Adapted from an original published in Communicating Process Architectures 2008
Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.)
IOS Press, 2008

Communicating Scala Objects (2017 Revision)

Bernard SUFRIN

Oxford University Department of Computer Science
and
Worcester College,
Oxford OX1 2HB, England

Bernard.Sufrin@cs.ox.ac.uk

Abstract. In this paper we introduce the core features of CSO (Communicating Scala Objects) – a notationally convenient embedding of the essence of occam in a modern, generically typed, object-oriented programming language that is compiled to Java Virtual Machine (JVM) code.

This revision uses CSO notation compatible with version 1.2 of CSO.

Keywords. occam model, concurrency, Scala, JCSP.

Introduction

On the face of it the Java virtual machine (JVM) is a very attractive platform for realistic concurrent and distributed applications and systems. On the other hand, the warnings from at least parts of the "Java establishment" to neophyte Java programmers who think about using threads are clear:

> *If you can get away with it, avoid using threads. Threads can be difficult to use, and they make programs harder to debug.*
>
> *It is our basic belief that extreme caution is warranted when designing and building multi-threaded applications ... use of threads can be very deceptive ... in almost all cases they make debugging, testing, and maintenance vastly more difficult and sometimes impossible. Neither the training, experience, or actual practices of most programmers, nor the tools we have to help us, are designed to cope with the non-determinism ... this is particularly true in Java ... we urge you to think twice about using threads in cases where they are not absolutely necessary ...[8]*

But over the years a number of Java libraries [7,3,4,1,2] have demonstrated that the occam programming model can be used very effectively to provide an intellectually tractable *discipline* of concurrent Java programming that is harder to achieve by those who rely on the lower level, monitor-based, facilities provided by the Java language itself.

So in mid-2006, faced with teaching a new course on concurrent and distributed programming, and wanting to make it a *practical* course that was easily accessible to Java programmers, we decided that this was the way to go about it. We taught the first year of this course using a Java library.[1] [†]

Our students' enthusiastic reaction to the occam model was as gratifying as their distaste for the notational weight of its embedding in Java was dismaying. Although we discussed *designs* for our concurrent programs using a CSP-like process-algebra notation and a simplified form of ECSP [5,6], the resulting *coding gap* appeared to be too much for most of the students to stomach.

At this point one of our visiting students introduced us to Scala [9], a modern object-oriented language that generates JVM code, has a more subtle generic type system than Java, and has other features that make it very easy to construct *domain-specific languages* – libraries that appear to be notational extensions.

After toying for a while with the idea of using Scala's Actor library [12,14], we decided instead to develop a new Scala library to implement the occam model independently of existing Java libraries,[2] and of Scala's Actor library.[3] Our principal aim was to have a self-contained library we could use to support subsequent delivery of our course (many of whose examples are toy programs designed to illustrate patterns of concurrency), but we also wanted to explore its suitability for structuring larger scale Scala programs.

This paper is an account of the most important features of the core of the Communicating Scala Objects (CSO) library that emerged. We have assumed a little familiarity with the conceptual and notational basis of occam and and some familiarity with Scala.

1. Processes

A CSO process is a value with Scala type PROC and is what an experienced object oriented programmer would call a *stereotype* for a thread. When a process is *started* any fresh threads that are necessary for it to run are acquired from a pool; they are returned to the pool when the process terminates.[4]

1.1. Process notation

Processes (p : PROC) are first-class Scala values, denoted by one of the following forms of expression:

1a. **proc** $\{ \; expr \; \}$
1b. **proc** $(name : String) \; \{ \; expr \; \}$
A simple process ($expr$ must be a command, *i.e.* have type Unit)
If a $name$ is not given explicitly, one is automatically associated with the process value as it is constructed.
2. $p_1 \; || \; p_2 \; || \; ... \; || \; p_n$
A parallel composition of n processes (each p_i must have type PROC)
3. $|| \; collection$
Parallel composition of a finite collection of PROC values.
When $collection$ comprises $p_1 ... p_n$ this is equivalent to $p_1 \; || \; p_2 \; || \; ... \; || \; p_n$.

