on events in the set A and interleaving other events.
- $g \Rightarrow P$ is the process P if g is true, and is *Stop* otherwise. (Some authors use $g \& P$ for this.)

We refer readers to Hoare's text (or a later text by Roscoe [Ros98] or Schneider [Sch00]) for more information about CSP.

We start from our CSP model of SCOOP [BPJ07]. It is important to note that we are capturing *only* the concurrent aspects of our alternative model; in particular, we are not interested in internal state changes within objects, nor are we interested in capturing inheritance at this stage.

6.1. Definitions and CSP events

We use several definitions similarly to the CSP model of SCOOP. A *system* comprises a number of objects, each of which has its own notional thread of control. The overall system 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 objects. Here, we are borrowing some elements and terms of the Ada 95 model of distributed processing [TD97].

The CSP events involved are listed in Appendix 9. Some are different from [BPJ07] in this alternative model, including

- *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.

Additional events have been added for exceptions, and are described in Sect. 6.7.

6.2. Processes

As in [BPJ07], a system is modelled by the alphabetised parallel composition of its components. We adopt the approach of allowing all behaviours (i.e., all possible traces) then each individual component restricts the permitted behaviours in different ways (e.g., $RESERVATIONS$ restricts the events permitted to enforce the rules on locking).

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

If a particular analysis is not concerned with behaviour due to exceptions, then the last two processes (*EXCEPTIONS* and *DEADOBJECTS*) are omitted.

A number of (book-keeping) processes are unchanged from the SCOOP model [BPJ07].

- *CALLCOUNT* allows a call c to obtain the next call number c' from a simple counter.
- *CALLPARAMS* records the parameters to a particular call.
- *CALLSEPPARAMS* records the objects that need to be reserved before a particular call can proceed. In this model, all objects given as parameters are to be locked unless annotated with **unlocked**.
- *OBJECTLOCALS* represents the local variables for a particular object.
- *BIGLOCK* provides a simple model-wide mutex to prevent races when obtaining reservations.

These clearly perform the same function in both models, and would could arise in most models of object-oriented concurrency.

6.3. New and modified processes

We now describe some of the changes from the model in [BPJ07]. *EXCEPTIONS* and *DEADOBJECTS* are described in Sect. 6.7.

6.3.1. SEQOBJECTS

SEQOBJECTS is a new process (for both the SCOOP and alternative models) 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).

6.3.2. RESERVATIONS

RESERVATIONS is also slightly modified: the last clause of (12) in [BPJ07] is (for this model) written

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

following the removal of subsystems. *addCallRemote* now represents an object i' requesting the enqueuing of feature f as call c on object i. So the full definition becomes

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

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.
- The next clause allows the event $\mathit{reserved}.c.p$ which indicates that the objects in the set p have been reserved for call c.
- Next, the contrary event, $\mathit{notReserved}.c.p$, is offered if $\mathit{reserved}.c.p$ 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 remote 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 model whereby a call c' running on object i' can enqueue call c on the remote object i only if c' holds the active reservation on i.

6.3.3. *ALLOBJECTS*

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

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

Essentially, each object is created before it can do anything, except for the root object, which is implicitly created during bootstrap. Thereafter, the various reservation events are enabled.

6.3.4. *ALLSCHEDULERS*

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. 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.

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

The scheduler for the root object has the root call already in its queue; the schedulers for all other objects have empty queues and the object must be created before anything else can happen to it. When a queue is empty, the scheduler offers the *terminate* event: only when all schedulers offer *terminate* can the whole system finally stop.

$$\begin{aligned} SCHEDULER(Object.rootClass.1) &\triangleq SCHEDULER2(\langle (Call.0, Object.rootClass.1, rootFeature)\rangle, \langle\rangle) \\ SCHEDULER(i) &\triangleq createObject.i?c \rightarrow SCHEDULER2(\langle\rangle, \langle\rangle) \\ \text{where } i \neq Object.rootClass.1 & \quad \Box terminate \rightarrow Stop \end{aligned}$$

Finally, a live object has a number of events open to it:

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

Taking this clause-by-clause:

- remote calls can be enqueued (at the end of the FIFO queue, Q);
- if the list of current calls, C, is empty, then a remote call from Q;
- if no calls are currently running (i.e., C is empty) and there are no remote calls waiting (Q is empty), then *terminate* is offered;
- a local call can be enqueued, which causes that call to be immediately scheduled;
- a running call can be unscheduled (i.e., when it has finished).

6.3.5. *ALLDOCALLS*

ALLDOCALLS has a moderate number of changes (again, to remove references to subsystems and in some cases, add further references to objects). This process simulates the behaviour of any given call: it deals with

- a call attempting to obtain the necessary locks to proceed;
- checking preconditions;
- performing the work of each call, which is dependent on the exact program being simulated: this itself can involve creating more objects and enqueuing further calls;
- then releasing the obtained locks; and also
- dealing with the release of locks and re-scheduling if the call fails to obtain the necessary locks, or the preconditions fail.

6.4. 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 call at 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:

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

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,

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

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

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

to

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

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.

6.5. Adding calls

The alternative model has a slightly different definition of $ADDCALL(c, c', i, j, f, a)$, which represents call c on object i enqueuing a call c' of feature f on object j. Also note the additional parameter

$$a \in ADDCALL\_MODES = \{sync, async\}$$

which is used to indicate the use of the **async** keyword (or lack thereof). Whereas all remote calls in SCOOP are asynchronous (when assuming wait-by-necessity for functions), remote calls are only asynchronous in Cameo when explicitly qualified with **async**; otherwise, the call is synchronous.

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

Thus *ADDCALL* says that

- an object can add a local call to itself (note that this model deals with the objects, not entities, so the problems of aliasing do not arise directly);
- an object can add remote calls in either asynchronous or synchronous modes;
- both local and remote synchronous calls have to wait for the call to complete before they progress.

6.6. Lazy locks

The alternative model presented earlier allows lazy locks (see Sect. 4.3). This is a syntactic mechanism rather than semantic. This means that the implicit wrapper needs to be explicitly included in the CSP simulation of a particular program. Thus there is no other special modification to the model to accommodate lazy locks.

For example, in the Cameo version of the buffer-consumer example, the call local_b.put(m) in PRODUCER.loop_forever has to be implicitly rewritten as a call put(local_b, m) (in PRODUCER) where put is defined as

```
put (b : BUFFER, m : MESSAGE)
  require
    not b. is_full
do
    b.put(m)
end
```

which, by our rules, requires a lock on the buffer. The precondition is lifted from BUFFER.put.

