A lazy CSP simulator for a concurrent OO model

Phil Brooke

University of Teesside
pjb@scm.tees.ac.uk

Overview

- Eiffel
- Concurrency in Eiffel (SCOOP)
- CSP model of SCOOP
- SCOOP and CSPsim
- Description of CSPsim and some internal issues
- Alternative model for concurrency in Eiffel
- Comparison with CSP tools
- Future plans

Eiffel

Eiffel is an OO language due to Bertrand Meyer that takes many lessons from the formal methods world (and calls this 'design by contract')

Has typical OO constructs: classes, objects, inheritance, associations, composite ("expanded") types, generic (parameterised) types, polymorphism and dynamic binding, and automatic memory management

- Classes can declare invariants that must be true at all *stable* points
- Routines can have preconditions (*require*) and postconditions (*ensure*)
- There is one loop structure: this has invariants and variants

Eiffel (2)

Eiffel is intended to be a language that enables

- reusability;
- extendibility; and
- reliability.

Additional factors include • efficiency; • openness; and • portability. (So said Eiffel Software's web page some time ago. . .)

- One main compiler: EiffelStudio (recently available under an open source licence)
- Other compilers: SmartEiffel (*a.k.a.* SmallEiffel), gec, a few other minor commercial players
- Standard-ish libraries (Gobo, EiffelBase)

Example program

```
class HELLO

create
  make

feature

  make is
    do
      io.put_string ("Hello!%N")
    end

end
```

Eiffel (3)

Encourages a particular style of programming (command-query separation) — but this isn't enforced

Language is standardised through ECMA (367), going for ISO

The language does not have real-time features

Eiffel does have concurrency — via a threading library (or calling out to the OS, *e.g.*, fork). But no language-level concurrency (except for SCOOP. . .)

Concurrent objects

Robin Milner is reported to have said (in early 1990s)

> "I can't understand why objects are not concurrent in the first place (in OO-languages such as Smalltalk)."

(Matsuoka 1993)

SCOOP

SCOOP = 'Simple Concurrent Object Oriented Programming'

Adds one keyword: **separate**

May be applied to

- definition of a class: **separate class** FOO
- declaration of an entity (a variable): x : **separate** FOO
- formal routine argument: $f(y$: **separate** $FOO)$

Separate objects have their own thread of control (conceptually)

SCOOP semantics

Access to an entity or formal argument in (non-SCOOP) Eiffel is synchronous

Access in SCOOP is

- asynchronous for routine (*a.k.a.* procedure) calls
- synchronous for function calls (but can be lazy evaluation — Caromel's *wait-by-necessity* mechanism)

Races are prevented by the convention that separate calls require the target to be locked — called *reservation* in SCOOP. Separate formal arguments to a feature are locked

SCOOP processors

Distinct from CPUs!

A SCOOP *processor* has a single (real) thread of control (from a real CPU, an OS process or thread — we call these *partitions*). Each object is handled by a single processor

In general, systems have many processors (nasty potential multi-programming problem)

There is also a notion of *subsystems*: a processor together with the objects it handles

Preconditions and waiting

Sequential Eiffel: a precondition that evaluates to false causes an exception to propogate

In SCOOP, a precondition is a guard — the feature waits until the guard is true (*e.g.*, a consumer process waiting for its input buffer to be non-empty)

Example: buffer-consumer in SCOOP

```
class ROOT_CLASS creation
        make
feature

        b: separate BUFFER[INTEGER]
        p: PRODUCER
        c: CONSUMER

        make is
                do
                        create b.make
                        create p.make(b)
                        create c.make(b)
                end
end -- ROOT_CLASS
```

Example: buffer-consumer in SCOOP

```
separate class PRODUCER creation
        make
feature
        buffer : separate BUFFER[INTEGER]

        make (b: separate BUFFER[INTEGER]) is do buffer := b; keep_producing end

        keep_producing is
        do
                ...
                store_in_buffer (buffer, i)
                ...
        end

        store_in_buffer (b : separate BUFFER[INTEGER]; i:INTEGER) is
                require
                        not b.is_full
                do
                        b.put(i)
                ensure
                        b.has(i)
                end
end -- PRODUCER
```