A frequently-occuring pattern of this latter form of composition is one in which the collection is an iterated form, such as: || (**for** (i←0 until n) **yield** $p(i)$). This form denotes a process equivalent to: $p(0) \; || \; p(1) \; || \; ... \; || \; p(n-1)$,

[†]Notes appear on page 19.

1.2. Starting and running processes

If p is a process, then evaluation of the expression $p()$ runs the process.[5] The following cases are distinguished:

1a. p is **proc** $\{$ $expr$ $\}$
1b. p is **proc** $(name : String)$ $\{$ $expr$ $\}$

- $p()$ causes $\{$ $expr$ $\}$ to be evaluated in the current thread (*ie.* the thread that started the evaluation of $p()$).
- The process as a whole terminates when the evaluation of $\{$ $expr$ $\}$ terminates or throws an (uncaught) exception.
- The behaviour of the expression p() cannot be distinguished from that of the expression {expr}, except that its $name$ is used to identify the thread running p until p terminates. This identification can be helpful when inspecting a running CSO program using the CSO debugger.

1. p is $p_1 \mid\mid p_2 \mid\mid \ldots \mid\mid p_n$
   - $p()$ causes all the processes $p_1 \ldots p_n$ to be run concurrently.
   - Each of the processes except one is run in a new thread of its own; the remaining process is run in the current thread.
   - The process as a whole terminates only when *every* component p_i has terminated. But if one or more of the components terminated by throwing an uncaught exception then *when and only when they have all terminated* these exceptions are bundled into a ParException which is re-thrown, *unless they are all subtypes of* io·threadcso·process·Stopped; in which case a single io·threadcso·process·Stopped is thrown.[6]

2. Ports and Channels

2.1. Introduction

A CSO channel has two ports, one at each end, and in general is intended to transmit to its *input port* the data that is written to its *output port*. Ports are parameterized by the type of data the channel transmits, and we define the abbreviations ?[T] and ![T] respectively for InPort[T] and OutPort[T].

The most important method of an ![T] is its write method

```
! (value: T)
```

and the most frequently-used methods of an ?[T] are its read method

```
? () : T
```

and its *read and evaluate* method

```
? [U] (f: T ⇒ U) : U
```

The expression $port?(f)$ has exactly the same effect as $f(port?())$, namely to read a datum from $port$ (waiting, if necessary, for one to become available) then apply the function f to it.

The type Chan[T] is the interface implemented by all channels that carry values of type T: it is declared by:

```
trait Chan[T] extends InPort[T] with OutPort[T] { ··· }
```

This makes Chan[T] a subtype of both InPort[T] and OutPort[T]. It makes sense to think of a Chan as embodying both an InPort and an OutPort.

The implicit contract of every conventional Chan implementation is that it transmits the data written at its output port to its input port in the order in which the data is written. Different implementations have different synchronization behaviours and different restrictions on the numbers of processes that may access (*i.e.* use the principal methods of) their ports at any time. Channels may be closed in various ways, in which case they (eventually or immediately) cease to transmit data: see section 4 for a fuller discussion of this.

The CSO core comes with several predefined channel implementations, the most notable of which for our present purposes are:

- The *synchronous* channels. These all synchronize termination of the execution of a ! at their output port with the termination of the execution of a corresponding ? at their input port.[7]
  - OneOne[T] – no more than one process at a time may write to its output port or read from its input port.[8] This is the classic occam-style point to point channel. The channel stops transmitting data when it has been closed for output or closed for input.
  - N2N[T](writers: Int, readers: Int) – several different processes at a time may write to its (shared) output port and likewise several may read from its (shared) input port. Each value that is read is read by only one of the processes. The channel stops transmitting data when it has been closed for output $writers$ times or closed for input $readers$ times.
- The *buffered* channels:
  - OneOneBuf[T](n) – a one-to-one buffer of capacity n. It stops transmitting data when it has been closed for input, or when it has been closed for output and no longer has any buffered data available to input.
  - N2NBuf[T](n: Int, writers: Int, readers: Int) – a buffer of capacity n. It stops transmitting data when it has been closed for input $readers$ times, or when it has been closed for output $writers$ times and no longer has any buffered data available to input.