6.7. Exceptions

Synchronous exceptions can arise in Eiffel as well as concurrent variants of Eiffel. When an exception arises, the routine can handle it by calling **retry** in a **rescue** clause. Otherwise, if there is no **rescue** clause or **retry** is not used, then the routine fails and propagated the exception. If the root procedure fails, then the whole system stops with an exception.

Thus our model need to simulate exceptions being raised, being handled via **retry**, being propagated to the caller and causing the root procedure to stop.

An asynchronous exception is caused when an asynchronous routine fails: because the calls is asynchronous, the caller has (possibly) already terminated so there is no obvious caller to report the failure to. (Note that *synchronous* remote calls do not suffer this problem, as the caller still exists because the call is synchronous.)

Each object can be configured to respond to asynchronous exceptions by either

- halting the entire system;
- causing that object to 'fail'; or
- ignoring the exception [BP07a].

We add two processes to the model for exceptions (both for SCOOP and Cameo).

The first, *EXCEPTIONS* monitors individual calls being scheduled. Local, synchronous calls made by that call are themselves watched, then after unscheduling, the final state (successful or failed) can be obtained. Remote, asynchronous calls behave similarly until unscheduled: if the call failed, then *DEADOBJECTS* is told that the object should die. The two processes given here are for the SCOOP version: the Cameo version is very similar.

Firstly, every call is associated with an *EXCEPTIONS2* process:

$$EXCEPTIONS \triangleq |||\, c : CALLS \bullet EXCEPTIONS2(c)$$

If it relates to the bootstrap call, then that call is already implicitly added, so the first event of the overall system is to schedule the call. This is then effectively a remote call:

$$EXCEPTIONS2(Call.0) \triangleq schedule?j.c?i?f.remote \rightarrow EXCEPTIONSR1(c, i)$$

All other calls depend on local vs. remote, synchronous vs. asynchronous:

$$\begin{aligned} EXCEPTIONS2(c) \triangleq\ & addCallLocal?j.c?i?f \rightarrow schedule?j.c?i?f.local \rightarrow \\ \text{where } c \neq Call.0 \quad & EXCEPTIONSL1(c) \\ & \Box addCallRemote?j'?j.c?i?f.async \rightarrow schedule?j.c?i?f.remote \rightarrow \\ & EXCEPTIONSR1(c, i) \\ & \Box addCallRemote?j'?j.c?i?f.sync \rightarrow schedule?j.c?i?f.remote \rightarrow \\ & EXCEPTIONSL1(c, i) \end{aligned}$$

Remote asynchronous calls effectively break the call chain for the purpose of propagating exceptions; local calls and remote synchronous calls do not.

A local or remote synchronous call records exceptions arising (*fail*) and being successfully handled (*retry*). It is also possible to extract the current state of the call via *okaySoFar* and *failSoFar*. *EXCEPTIONSL1* is the current process when the call is currently successful; *EXCEPTIONSL2* when it has failed (so far).

$$\begin{aligned} EXCEPTIONSL1(c) \triangleq\ & fail.c \rightarrow EXCEPTIONSL2(c) \\ & \Box retry.c \rightarrow EXCEPTIONSL1(c) \\ & \Box okaySoFar.c \rightarrow EXCEPTIONSL1(c) \\ & \Box unschedule?j.c \rightarrow notDie.c \rightarrow EXCEPTIONSL3(c) \\ EXCEPTIONSL2(c) \triangleq\ & fail.c \rightarrow EXCEPTIONSL2(c) \\ & \Box retry.c \rightarrow EXCEPTIONSL1(c) \\ & \Box failSoFar.c \rightarrow EXCEPTIONSL2(c) \\ & \Box unschedule?j.c \rightarrow notDie.c \rightarrow EXCEPTIONSL4(c) \end{aligned}$$

EXCEPTIONSL1 and *EXCEPTIONSL2* can both engage in the event *unschedule*. Because the call chain is unbroken for propagating exceptions, both engage in *notDie* (which is also agreed by *DEADOBJECTS*, below). Thereafter, as *EXCEPTIONSL3* and *EXCEPTIONSL4* respectively, they report the success or failure of the call that has now ended.

$$\begin{aligned} EXCEPTIONSL3(c) &\triangleq successful.c?c' : (CALLS \backslash \{c\}) \rightarrow EXCEPTIONSL3(c) \\ EXCEPTIONSL4(c) &\triangleq failed.c?c' : (CALLS \backslash \{c\}) \rightarrow EXCEPTIONSL4(c) \end{aligned}$$

EXCEPTIONSR1 and *EXCEPTIONSR2* and are the remote asynchronous versions of *EXCEPTIONSL1* and *EXCEPTIONSL2*. In this case, a failed call engages in *die* to force the object concerned to fail; and it is not possible to extract the success or failure state of the object.

$$\begin{aligned} EXCEPTIONSR1(c, i) \triangleq\ & fail.c \rightarrow EXCEPTIONSR2(c, i) \\ & \Box retry.c \rightarrow EXCEPTIONSR1(c, i) \\ & \Box okaySoFar.c \rightarrow EXCEPTIONSR1(c) \\ & \Box unschedule?j.c \rightarrow notDie.c \rightarrow Stop \\ EXCEPTIONSR2(c, i) \triangleq\ & fail.c \rightarrow EXCEPTIONSR2(c, i) \\ & \Box retry.c \rightarrow EXCEPTIONSR1(c, i) \\ & \Box failSoFar.c \rightarrow EXCEPTIONSR2(c) \\ & \Box unschedule?j.c \rightarrow die.i.c \rightarrow Stop \end{aligned}$$

Thus a call may engage in the events *fail* to simulate an exception being raised, and *retry* to simulate the exception being handled. When the call is unscheduled, local and synchronous remote calls can report their success or failure (as appropriate). Failed asynchronous remote calls cause the object concerned to die (simulating the asynchronous exception).