Example: buffer-consumer in SCOOP

```
separate class CONSUMER creation
        make
feature
        buffer : separate BUFFER[INTEGER]

        make (b: separate BUFFER[INTEGER]) is do buffer := b; keep_consuming end

        keep_consuming is
        local
                ...
        do
                ...
                i := consume_from_buffer( buffer )
                ...
        end

        consume_from_buffer (b: separate BUFFER[INTEGER]) : INTEGER is
                require
                        not b.is_empty
                do
                        Result := b.item
                        b.remove
                ensure
                        b.count = old b.count - 1
                end

end -- CONSUMER
```

Example: buffer-consumer in SCOOP

```
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

end −− BUFFER
```

CSP model

Sometime in 2004: attempted to write a simple CSP model of SCOOP

The model of SCOOP (now) has 12 major components composed with alphabetised parallel. The state space of some of these components is large

(More details of this problem are in a November 2007 FACJ paper)

CSP model (2)

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

CSP model (3)

- *ALLOBJECTS*: creation of objects, records their handler, allows locking after creation
- *RESERVATIONS*: records which calls have locked which objects, enables further locking
- *ALLDOCALLS*: represents the execution of each possible call
  A particular call may also refer to the processes *ADDCALL* (enqueue a call) or *ENDFEATURECALLS* (wait for calls to end — maybe)
- *ALLSCHEDULERS*: enqueueing of calls against objects and causes *ALLDOCALLS* to start executing
- *RESCHEDULER*: prevents suspended calls restarting until all the blocking objects have been freed

CSP model (4)

- $BIGLOCK$: a simple mutex to prevent races while locking objects
- $CALLCOUNT$: assigns a unique number to each call
- $ALLCALLSDONE$: records that a call has been completed
- $CALLSEPPARAMS$: records the objects that should be locked for a call
- $CALLPARAMS$: records the actual arguments for each call
- $OBJECTLOCALS$: records the state of variables within an object
- $SEQOBJECTS$: imposes an ordering on the creation of objects

CSP model (5)

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
- $rootClass$ and $rootFeature$ indicate which class and feature start the execution of the system

Example translation

```
class A

  p (a_x : separate X) is do a_x.g1; a_x.g2; a_x.g3; a_x.g4 end

end

class H

  r (a_x : separate X) is do ...; a_x.f; ... end

end

-- An object b of type B is the root object, with m as the creation procedure.

class B
creation m

  a : A
  h : H
  x : separate X

  m is do create a; create h; create x; q end

  q is do a.p(x); h.r(x) end

end
```

CSP for feature `p` of class `A`

$$
\begin{array}{l}
WORK(c, i, f) \triangleq \\
\quad startWork.c.i.f \rightarrow (\\
\quad f = f_A_p \Rightarrow (getParam.c.1.a_x \rightarrow \\
\quad\quad callCount.c.c_1 \rightarrow ADDCALL(c \leftarrow c, c' \leftarrow c_1, i \leftarrow i, j \leftarrow a_x, f \leftarrow f_X_g1); \\
\quad\quad callCount.c.c_2 \rightarrow ADDCALL(c \leftarrow c, c' \leftarrow c_2, i \leftarrow i, j \leftarrow a_x, f \leftarrow f_X_g2); \\
\quad\quad callCount.c.c_3 \rightarrow ADDCALL(c \leftarrow c, c' \leftarrow c_3, i \leftarrow i, j \leftarrow a_x, f \leftarrow f_X_g3); \\
\quad\quad callCount.c.c_4 \rightarrow ADDCALL(c \leftarrow c, c' \leftarrow c_4, i \leftarrow i, j \leftarrow a_x, f \leftarrow f_X_g4); \\
\quad\quad ENDFEATURECALLS(c \leftarrow c, C \leftarrow \{c_1, c_2, c_3, c_4\})) \\
\quad \Box\; f = f_H_r \Rightarrow \ldots \\
\quad \vdots \\
\quad); endWork.c.i.f \rightarrow Skip
\end{array}
$$

Motivation for CSPsim

Built an FDR model. . .