Access restrictions are enforced by a combination of:

- Type constraints that permit sharing requirements to be enforced statically.
  - All output port implementations that support shared access have types that are subtypes of SharedOutPort.
  - All input port implementations that support shared access have types that are subtypes of SharedInPort.
  - All channel implementations that support shared access to both their ports have types that are subtypes of SharedChannel.
  - Abstractions that need to place sharing requirements on port or channel parameters do so by declaring them with the appropriate type.[9]
- Run-time checks that offer *partial* protection against deadlocks or data loss of the kind that can could otherwise happen if unshareable ports were inadvertently shared.
  - If a read is attempted from a channel with an unshared input port before an earlier read has terminated, then an illegal state exception is thrown.

```
def producer(i: int, ![T]) : PROC = ···
def consumer(i: int, ?[T]) : PROC = ···

def mux[T] (ins: Seq[?[T]],   out: ![(int, T)]) : PROC = ···
def dmux[T](in:  ?[(int, T)], outs: Seq[![T]])  : PROC = ···


val left, right =
    for (_ ← 0 until n) yield OneOne[T] // 2 arrays of n channels
val mid = OneOne[(int, T)]              // a channel

(   || (for (i←0 until n) yield producer(i, left(i)))
 || mux(left, mid)
 || dmux(mid, right)
 || || (for (i←0 until n) yield consumer(i, right(i)))
)()
```

Program 1. A network of producers connected to consumers by a multiplexed channel

```
def producer(i: int, ![T]) : PROC = ···
def consumer(i: int, ?[T]) : PROC = ···

val con = for (_ ← 0 until n) yield OneOne[T]

(   || (for (i←0 until n) yield producer(i, con(i)))
 || || (for (i←0 until n) yield consumer(i, con(i)))
)()
```

Program 2. A network in which producers are connected directly to consumers

∗ If a write is attempted to a channel with an unshared output port before an earlier write has terminated, then an illegal state exception is thrown.

These run-time checks are limited in their effectiveness because it is might be possible for a single writer process to work fast enough to satisfy illegitimately sharing reader processes without being detected by the former check, and for the dual situation to remain undetected by the latter check.

2.2. Examples

In program 1 we show how to connect a sequence of n producers to a sequence of n consumers using a single multiplexed channel that carries values accompanied by the index of their producer to a demultiplexer that dispatches these values to the corresponding consumer. Readers familiar with JCSP may find it useful to compare this with the network illustrated in section 1.5 of [4].

As observed in that paper this isn't the most efficient way of connecting the producers to the consumers within a single JVM; and in program 2 we show a network in which producers and consumers are connected directly.

The signatures of the components producer, consumer, mux, and dmux in programs 1 and 2 specify the types of port (channel end) they require; but the subtype relation between channels and ports means that when connecting these components we can simply provide the connect-

```
def mux1[T] (ins: Seq[?[T]], out: ![(Int, T)]) : PROC =
{ val mid = N2N[(Int, T)](0, 1) // Many writers; one reader
  (  proc { while(true) { out!(mid?()) } } ||
  || (for (i←0 until ins.length) yield
          proc { while(true) ins(i) ? { v ⇒ mid!(i, v)} })
  )

}

def mux2[T] (ins: Seq[?[T]], out: SharedOutPort[(Int, T)]) : PROC =
    || (for (i←0 until ins.length) yield
            proc { while (true) {out!(i, ins(i)?())} })

def dmux[T](in:  ?[(Int, T)], outs: Seq[![T]]) : PROC = proc {
  while (true) { val (n, v) = in?(); outs(n)!v  }
}
```