DEADOBJECTS comprises a process for each object:

$$DEADOBJECTS \triangleq ||| i : OBJECTS \bullet DEADOBJECT1(i)$$

A 'live' (not failed) object allows calls to be added, calls to be scheduled, and can also die:

$$\begin{aligned} DEADOBJECT1(i) \triangleq\ & addCallLocal?j?c.i?f \rightarrow DEADOBJECT1(i) \\ & \Box addCallRemote?j'?j?c.i?f \rightarrow DEADOBJECT1(i) \\ & \Box schedule?j?c.i?f?m \rightarrow DEADOBJECT1(i) \\ & \Box die.i?c \rightarrow DEADOBJECT2 \end{aligned}$$

However, once the object has died, it no longer allows calls to be added (by not offering the relevant event), nor can any further calls that have already been added be scheduled:

$$\begin{aligned} DEADOBJECT2(i) \triangleq\ & die.i?c \rightarrow DEADOBJECT2(i) \\ & \Box deadObject.i?c \rightarrow DEADOBJECT2(i) \\ & \Box failedSchedule?j?c.i?f?m \rightarrow DEADOBJECT2(i) \end{aligned}$$

If the exception policy is to ignore an exception, then none of this is triggered (i.e., the *fail* event should be omitted in simulated programs). If the whole system should halt, then the first *die* event should stop the whole system (this is currently not modelled).

These processes can cause *schedule* events to be refused in $ALLSCHEDULERS$ (see Sect. 6.3.4). Thus the clauses which offer *schedule* have to be modified to allow an external choice of *schedule* and *failedSchedule*.

Finally, to simulate the cascade of synchronous exceptions (both local and remote), the SCOOP version of $ADDCALL$ is modified to read

$$\begin{aligned} ADDCALL(c, c', i, j, f) \triangleq\ & (okaySoFar.c \rightarrow \\ & getHandler.i?h_i : SUBSYSTEMS \rightarrow \\ & \quad getHandler.j?h_j : SUBSYSTEMS \rightarrow \\ & \quad \Big(h_i = h_j \Rightarrow (addCallLocal.h_i.c'.j.f \rightarrow \\ & \qquad (successful.c'.c \rightarrow Skip \Box failed.c'.c \rightarrow fail.c \rightarrow Skip)\Big) \\ & \quad \Box h_i \neq h_j \Rightarrow addCallRemote.h_i.h_j.c'.j.f.async \rightarrow Skip \\ & \quad \Box deadObject.i.c' \rightarrow fail.c \rightarrow Skip\Big) \\ & \Box failSoFar.c \rightarrow notAdding.c' \rightarrow Skip) \end{aligned}$$

and the Cameo version becomes

$$\begin{aligned} ADDCALL(c, c', i, j, f, a) \triangleq\ & (okaySoFar.c \rightarrow \\ & \quad \Big(i = j \Rightarrow (addCallLocal.c'.j.f \rightarrow \\ & \qquad (successful.c'.c \rightarrow Skip \Box failed.c'.c \rightarrow fail.c \rightarrow Skip)\Big) \\ & \quad \Box i \neq j \wedge a = async \Rightarrow addCallRemote.i.c'.j.f.a \rightarrow Skip \\ & \quad \Box i \neq j \wedge a = sync \Rightarrow (addCallRemote.i.c'.j.f.a \\ & \qquad (successful.c'.c \rightarrow Skip \Box failed.c'.c \rightarrow fail.c \rightarrow Skip)\Big) \\ & \quad \Box deadObject.i.c \rightarrow fail.c \rightarrow Skip\Big) \\ & \Box failSoFar.c \rightarrow notAdding.c' \rightarrow Skip) \end{aligned}$$

The initial check on *okaySoFar*/*failSoFar* stops additional attempts to enqueue later calls if an earlier call failed: this simulates the routine not continuing normally if an exception has already been raised. Additionally, contacting a dead object (*deadObject*) causes this call to fail. Finally, references to *doneCall* are replaced by a choice of *successful* and *failed* to propagate exceptions by engaged in $fail.c$ after $failed.c'.c$.

7. Preliminary mechanical implementation

As the CSP model is a modification of that presented in [BPJ07], it is relatively easy to produce an initial mechanical implementation using CSPsim,[4] again by modification. Some example output is available online:[5] example5 is the buffer-consumer example (from Section 5) and example6 is the older example1 [BPJ07] modified to examine the behaviour of exceptions.

7.1. Initial results

It is easy to produce a number of random traces using CSPsim. Initial attempts to use the (inefficient) exhaustive state space checker demonstrated that the state space appears to be larger for Cameo than it does for the SCOOP model. Examining some example traces to investigate this, we concluded that there are greater opportunities for parallelism in this model, but also more locking. Thus locking overhead has to be kept low. This is because we now lock for every non-**Current** call, whereas SCOOP only locked for calls to a different subsystem.

Essentially, we are hoping to gain greater progress due to parallelism by removing subsystems, but this is encumbered by more coordination requirements. We argue that locking resources on a local partition should be cheaper than locking on a non-local partition: the net performance result is difficult to assess at this stage. Our future work will need to include larger example systems to see which factors matter most.

Overall, the alternative model is a little more intractable to model-checking compared to SCOOP because there are many more opportunities for locks to be requested, and for these locks to fail. These failures cause the call to be suspended and then rescheduled. Because there are more concurrent elements (i.e., each object rather than each subsystem) there is much more opportunity for interleaving of these events.

The issue above motivated us to add the CSP process, *RESCHEDULER* (see Sect. 6.4) during this work. A real system would include a restriction on re-scheduling: calls should not be re-scheduled until the object they had failed to lock is freed. This process is added to both the SCOOP and Cameo models.

We briefly considered a more proactive scheduler that would prevent calls starting if they would not be able to obtain their desired reservations. But this would require more global oversight. The re-scheduler solution appears to be more consistent with the FIFO approach to suspensions suggested by Meyer's book [Mey97].

7.2. Buffer-consumer

The buffer-consumer example was implemented using a CSP test harness to model the buffer (since we do not manage state in the Eiffel model) and to simulate data being produced. The harness can decide if the preconditions for particular calls are true or false.