The problem appeared too large for FDR2. ProBE was some use for exploring, but had some UI problems

Arbitrary limits can be placed on lengths of queues, but this helps us little (and makes it harder to explain to Eiffel specialists)

However, the resulting state space of the overall system is (relatively) small. . .

CSPsim: Current status and intention

Implemented in Ada. ‘Clients’ are also implemented in Ada
UI is basic, line-oriented with tab-completion

Initial aim: explore systems with large components

Subsequently broadened into an exhaustive checker (but only for systems that have a *small* resulting state space, even if intermediate components are conceptually large)

Working on symbolic/unification-style description of events (more shortly)

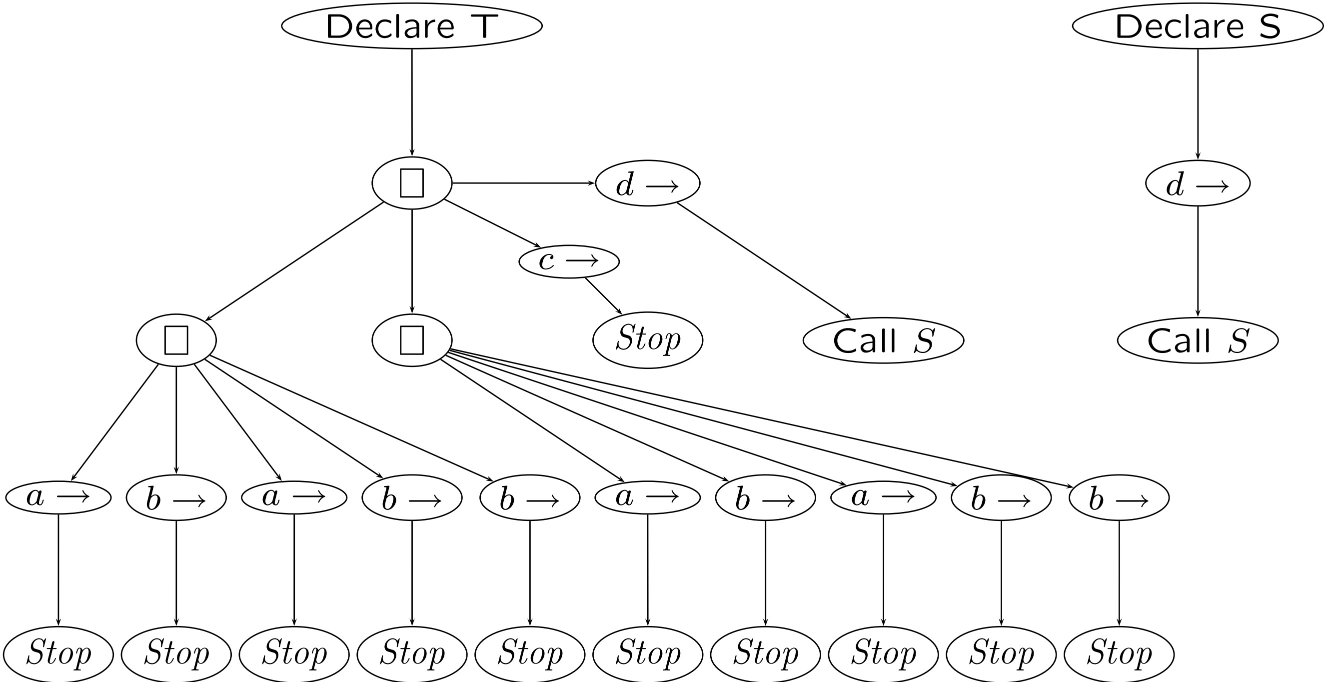
Current code is embarrassingly messy and plagued by memory issues (easily fixed, but needs redesign from scratch)

Process description

- Processes are also described in Ada (via a `Factory` façade). Non-trivial expressions are handled via user Ada code (*e.g.*, to update parameters)
- Processes are described as an acyclic graph
- Processes can be given a name (used for recursion and constructing loops)
- All processes can carry parameters. The parameters are named and typed (integers, strings, arrays of strings, non-rectangular 2d arrays of strings)
  Names and types are checked when processes are 'called'