Program 3. Two multiplexers and a demultiplexer

ing channels as parameters, and that the components take the required *views* of them. This means we needn't name the ports explicitly, and significantly reduces the degree of *formal clutter* in the network description.[10]

In program 3 we show how to implement two (unfair) multiplexers and a demultiplexer of the kind that might have been used in program 1.[11]

A multiplexer process generated by mux1 is the concurrent composition of a collection of "labelling" processes, each of which outputs labelled copies of its input, via an N2N[(Int,T)] channel, to a forwarding process that writes them to the out port. The forwarding process is necessary because the type-signature of mux1 does not constrain the kind of port that is passed to it as a parameter, so in programming mux1 we must assume that that the port is not shareable.

On the other hand, mux2 requires that its out parameter is shareable, so it composes a collection of labelling processes that write directly to out.

The function dmux generates demultiplexer processes that forward labelled inputs to the appropriate output ports.

3. Extended Rendezvous

3.1. Introduction

As we explained earlier, the *synchronous* channel implementations ensure that *termination* of a write (!) at their output port is synchronized with the termination of the corresponding read (?) at their input port. Although a standard read (or read-and-evaluate) terminates once the data is transferred between the writer and the reader process, an *extended rendezvous read* specifies that a computation on the transferred data is to take place *in the reader process*. It is only when this computation terminates that the read is considered to have terminated and the writing process is permitted to proceed.

The usual form of an extended rendezvous read from in: ?[T] is[12]

```
in ?? { bv ⇒ body }
```

It is evaluated by transferring a value, v, from the process at the output end of the channel (if necessary waiting for one to become ready), then applying the (anonymous) function { bv ⇒ body } to v. The read is considered to have terminated when this application has been completely evaluated. At this point the writing process is permitted to proceed and the result of the application is returned from the read.

3.2. Example: monitoring interprocess traffic

An easily understood rationale for extended rendezvous is given in [3]. We are asked to consider how to monitor the interprocess traffic between a producer process connected to a consumer process via a simple channel *without interfering with producer-consumer synchronization*. We want to construct a process that is equivalent to

```
{ val mid = OneOne[T]
  producer(mid) || consumer(mid)
}
```

but which also copies traffic on mid to a monitor process of some kind.

A first approximation to such a process is

```
{ val left, mon, right = OneOne[T]
  (  producer(left)
  || proc { repeat { val v = left?(); mon!v; right!v }
  || consumer(right)
  || monitor(mon)
  )
}
```

But this interferes with producer-consumer synchronization, because once left ?() has been executed, producer is free to proceed. More specifically, it is free to proceed before consumer reads from right. If the context in which this network of process runs is tolerant of an additional degree of buffering this is not problematic; but if it is not, then we need to be able to synchronize the read from right with the write to left.

The problem is solved by replacing the body of the copying process

```
{ val v = left?(); mon!v; right!v }
```

with a body in which the outputs to mon and right are part of an extended rendezvous with the producing process, namely:

```
{ left ?? { v ⇒ {mon!v; right!v} } }
```

The extended rendezvous is executed by reading a value from left, then applying the function { v ⇒ {mon!v; right!v} } to it. Termination of the write to left is synchronized with termination of the evaluation of the function body, so the producer writing to left cannot proceed until the consumer has read from right.

The extended rendezvous doesn't terminate until {mon!v; right!v} has terminated, but delays the output to right until the output to mon has terminated. The following reformulation relaxes the latter constraint, thereby removing a potential source of deadlock:

```
{ left ?? { v ⇒ {(proc{mon!v} || proc{right!v})()} } }
```

It is a simple matter to abstract this into a reusable component:

```
def tap[T](in: ?[T], out: ![T], mon: ![T]) =
    proc
    { repeat { in ? { v ⇒ {(proc{mon!v} || proc{out!v})()} } } }
```

3.3. Example: simplifying the implementation of synchronous inter-JVM channels

Extended rendezvous could also be used to good effect in the implementation of synchronized inter-JVM or cross-network connections, where it can keep the overt intricacy of the code manageable. Here we illustrate the essence of the implementation technique, which employs the two "network adapter" processes.

```
def copyToNet[T](in: ?[T], net: ![T], ack: ?[Unit]) =
    proc { repeat { in ?? { v ⇒ { net!v; ack?() } } } }
```

and

```
def copyFromNet[T](net: ?[T], ack: ![Unit], out: ![T]) =
    proc { repeat { out!(net?()); ack!(()) } }
```

The effect of using the extended rendezvous in copyToNet is to synchronize the termination of a write to in with the reception of the acknowledgement from the network that the value written has been transmitted to out.