Each call has one instance of the following process:

$$
\begin{aligned}
HARNESS2(c : CALLS;\ i : Integer) \quad = \quad & \Box i < Buffer\_Size \Rightarrow buffer\_add \rightarrow HARNESS2(c, i+1) \\
& \Box i > 0 \Rightarrow buffer\_remove \rightarrow HARNESS2(c, i-1) \\
& \Box preconditionsOkay.c \rightarrow HARNESS2(c, i) \\
& \Box tests\_not\_is\_full.c \rightarrow HARNESS3(c, i) \\
& \Box tests\_not\_is\_empty.c \rightarrow HARNESS4(c, i) \\
HARNESS3(c : CALLS;\ i : Integer) \quad = \quad & \Box i < Buffer\_Size \Rightarrow buffer\_add \rightarrow HARNESS3(c, i+1) \\
& \Box i > 0 \Rightarrow buffer\_remove \rightarrow HARNESS3(c, i-1) \\
& \Box i < Buffer\_Size \Rightarrow preconditionsOkay.c \rightarrow HARNESS3(c, i) \\
& \Box i = Buffer\_Size \Rightarrow preconditionsFail.c \rightarrow HARNESS3(c, i)
\end{aligned}
$$

[4] https://www.scm.tees.ac.uk/p.j.brooke/cspsim/
[5] https://www.scm.tees.ac.uk/p.j.brooke/ce2/

$$\begin{aligned} HARNESS4(c : CALLS;\ i : Integer) \quad = \quad & \Box i < Buffer\_Size \Rightarrow buffer\_add \rightarrow HARNESS4(c, i+1) \\ & \Box i > 0 \Rightarrow buffer\_remove \rightarrow HARNESS4(c, i-1) \\ & \Box i > 0 \Rightarrow preconditionsOkay.c \rightarrow HARNESS4(c, i) \\ & \Box i = 0 \Rightarrow preconditionsFail.c \rightarrow HARNESS4(c, i) \end{aligned}$$

NETWORK_MESSAGES generates a set number of messages:

$$\begin{aligned} NETWORK\_MESSAGES(i : Integer) \quad = \quad & \Box i > 0 \Rightarrow produced\_data \rightarrow NETWORK\_MESSAGES(i-1) \\ & \Box i = 0 \Rightarrow Stop \end{aligned}$$

Then both are combined thus:

$$\begin{aligned} HARNESS \quad = \quad & \|_{A_c}\, c : CALLS \bullet HARNESS2(c, 0) \\ & |||_B\ NETWORK\_MESSAGES(Connections) \end{aligned}$$

where

$$\begin{aligned} A_c \quad = \quad & \{buffer\_add, buffer\_remove, preconditionsOkay.c, \\ & preconditionsFail.c, tests\_not\_is\_full.c, tests\_not\_is\_empty.c\} \end{aligned}$$

and $B = \{produced\_data\}$.

Figures 2 and 3 graphically illustrate two particular traces, one for each of SCOOP and Cameo. Both are set to disallow exceptions, to allow only direct lock-passing, and not to wait for enqueued calls to terminate. Although we were unable to exhaustively explore the example systems, the example traces we randomly produced were all similar to these figures.

From examination of these traces, we note little difference: the behaviours are broadly similar. The start_all procedure is not included in the Cameo version; the SCOOP versions' make procedures for PRODUCER and CONSUMER both terminate quickly before loop_forever is triggered by ROOT. In hindsight, this is unsurprising: each object is mapped to a single subsystem in the SCOOP example, so it has similar concurrency characteristics to Cameo. However, where multiple objects share a subsystem, then they must be serialised. The question becomes "do interesting (real) systems include circumstances when multiple objects on the same subsystem have work that they could perform concurrently?" This has to be compared against the cost of type-systems, traitor prevention, etc.

We note that an earlier version of the SCOOP example in Sect. 5 (which matches other authors' versions) did not include the call to start_all. Instead, this has been previously written similarly to the Cameo version, whereby (for example) PRODUCER.make includes the call to PRODUCER.loop_forever. However, initial example runs demonstrated that this could not work because of the blocking on the buffer.

So while we are unable to exhaustively analyse these examples, we were able to find and diagnose bugs. It is clear that the subtlety of locking remains challenging. Perhaps explicit locks are required to aid programmers: if we were designing Cameo without concern for old code, we would make unlocked the default and require explicit locking (compare with our rules in Sect. 4.3).

7.3. Exceptions

The CSP model of exceptions (see Sect. 6.7) was implemented in CSPsim. The resulting behaviour is predictable and unremarkable in the small examples we tried. Figures 4 and 5 show two example traces. The value of this model is for future work: we will be able to incorporate exceptions into more complicated examples where the presence of exceptions could have significant implications.

8. Discussion

8.1. Remarks on our model

We allow compilers to implement synchronous calls asynchronously if they can guarantee semantically equivalent behaviour. This is so that multiple processing nodes can be utilised in parallel. However, there is the question of defining 'semantically equivalent behaviour'.