Example process

$$
\begin{aligned}
T \;=\; & \quad (a \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop} \;\Box\; a \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop}) \\
& \Box\; (a \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop} \;\Box\; a \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop} \;\Box\; b \rightarrow \mathit{Stop}) \\
& \Box\; c \rightarrow \mathit{Stop} \;\Box\; d \rightarrow S \\
S \;=\; & \; d \rightarrow S
\end{aligned}
$$



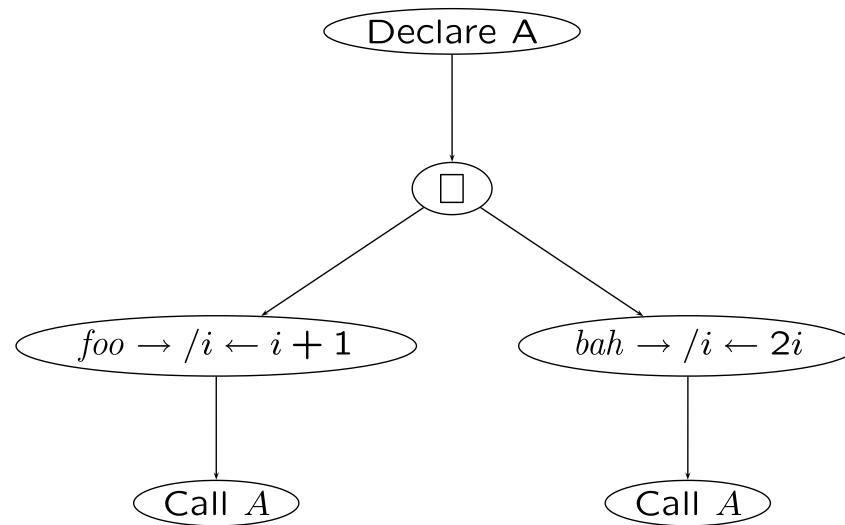
Example process (Ada)

```
with Factory;        use Factory;
with Processes;      use Processes, Processes.PXA;

procedure Test2b is
begin
   Decl("TEST2b",
        ExtChoice(+ ExtChoice(+ Prefix("a", Stop)
                              + Prefix("b", Stop)
                              + Prefix("a", Stop)
                              + Prefix("b", Stop)
                              + Prefix("b", Stop))
                  + ExtChoice(+ Prefix("a", Stop)
                              + Prefix("b", Stop)
                              + Prefix("a", Stop)
                              + Prefix("b", Stop)
                              + Prefix("b", Stop))
                  + Prefix("c", Stop)
                  + Prefix("d", Call("SLOOP"))));
   Decl("SLOOP",Prefix("d", Call("SLOOP")));
   Explore;
end Test2b;
```