At the producer end of the connection, we set up a bidirectional network connection that transmits data and receives acknowledgements. Then we connect the producer to the network via the adapter:

```
def producer(out: ![T]) = ···
val (toNet, fromNet): (![T], ?[Unit]) = ···
val left = OneOne[T]
( producer(left) || copyToNet(left, toNet, fromNet) )()
```

At the consumer end the dual setup is employed

```
def consumer(in: ?[T]) = ···
val (toNet, fromNet): (![Unit], ?[T]) = ···
val right = OneOne[T]
( copyFromNet(fromNet, toNet, right) || consumer(right) )()
```

4. Closing Ports and Channels (clean termination)

4.1. Introduction

A port may be *closed* at any time, *including after it has been closed*. The trait InPort has method

```
closeIn: Unit
```

whose invocation embodies a promise *on the part of its invoking thread* never again to read from that port. Once it has been invoked, the method $canInput$ will always yield false for that port. Similarly, the trait OutPort has method

```
closeOut: Unit
```

whose invocation embodies a promise *on the part of its invoking thread* never again to write to that port. Once it has been invoked, the method $canOutput$ will always yield false for that port.

It can sometimes be appropriate to *forbid* a channel to be used for further communication, and the Chan trait has an additional method for that purpose, namely:

```
close: Unit
```

The important design questions that must be considered are:

1. What happens to a process that attempts, or is attempting, to communicate through a port whose peer port is closed, or which closes during the attempt?
2. What does it mean to close a *shared* port?

Our design can be summarised concisely; but we must first explain what it means for a channel to be closed:

> *Definition:* A channel is *closed* if it has been closed by enough calls of closeOut at its OutPort or by enough calls of closeIn at its InPort, or by a call of its close method.

Channel type	Enough closeOut	Enough closeIn	Close takes effect on readers
OneOne	1	1	immediately
OneOneBuf(n)	1	1	when drained
N2N(n, $writers$, $readers$)	$\widehat{writers}$	$\widehat{readers}$	immediately
N2NBuf(n, $writers$, $readers$)	$\widehat{writers}$	$\widehat{readers}$	when drained
ManyOne	∞	1	immediately
OneMany	1	∞	immediately
ManyMany	∞	∞	never

The table above summarises what we mean by "enough" – using the notation $\widehat{num}$ to mean ∞ when $num = 0$ and num otherwise. For example an N2N channel specified with $writers > 0$, and $readers > 0$ closes after either $writers$ closeOut calls or $readers$ closeIn calls; but if $writers = 0$ then any number of calls of closeOut can be made without the channel closing, and if $readers = 0$ then any number of calls of closeIn can be made without the channel closing.

The rationale for this is that shared ports are used as "meeting points" for senders and receivers, and that the fact that one sender or receiver has undertaken never to communicate should not necessarily result in the right to do so being denied to others.[13]

The effects of closing ports and/or channels now can be summarised as follows:

- Writer behaviour
  1. An attempt to write to a closed channel raises the exception Closed in the writing thread.

```
def copy[T](in: ?[T], out: ![T]) =
proc {
  repeat { out!(in?()) }     // copying
  out·closeOut; in·closeIn   // close-down
}
```

Program 4. A terminating copy component

2. Closing a channel whose OutPort is waiting in a write raises the exception Closed in the writing thread.

- Reader behaviour

1. An attempt to read from a closed channel raises the exception Closed in the reading thread. If the channel is buffered then this exception is raised only once the last remaining buffered value has been read.
2. Closing a channel whose InPort is waiting in a read raises the exception Closed in the reading thread.