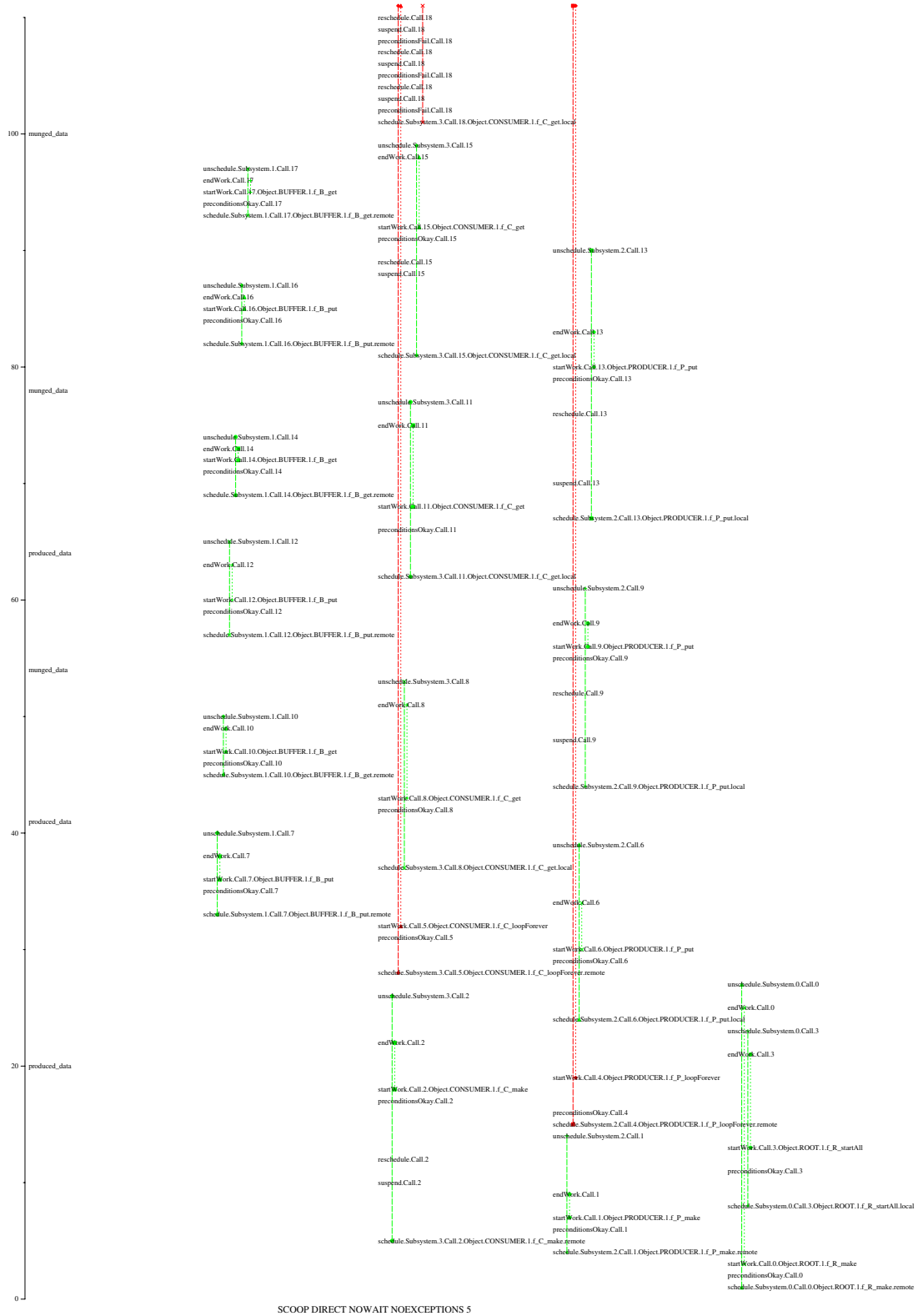

Fig. 2. SCOOP buffer-consumer example

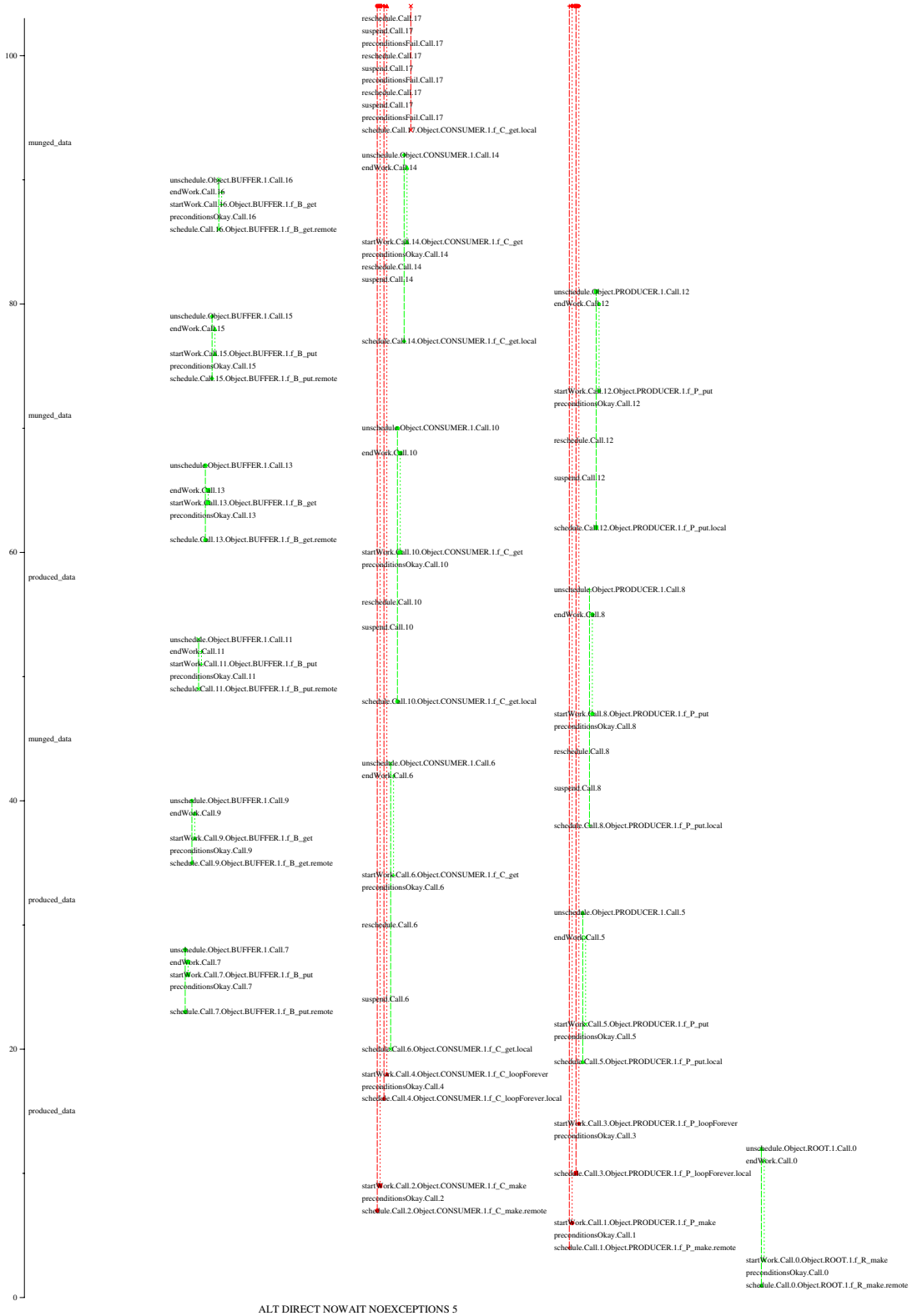

Fig. 3. Cameo buffer-consumer example

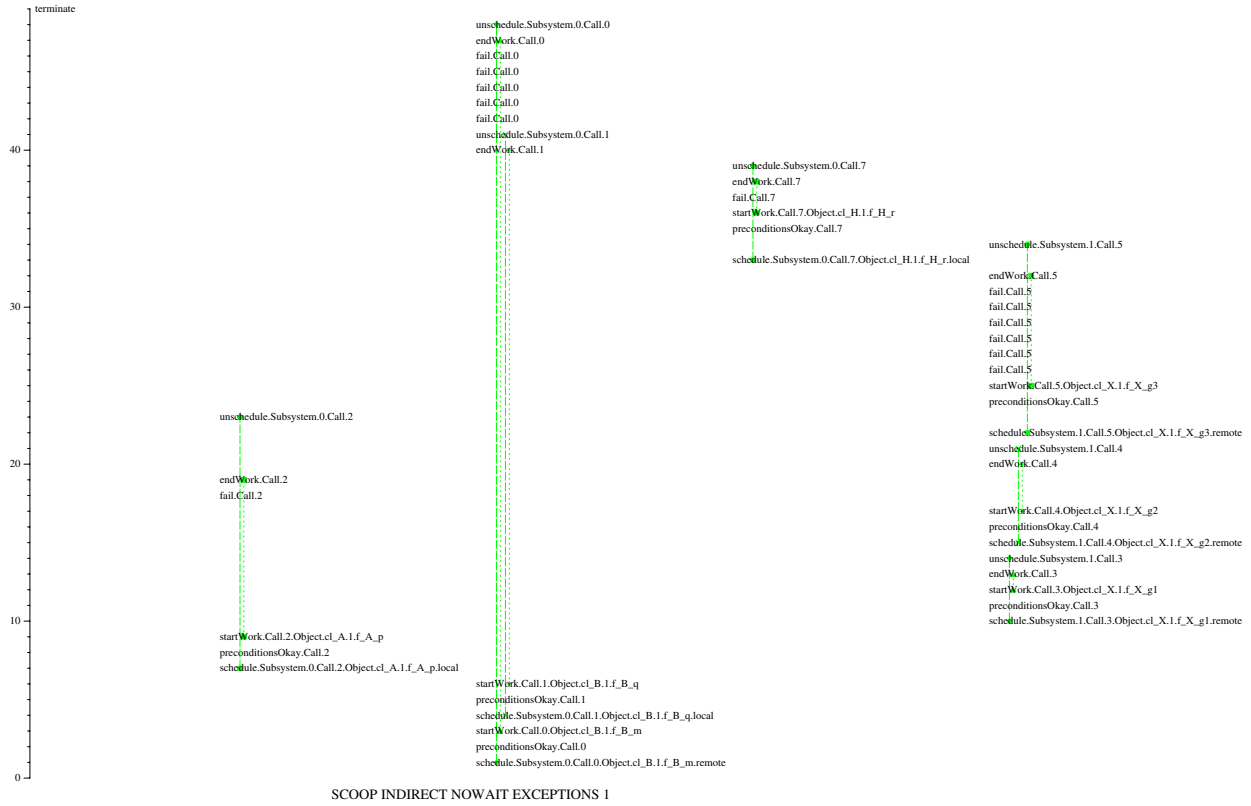

Fig. 4. SCOOP exceptions example

Lazy locks may admit deadlocks when old sequential code is used as a component in concurrent programs, but this is better than allowing races. Without lazy locks, our model would result in most (if not all) existing sequential programs having missing locks in many places. Even then, some sequential code may not work in a concurrent environment due to races by interleaving calls from multiple concurrent clients, or deadlocks by other clients locking a callee. We need to evaluate this further, perhaps by implementation in a transparent compiler (the subject of a proposal itself).

We initially considered the use of both preconditions *and* wait conditions. However, we have been unable to justify separate preconditions. Instead, this model allows an immediate synchronous exception when it is clear that a particular wait condition will never be satisfied.

So far, we have not addressed real-time programming, although a utility class could offer ‘delay’ and ‘delay until’ (as for Ada [TD97]).

The CSP sketch is complete (given the scope we have set) and has been implemented in CSPsim. Further large examples are required (for example, a model of a web server). It is derived from the model proposed for SCOOP [BPJ07], is a little simpler and admits a larger state space. Whether this model translates into improved performance and clarity of understanding remains undecided, and one that we will address in the continuation of this work.

Finally, we note that the complexity of the model has reached the point where a combined state/process-algebra model is more appropriate. The experience of this and prior work should enable a more elegant, useful model to be constructed.

8.2. Comparison with SCOOP

We compare our model against SCOOP and the problems in Sect. 3.