Example process

$$A(i) = foo \rightarrow A(i+1) \Box bah \rightarrow A(2i)$$



```
with Factory;                      use Factory;
with Parameters;                   use Parameters, Parameters.PMA;
with Parameters.Integers;          use Parameters.Integers;
with Processes;                    use Processes, Processes.FPA, Processes.PXA;
with Testlib;                      use Testlib;

procedure Test2c is
begin
   Decl("TEST2c",
        Call("A",
             PS => +Integer_Param("i", 0)));
   Decl("A",
        +Integer_Formal("i"),
        ExtChoice(+Prefix("foo", Call("A"), PU => Test2ca_PU'Access)
                  +Prefix("bah", Call("A"), PU => Test2cb_PU'Access)));
   Explore;
end Test2c;
```

```
./test2c.exe -select TEST2c

CSPsim, release 0.7.3    Copyright (C) 2006-2007 Phillip J. Brooke

Exploring process A()

Events available : bah foo   (2 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .currentParams
Current process: <<i=0>>(<<i=0>>foo -> (A())) [] (<<i=0>>bah -> (A()))
Explore> foo
Taking event 'foo'
 1 event(s) in trace so far


Events available : bah foo   (2 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .currentParams
Current process: <<i=1>>(<<i=1>>foo -> (A())) [] (<<i=1>>bah -> (A()))
Explore> bah
Taking event 'bah'
 2 event(s) in trace so far


Events available : bah foo   (2 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .currentParams
Current process: <<i=2>>(<<i=2>>foo -> (A())) [] (<<i=2>>bah -> (A()))
Explore> bah
Taking event 'bah'
```

```
 3 event(s) in trace so far


Events available : bah foo   (2 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .currentParams
Current process: <<i=4>>(<<i=4>>foo -> (A())) [] (<<i=4>>bah -> (A()))
Explore> .maxSteps
Integer> 20
Explore> .randomRunMode
  [.......]
Taking event 'bah'
 24 event(s) in trace so far


Events available : bah foo   (2 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .currentParams
Current process: <<i=9348>>(<<i=9348>>foo -> (A())) [] (<<i=9348>>bah -> (A()))
Explore> .trace
<foo, bah, bah, bah, bah, foo, bah, bah, foo, foo, foo, foo, bah, bah,
foo, foo, foo, foo, bah, bah, bah, foo, bah, bah>
Explore>
```

Prefix

Prefix is the only way to introduce events. Four variations of prefix are available in CSPsim:

1. A single, fixed event is offered by $a \rightarrow P$
2. Any event from a fixed, given set is offered by $a : A \rightarrow P$
3. Any event is offered from a set that is calculated at run-time:

   $$a : F \rightarrow P$$

   where F returns a set of events (given the current parameters). This is used when the events to be offered are unknown at compilation time, *e.g.*,

   $$A(i) = read.i \rightarrow \ldots$$

Prefix (2)

4. Any event a such that a particular function F returns true for that a — used when it is very expensive or impossible to calculate a full set of offered events (for example, there could be an infinite number of acceptable events, say, any integer)

All variants can update parameters when an event is taken

Nondeterminism and hiding

The operational model causes complications with nondeterminism, *e.g.*,

$$a \rightarrow P \,\Box\, a \rightarrow Q$$

The environment has no control over 'which' a is chosen. But simulators must resolve the decision...

Dummy events are inserted into the trace recording which arm was selected (user choice or random)

```
$ ./test2b.exe -select TEST2b
CSPsim, release 0.7.3    Copyright (C) 2006-2007 Phillip J. Brooke

 Exploring process ((a -> (STOP)) [] (b -> (STOP)) [] (a -> (STOP)) []
 [... etc ...]

Events available : a b c d   (4 event(s))
Type the event to take or a '.' command (try '.help').
Explore> a
Taking event 'a'

This event is nondeterministic over up to 4 arms.
Matches in ExtChoice in arms 1 2  (0..1)
Take event:>|Integer> 1

This event is nondeterministic over up to 5 arms.
Matches in ExtChoice in arms 1 3  (0..1)
Take event:>/Integer> 3

Take event:(2.0 seconds)
 3 event(s) in trace so far

No events available !  (deadlock)
Type the event to take or a '.' command (try '.help').
Explore> .trace
<a, _NDR_1, _NDR_3>
Explore>
```

```
Explore> .reset
Resetting ...
Exploring process ((a -> (STOP)) [] (b -> (STOP)) [] (a -> (STOP)) []
 [... etc ...]

Events available : a b c d   (4 event(s))
Type the event to take or a '.' command (try '.help').
Explore> .nondetRandom
Set nondeterminism resolver to 'random'
Explore> a
Taking event 'a'
 Nondeterministic event over up to 4 arms.
Matches in ExtChoice in arms 1 2  (0..1)
 Nondeterministic event over up to 5 arms.
Matches in ExtChoice in arms 1 3  (0..1)
 3 event(s) in trace so far


No events available !  (deadlock)
Type the event to take or a '.' command (try '.help').
Explore> .trace
<a, _NDR_2, _NDR_1>
Explore>
```

Optimisations

Events are stored as integers with hash structures to speed up the questions "Is this event already known?" and "What integer represents this event?"