4.2. Termination of networks and components

The Closed exception is one of a family of runtime exceptions, the Stop exceptions, that play a special role in ensuring the clean termination of networks of communicating processes.

The form **repeat** $(expr_{guard})$ $\{\ expr_{body}\ \}$ behaves the same as **while** $(expr_{guard})$ $\{\ expr_{body}\ \}$ except that the raising of a Stop exception during the execution of the $expr_{body}$ causes it to terminate normally. The form **repeat** $\{\ expr_{body}\ \}$ is equivalent to **repeat** (**true**) $\{\ expr_{body}\ \}$

The behaviour of **repeat** simplifies the description of cleanly-terminating iterative components that are destined to be part of a network. For example, consider the copy component of program 4, which has an iterative copying phase followed by a close-down phase. It is evident that the copying phase terminates if the channel connected to the input port is closed before that connected to the output port. Likewise, if the channel connected to the output port is closed before (or within) a write operation that is attempting to copy a recently-read datum. In either case the component moves into its close-down phase, and this results in one of the channels being closed again while the other is closed anew. In nearly all situations this behaviour is satisfactory, but it is worth noticing that it can result in a datum being silently lost (in the implicit buffer between the in?() and the out!) when a network is closed from “downstream”.[14]

In section 1.2 we explained that on termination of all the components of a concurrent process: (a) if they all terminated normally then the concurrent process itself terminates normally; (b) if all components that terminated abnormally terminated with a Stop exception then the concurrent process itself terminates by throwing a Stop exception; (c) otherwise the concurrent process terminates by throwing a ParException.

One consequence of (b) is that it is relatively simple to arrange to reach the closedown phase of an iterated component that does concurrent reads and/or writes. For example, the tee component of program 5 broadcasts data from its input port to all its output ports concurrently: if the input port closes, or if any output port is closed before or during a broadcast, then the component stops broadcasting and closes all its ports.[15]

```
def tee[T](in: ?[T], outs: Seq[![T]]) =
proc
{ var data      = in·nothing   // unspecified initial value
  val broadcast = || for (out←outs) yield proc { out!data }
  repeat { in ?? { d ⇒ { data=d;  broadcast() }}}
  for (out←outs) out·closeOut; in·closeIn
}
```

Program 5. A data broadcasting component

This is because closing in results in a Closed exception being thrown at the next in??; and because closing an output port causes the corresponding out!data to terminate by throwing a Closed, which is propagated in turn by the || when it terminates.[16]

Careful programming of the closedown phases of communicating components is needed in order to assure the clean termination of networks of interconnected processes, and this is facilitated by the Stop-rethrowing behaviour of ||, and the behaviour of **repeat** when its body Stops.

5. Alternation

5.1. Input-guarded events

Alternation constructs enable an input or output action to be performed after being selected from those that are ready on one or more ports. The simplest form of an **alt** consists of a collection of *guarded events*:[17]

alt ($(guard_1$ && $port_1)$ =?=> { bv_1 => cmd_1 }
| ...
| $(guard_n$ && $port_n)$ =?=> { bv_n => cmd_n}
)

An input event of the form $(guard\&\&port)$ =?=> { bv => cmd }

- is said to be *enabled*, if $port$ is open and $guard$ evaluates to true
- is said to be *ready* if $port$ is ready to read
- is *fired* by reading $port$, binding the value read to bv and then executing cmd.

The execution of an **alt** proceeds in principle[18] in phases as follows:

1. All the event guards are evaluated, and then
2. The current thread waits until (at least one) enabled event is ready, and then
3. One of the ready events is chosen and fired.

If no events are enabled after phase 1, or if all the channels associated with the ports close while waiting in phase 2, then the Abort exception (which is also a form of Stop exception) is raised.