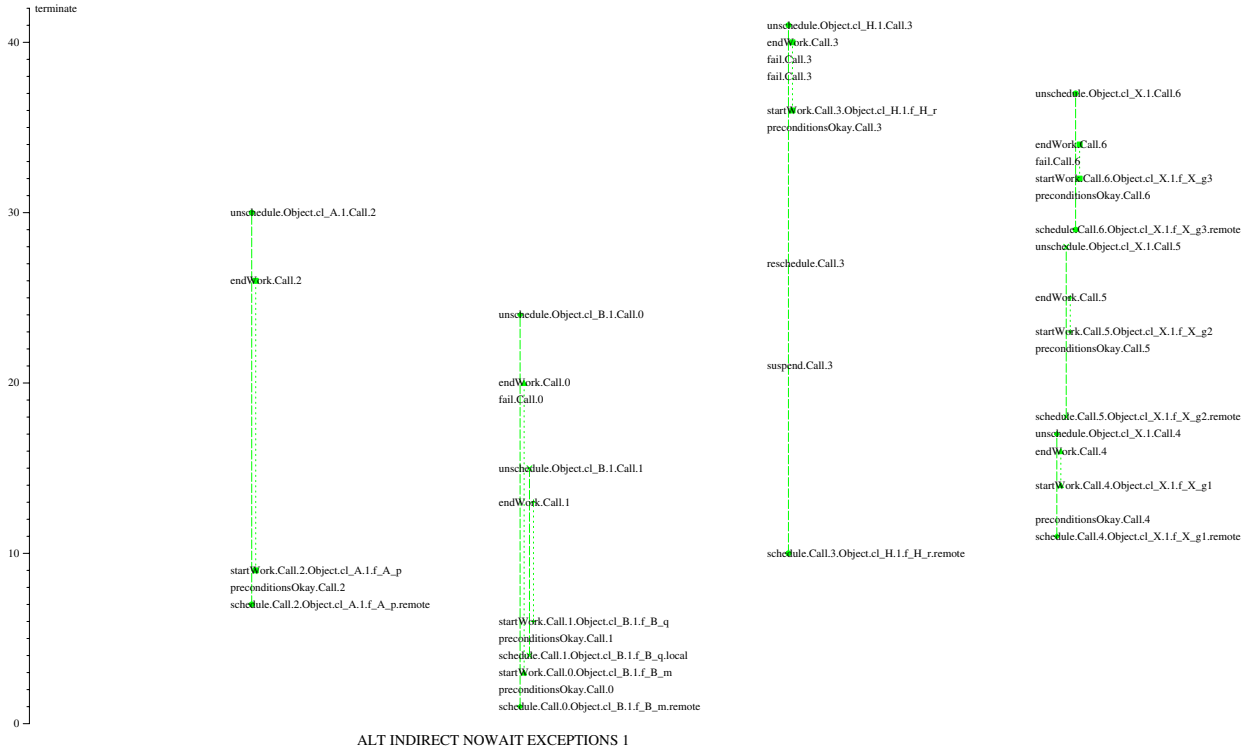

Fig. 5. Cameo exceptions example

1. *Chains of calls and reservations.* Although we identify objects to be locked differently, the handing-on of reservations through call chains is the same mechanism that we propose for SCOOP itself in [BPJ07].
2. *Release of reservations.* Lazy locks are released as soon as the call is enqueued; other locks are only released at the end of the feature concerned. (This matter requires further investigation for both SCOOP and this alternative formulation.)
3. *Reduced parallelism by subsystems and implicit reservations.* This model removes subsystems, but still requires reservations to enqueue feature calls. Whether this produces practical results is unclear.
4. *Rescheduling of blocked calls.* We have explicitly given some details of the handling of suspended calls, although we recognise that more details are needed. In particular, real-time behaviours would likely give a different set of requirements.
5. *Priorities and call queues.* Calls are enqueued in both models, although SCOOP enqueues them against the subsystem (potentially many objects) whereas our model has per-object queues. The objects themselves can be given priorities for processing by scarce resources, although there are a number of real-time programming issues still to address.
6. *Complications of separate-ness (subsystems, traitors, reservation via arguments).* This model does not contain subsystems. A mechanism is included to allow objects to be given in argument lists without implicitly reserving them.
   We claim that this makes this model simpler: we have discarded entirely the whole concept of handlers and subsystems. Because all objects are ‘separate’, we no longer require the concept of traitors, or separateness consistency rules. Thus, the type system no longer needs fixing for library calls, e.g., having separateness being a dimension of type.
   Making all objects concurrent gives a more symmetric semantics (we no longer find some objects local with respect to others). However, our model does not break existing sequential programs because asynchronous calls are made explicitly rather than implicitly (via a call on a separate object) as in SCOOP.

7. *Omission of asynchronous exceptions, real-time and interrupts.* Mechanisms are given for handling asynchronous exceptions and interrupts. We propose both mechanisms for inclusion in SCOOP itself. We have not attempted to address real-time.
8. *Implementation matters.* Most of these issues can be solved equally for SCOOP and our model, but we have not attempted to do so here. Both require a CCF (for mapping objects to real resources) and deadlock detection. The notion of partitions is given in this model, and an explicit exception identified for failure of the underlying implementation (both have been previously suggested for SCOOP by us). Object migration and replication is easily admitted by this model.
   Notably, scalability remains an open question for this model.

As well as the points above, we should also consider the pragmatics of programming using the notation. This needs to be tested with large programs.

8.3. Tool support

CSPsim remains useful in this work, although larger examples require increasing time for explorations. Implementation of the proposed 'symbolic descriptions' of event sets (e.g., internally passing a description of the events in a set rather than enumerating each one) [BP07b] is the most likely means to improve performance. This would also bring more systems within reasonable range of exhaustive checking.

Another possible approach include a mechanism for making sequences of events behave atomically, i.e., preventing these sequences being split by other events. This would be used to reduce the state space by reducing the amount of interleaving, but we will need to be careful of our choice of events to retain validity of the output.

Finally, integrating a code generator within Eiffel Studio that outputs example CSPsim code or input for FDR2 would reduce the amount of work and potential error in constructing these examples.

9. Conclusion

We have briefly described SCOOP, a major existing proposal for adding concurrency to Eiffel. After identifying some problems with SCOOP, we have proposed an alternative model, Cameo, and sketched a model in CSP. A preliminary mechanical implementation shows a larger state space that is (at this time) intractable to search exhaustively with our current tools. It is possible to obtain example traces, and we used them to identify an error in the SCOOP version of the buffer-consumer example. These example traces then show no significant difference between the two semantic models.

The differences are described in the previous section. The most significant concern the removal of subsystems so that all objects are independently concurrent (at least conceptually), and the introduction of an explicit mechanism for introducing asynchrony. The former has the major benefit of simplifying the type system and removing the rules on traitors.

The next steps in our work are

1. develop some larger examples to further explore the semantic variations of SCOOP versus Cameo, the direct versus indirect lock-passing and to explore the impact of exceptions.
2. further develop the CSP tools (perhaps adding a means of reducing interleaving) so that we can further analyse the models for undesirable behaviours (ideally, exhaustively);
3. implement this alternative model either as a preprocessor to an existing Eiffel compiler, or in a specially-produced compiler; and
4. assess requirements for, and implement any needed support for real-time programming, and safety- and security-critical systems.

Acknowledgments

We thank the anonymous referees, Jeremy Jacob and the CORDIE attendees for their helpful comments and discussions.

Appendix A: Glossary of events

We list the events in Cameo and identify the processes that engage in those events. Note that some events vary slightly between the SCOOP and Cameo versions (see Sect.6).