Some CSP laws are used to reduce the state space (much more to do here)

Deleting redundant arms of external choice, replacing prefix and guard processes with their successor

Differential updates: parallel operators can re-use caches of previous calculations and determine the changes instead of calculating from scratch. Sometimes faster, sometimes slower — lots of changes in underlying processes makes this slow. Need a heuristic to guess when to enable and disable this mechanism

Exhaustive search

CSPsim started out as naïve simulator. Features such as replay and (repeated) random walks generally useful in the motivating problem

An exhaustive search mode was added, exploiting save/restore of state (available in general UI)

Simple search algorithm: start with initial state, create queue of states to explore. States are saved on disk (in a normalised plaintext format). Continues until no more states to explore, or limits are reached

Post-processing and ‘dot’ can produce graphical output

Exhaustive search (2)

This is very slow relative to FDR2, but did realise results for the problem

More recent examples from the original problem are too large for this explorer — limited by either a memory leak (most fixed) or more recently, running out of disk for caching (needs compressing)

Optimisations are usually added when bored of waiting for it to finish. . .

CSPsim and SCOOP model

Produced some useful examples. Conclusions are that

- Callers should not wait for child calls to complete $\because$ waiting substantially reduces parallelism, does not increase safety, and makes callback deadlocks more likely
- Lock-passing must be allowed, but inconclusive on direct *vs.* indirect
- Identified other issues (*e.g.*, deadlock, termination, run-time system and resource failure) while constructing model

Alternative models

SCOOP is exclusively message-passing

Others have suggested memory-sharing approaches

Suggested an alternative model such that existing (sequential) programs to have the same behaviour as now *without changing the text of the program*

- We start from sequential Eiffel, rather than from SCOOP
- We want to preserve as much use of existing libraries as possible even when used in a concurrent context

Alternative model (2)

Particular SCOOP issues:

- Reservations need to be passed on to prevent trivial deadlock
- Subsystems/processors directly affect the amount of possible concurrency and introduce further possible deadlocks
- Separate-ness is complicated, introduces *traitors* and overloading of formal arguments to indicate locks

Alternative model (3)

1. Each object (including expanded) has a notional thread of control

2. All objects have a FIFO queue of feature calls made by other objects; at most one feature call can be executing on an object at any one time

3. *All calls are synchronous unless* unless qualified with **async**, *e.g.*, **async** $a.f$

   Function calls result in *one or two* entries being enqueued. Procedure calls only require one entry

   Creation procedures can be **async**

   We disallow the use of **async** on an unqualified call, since we do not (yet) address multiple threads of control in an object

Alternative model (4)

4. Feature calls can only be made on locked objects, *i.e.*, **Current** holds the 'active lock' on the callee

5. All objects given as formal arguments are locked unless explicitly excluded using the **unlocked** keyword in 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

Alternative model (5)

6. 'Lazy locks' are automatically made on unlocked objects that are the subject of a feature call for only the duration of that call

   Lazy locks are viewed as a call to an implicit wrapper on the caller; this wrapper has the same preconditions as the called feature

   This allows existing sequential code to work unchanged, even though an 'active lock' is required to enqueue feature calls on other objects

Alternative model (6)

7. Locks are passed on through call chains (as previously proposed by us). 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.

8. Wait conditions and suspension of calls applies as in SCOOP

    If 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

    Future work (dealing with real-time programming) may impose or suggest further scheduling policies.

Alternative model (7)

9. The developer should choose whether asynchronous exceptions* cause an object to either

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

* *Exceptions in concurrent Eiffel.* deals with this part of the proposal in more detail. To appear in JOT

Alternative model (8)

10. 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

11. A problem in the underlying partition communication system that prevents communication with an object results in the exception 'partition_communication_failure'

12. 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

Example: alternative buffer-consumer

```
class ROOT_CLASS creation
        make
feature

        b: BUFFER[INTEGER]
        p: PRODUCER
        c: CONSUMER

        make is
                do
                        create b.make
                        async create p.make(b)
                        async create c.make(b)
                end
end -- ROOT_CLASS
```

Example: alternative buffer-consumer

```
class PRODUCER creation
        make
feature
        buffer : BUFFER[INTEGER]

        make (b: unlocked BUFFER[INTEGER]) is do buffer := b; keep_producing end

        keep_producing is
        do
                ...
                buffer .put( i )
                ...
        end

end -- PRODUCER
```

Example: alternative buffer-consumer

```
class CONSUMER creation
        make
feature
        buffer : BUFFER[INTEGER]

        make (b: unlocked BUFFER[INTEGER]) is do buffer := b; keep_consuming end

        keep_consuming is
        local
                ...
        do
                ...
                i := consume_from_buffer( buffer )
                ...
        end

        consume_from_buffer (b: BUFFER[INTEGER]) : INTEGER is
                require
                        not b.is_empty
                do
                        Result := b.item
                        b.remove
                ensure
                        b.count = old b.count - 1
                end

end -- CONSUMER
```

Example: remarks

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
- In ROOT_CLASS.make, the creation of the buffer is unadorned, but the creation of the producer and consumer objects are annotated with **async** indicating that the calls to their creation procedures should be asynchronous calls

Example: remarks (2)

- 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

- store_in_buffer is not required in the alternative version, since it merely exists to obtain a lock on the buffer and call put on the buffer. A lazy lock suffices here

- A lazy lock is not sufficient to replace consume_from_buffer

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 a multi-item store_in_buffer would be required

CSP outline of alternative model

Closely based on our proposed model of SCOOP: the major difference are

- There are no subsystems/processors
- Scheduling is still similar, but moved to objects
- Explicit events for object creation, start/end of work, reserving/freeing, *etc.*.

A simpler model

The alternative model does not contain subsystems/processors. A mechanism is included to allow objects to be given in argument lists without implicitly reserving them

Simpler $\because$ all objects are notionally '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

Why generalise CSPsim?

Theorem prover approaches work for a small class of problems (typically inductive) — *e.g.*, PVS and ACL2

FDR2 *very* good for problems of a suitable size (not our original problem! — issues with composition of large components)

Larger systems might be explorable (and inefficiently searched) by lazy methods such as CSPsim

Symbolic 'binder' is necessary for further optimisation (the buffer-consumer example is too big now)

All three approaches are complementary

Symbolic events

In-progress: Want to symbolically calculate the intersection of 'large' sets, *e.g.*, for i and j integers, $a.i.1 \rightarrow P$ offers many events as does $a.2.j \rightarrow Q$, but the parallel composition offers only $a.2.1$

This will avoid passing large event sets in some cases. Most of the code for this is complete (but then, I thought that at CPA)...

Internal test example:

```
BA  := + C("a") + C(2) + B_Int("i", +C_Int);
BA2 := + C("a") + B_Int("j", +C_Int) + C(1);
```

```
Exp9
1. a.2.?(i : Int), 2. a.?(j : Int).1
++++>> a.2.1
```

'Binder' patterns — constructors

```
-- *** Constructors    ************************************************

-- B_Int will attempt to bind the pattern BA to integer parameter key K.
function B_Int (K : UBS; BA : Bind_Array_Access) return Bind_Element_Access;
function B_Int (K : String; BA : Bind_Array_Access) return Bind_Element_Access;

-- B_Str will attempt to bind the pattern BA to stringparameter key K.
function B_Str (K : UBS; BA : Bind_Array_Access) return Bind_Element_Access;
function B_Str (K : String; BA : Bind_Array_Access) return Bind_Element_Access;

-- Accept only integer X here.
function C (X : Integer) return Bind_Element_Access;
             -- is Bind_Integer_Range, used to be Bind_Integer

-- Accept an integer in the closed range I_Min to I_Max.
function C (I_Min, I_Max : Integer) return Bind_Element_Access;
                                                    -- Bind_Integer_Range

-- Accept any integer.
function C_Int return Bind_Element_Access;  -- Bind_Any_Integer

-- Accept the string X.
function C (X : UBS) return Bind_Element_Access;
function C (X : String) return Bind_Element_Access;
           -- used to be Bind_String, but merged into Bind_Choose_String

-- Copy or link to the given UBS array.  This will accept any
-- string in the given array.
function C_Copy (X : UBS_Array_Access) return Bind_Element_Access;
function C_Alias (X : UBS_Array_Access) return Bind_Element_Access;
                                               -- Bind_Choose_String
```