A.1 Visible events

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

Event(s)	Types	Components involved	Meaning
addCallLocal.$c.i.f$	c : *CALLS* i : *OBJECTS* f : *FEATURES*	*ALLDOCALLS* *ALLSCHEDULERS* *EXCEPTIONS* *DEADOBJECTS*	A call to feature f on object i is enqueued as call c
addCallRemote.$i'.c.i.f.a$	i' : *OBJECTS* c : *CALLS* i : *OBJECTS* f : *FEATURES* a : *ADDCALL_MODES*	*ALLDOCALLS* *ALLSCHEDULERS* *RESERVATIONS* *EXCEPTIONS* *DEADOBJECTS*	A call to feature f on object i is enqueued as call c by a call running on object i'
biglock.c *bigfree*.c	c : *CALLS*	*BIGLOCK* *ALLDOCALLS*	System-wide reservations for call$c$
createObject.$i.c$	i : *OBJECTS* c : *CALLS*	*ALLOBJECTS* *ALLDOCALLS* *ALLSCHEDULERS*	Call c causes creation of 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.$c.i.f.m$	c : *CALLS* i : *OBJECTS* f : *FEATURES* m : {*local*, *remote*}	*ALLDOCALLS* *ALLSCHEDULERS* *EXCEPTIONS* *DEADOBJECTS*	Call c (feature f on object i) is to start executing
failedSchedule.$c.i.f.m$	c : *CALLS* i : *OBJECTS* f : *FEATURES* m : *SCHEDULE_MODES*	*ALLSCHEDULERS* *DEADOBJECTS* *ALLCALLSDONE*	Call c (feature f on object i) is unable to start executing (failed object)
startWork.$c.i.f$	c : *CALLS* i : *OBJECTS* f : *FEATURES*	*ALLDOCALLS*	Start call c, where the call is feature f operating on object i
endWork.c	c : *CALLS*	*ALLDOCALLS*	Normal end of call c
retry.c *fail*.c	c : *CALLS*	*EXCEPTIONS* *ALLDOCALLS*	Call c has retries or fails
die.$i.c$	c : *CALLS* i : *OBJECTS*	*EXCEPTIONS* *DEADOBJECTS*	Object i dies because c propagated an asynchronous exception
notDie.c	c : *CALLS*	*EXCEPTIONS* *ALLSCHEDULERS*	Object i has not died (at least, yet)
deadObject.$i.c$	c : *CALLS* i : *OBJECTS*	*ALLDOCALLS* *DEADOBJECTS* *ALLCALLSDONE*	Call c is told that object i is dead
terminate		*ALLSCHEDULERS*	All objects are prepared to terminate
tooManyCalls		*CALLCOUNT*	Too many calls for this instance

A.2 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
$successful.a.b$ $failed.a.b$	$a : CALLS$ $b : CALLS$	$ALLDOCALLS$ $EXCEPTIONS$	Call b is told that call a has was successful or failed
$reserved.c.s$ $notReserved.c.s$	$c : CALLS$ $s : \mathcal{P}(OBJECTS)$	$RESERVATIONS$ $ALLDOCALLS$	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
$unschedule.j.c$	$j : OBJECTS$ $c : CALLS$	$ALLDOCALLS$ $ALLCALLSDONE$ $ALLSCHEDULERS$ $EXCEPTIONS$	Call c on object j has finished
$okaySoFar.c$ $failSoFar.c$	$c : CALLS$	$ALLDOCALLS$ $EXCEPTIONS$	A way to get the current fail/retry state of a call
$notAdding.c$	$c : CALLS$	$ALLDOCALLS$ $ALLCALLSDONE$	Indicates the call c will not be adding, so treat it as complete.

References

[AENV04] Arslan V, Eugster P, Nienaltowski P, Vaucouleur P (2004) SCOOP: concurrency made easy. In: Proc. dependable systems: software, computing, networks. Originally a draft paper from ETH Zürich, http://se.inf.ethz.ch/people/nienaltowski/papers/scoop_easy_draft.pdf

[BP06] Brooke PJ, Paige RF (2006) A critique of SCOOP. In: Paige RF, Brooke PJ (eds) Proc. first international symposium on concurrency, real-time, and distribution in eiffel-like languages (CORDIE), number YCS-TR-405. University of York, York

[BP07a] Brooke PJ, Paige RF (2007) Exceptions in concurrent Eiffel. J Object Technol 6(10):111–126

[BP07b] Brooke PJ, Paige RF (2007) Lazy exploration and checking of CSP models with CSPsim. In: McEwan AA, Ifill W, Welch PH (eds) Communicating process architectures, pp 33–50

[BPJ07] Brooke PJ, Paige RF, Jacob JL (2007) A CSP model of Eiffel's SCOOP. Formal Aspects Comput 19(4):487–512

[Car93] Caromel D (1993) Towards a method of object-oriented concurrent programming. CACM 36(9): 99–102

[Com00] Compton M (2000) SCOOP: an investigation of concurrency in Eiffel. Master's thesis, Australian National University

[CZT] Community Z (2007) Tools project. http://czt.sourceforge.net/. Last visited September 2007

[ECM05] ECMA-367: Eiffel analysis, design and programming language. ECMA International, June 2005, Geneva

[FC06] Freitas AF, Cavalcanti ALC (2006) Automatic translation from Circus to Java. In: FM 2006, number 4085 in LNCS. Springer, Heidelberg

[FOP04] Fuks O, Ostroff JS, Paige RF (2004) SECG: the SCOOP-to-Eiffel code generator. J Object Technol 3(10):143–160

[Hoa85] Hoare CAR (1985) Communicating Sequential Processes. Prentice-Hall International UK, New Jersey

[MD02] Mahony B, Dong JS (2002) Deep semantics links of TCSP and Object-Z: TCOZ approach. Formal Aspects Comput 13(2): 142–160

[Mey97] Meyer B (1997) Object-oriented software construction, 2nd edn. Prentice Hall, New Jersey

[Nie05] Nienaltowski P SCOOPLI implementation, 2005. http://se.inf.ethz.ch/research/scoop.html

[Nie07] Nienaltowski P (2007) Practical framework for contract-based concurrent object-oriented programming. PhD Thesis, ETH Zürich

[NMO08] Nienaltowski P, Meyer B, Ostroff JS (2008) Contracts for concurrency. Formal Aspects Comput. doi:10.1007/s00165-007-0063-2
[OTHS08] Ostroff JS, Torshizi F, Huang HF, Schoeller B (2008) Beyond contracts for concurrency. Formal Aspects Comput. doi:10.1007/s00165-008-0073-8
[PO04] Paige R, Ostroff J (2004) ERC—an object-oriented refinement calculus for Eiffel. Formal Aspects Comput 16(1)
[Ros98] Roscoe AW (1998) The Theory and Practice of Concurrency. Series in Computer Science. Prentice Hall, New Jersey
[Sch00] Schneider S (2000) Concurrent and real-time systems. Wiley, New York
[TD97] Taft T, Duff RA (eds) (1997) Ada 95 Reference Manual. Number 1246 in Lectures Notes in Computer Science. Springer, Heidelberg
[WC02] Woodcock J, Cavalcanti A (2002) The semantics of Circus. In: Proc. ZB, pp 184–203
[Wel06] Wellings A RECOOP: Real-time concurrent programming with Eiffel. In: Paige RF, Brooke PJ (eds) Proc. first international symposium on concurrency, real-time, and distribution in Eiffel-like languages (CORDIE), number YCS-TR-405, July 2006

Received 17 April 2007
Accepted in revised form 1 October 2008 by Dong Jin Song and J.C.P. Woodcock
Published online 29 November 2008