‘Binder’ patterns — constructors

```
-- Accept any string -- except for those with dots.
function C_Str return Bind_Element_Access;    -- Bind_Any_String

-- Accept any string -- including dots.  This should only be used
-- at the very end, since it will greedily eat the remaining
-- string.
function C_Tail return Bind_Element_Access;  -- Bind_Tail_String

-- Accept the string representation of a current parameter (integer
-- or string) with key K.
function P (K : UBS) return Bind_Element_Access;
function P (K : String) return Bind_Element_Access;
                                                   -- Bind_Param_Image

-- Normally, an array of Bind_Elements is separated by dots.  If
-- you don't want this, group them using 'S', which concatenate
-- elements together without dots.  This cannot cope with binding
-- or open element types!
function S (BA : Bind_Array_Access) return Bind_Element_Access;
```

(from binder.ads)

Binder issues

General problem: Given two binder patterns, return another binder pattern representing their intersection. Generalise to two *lists* of binder patterns. . .

Arbitrary patterns are hard to cope with, particularly where the set used by Bind_Choose_String may contain arbitrary strings (*i.e.*, different number of dots). Two partially-overlapping Bind_Choose_Strings is very difficult, *e.g.*

$$\begin{array}{l} a\,.\,b\,.\,x\,.\longleftarrow c \longrightarrow \\ a\,.\longleftarrow d \longrightarrow\ .\,y \end{array}$$

where

a chooses from a fixed lists of strings	A (0 dots in each string)
b	B (0 dots)
c	C (1 dot!)
d	D (2 dots!)
x, y are match/bind	

Binder issues (2)

A normalise function can break up some of these difficult sets to make subsequent intersection easier

The final option is to return ‘unknown’ — *i.e.*, the current code cannot resolve the problem (so fallback to expanding lists of events)

Binder issues (3)

Not currently built into the main event handling code, as it requires modification of much code...

Particularly nasty in terms of memory management (again — aliasing problems to potentially large sets — or too many copies of the same set)

Question: Is this an acceptable approach? Are there better ways?

(Aside: failing laptop battery + hibernation + cryptoloop partitions is a good way to lose work)

CSPsim: Summary

- Lazy evaluation allows large problems to be handled
- Typed parameters
- Imperative calculations (via Ada)
- (near future) Better handling of ‘big’ sets of events

but

- Very slow for exhaustive search
- Requires Ada compilation

Best viewed as a prototype in need of re-implementation. . .

Comparison

The major alternative is FDR2 (and ProBE)

For search, FDR2 is very fast — provided it can construct a suitable model. It can do refinement (at best, we can simulate trace refinement)

ProBE can explore (and has the same notation as FDR2), but the GUI is limited (*e.g.*, unordered events, no replay)

Other model checkers, *e.g.*, Bogor, SPIN, using different notations. CSP particularly suited our motivating problem

Future plans

- FDR2-style τ-loop compression
- Improve UI (especially the front-end); input language
- Integrate with FDR2 and ProBE to select best tool for each (part of a) problem?
  - More general — add links to theorem provers previous work in PVS c. 1997, some also in ACL2
  - Some similarities to CIRCUS project
  - EMF metamodel
- Implement the symbolic/unification-style resolution (for parallel composition and other places)
- Refinement checking, refusals, more algebraic rules, . . .
- Exploit Ada tasking and add distributed processing
- Need to arrange funding — this is too big now ($>$19k SLoC + $\approx$1300 SLoC tests + $\approx$4400 SLoC for SCOOP)

Reimplement — particularly to make better use of fat pointers, ‘immutable’ models of sets, . . .

The end

Demo?

Any questions (comments, suggestions, . . .)?

Email: pjb@scm.tees.ac.uk