If evs is a collection of guarded events, then **serve**(evs) executes these phases repeatedly (until a Stop exception is thrown), but the choices made in phase 3 are made in such a way that if the same group of guards turn out to be ready during successive executions, they will be fired in turn.

For example, the method tagger below constructs a tagging multiplexer that ensures that neither of its input channels gets too far ahead of the other. The tagger terminates cleanly when its output port is closed, or if both its input channels have been closed.

```
def tagger[T](l: ?[T], r: ?[T], out: ![(Int, T)]) =
proc
{ var diff = 0
  serve ( (diff < 5  && l) =?=> { x => out!(0, x); diff+=1 }
        | (diff > -5 && r) =?=> { x => out!(1, x); diff-=1 }
        )
  repeat { out!(0, l?()) }
  repeat { out!(1, r?()) }
  l.closeIn; r.closeIn; out.closeOut
}
```

Notice that the **serve** will also terminate if the “wrong” channel closes (for example l if $diff < 5$); but following such termination at most one of the subsequent **repeat**s (the “right” one) can perform a successful read and write.

A **prialt** is formed in the same way as an **alt**, and is executed in nearly the same way, but the choice of which among several ready guards to fire always favours the earliest in the sequence. A **priserve** repeats a **prialt**.

5.2. Output-guarded events

In late 2008 the *outport guard* notation was added to CSO. Its simplest form is exemplified in the following implementation of a merging component that buffers no more than 20 inputs tagged with sequence numbers, and outputs them when there is demand from `out`. The serve loop will terminate when none of the guarded events can occur – after which the three ports are closed.

```
def taggedMerge[T](l: ?[T], r: ?[T], out: ![(Int, T)]) =
proc
{ var seqn = 0 // sequence number
  var nbuf = 0 // number buffered
  val q       = scala.collection.mutable.Queue[(Int, Int, T)]
  serve ( (nbuf<20 && l)   =?=> { x => q.enqueue((seqn+=1, 0, x)); nbuf+=1; }
        | (nbuf<20 && r)   =?=> { x => q.enqueue((seqn+=1, 1, x)); nbuf+=1; }
        | (nbuf>0  && out) =!=> { nbuf-=1; q.dequeue }
        )
  l.closeIn; r.closeIn; out.closeOut
}
```

The most general form of output-guarded event is

$$(guard \&\& port) \ \ =\!!\!=> \ \ \{ \text{expression} \} ==> \{ \, cmd \, \}$$

It is *ready* if its guard is true and $port$ is ready to be written. It is *fired* by evaluating $expression$ (which can be a sequential composition) and writing its value (v, say) to $port$.[19] When it has been written cmd (known as the *epilogue*) is executed.[20]

If there is no need for an epilogue, the event can be written:

$$(guard\&\&port) \quad =!=> \quad \{\text{ expression }\}$$

The final guard of the `taggedMerge` serve loop could have been written with an epilogue, by doing the buffer-size accounting after the dequeued datum has been transmitted.

```
| (nbuf>0  && out) =!⇒ { q·dequeue } ⟹ { nbuf−=1; }
```

5.3. Collections of guards

Alternations can be composed of collections of guards, as illustrated by the fair multiplexer defined below.[21]

```
def fairPlex[T](ins: Seq[?[T]], out: ![T]) =
    proc { serve (| (for (in ← ins) yield in =?⇒ { t ⇒ out!t })) }
```

They can also be composed by combining collections and single guards. For example, the following is an extract from a multiplexer that can be dynamically set to favour a specific range of its input ports. It gives priority to its range-setting channels.

```
def primux[T](MIN: ?[Int], MAX: ?[Int], ins: Seq[?[T]], out: ![T]) =
proc
{ var min = 0
  var max = ins·length − 1
  priserve ( MIN =?⇒ { n ⇒ min = n }
           | MAX =?⇒ { n ⇒ max = n }
           | | (for (i ← 0 until ins·length) yield
                  (max>=i && i>=min && ins(i)) =?⇒ { t ⇒ out!t } )
           )
}
```

5.4. Timed Alternation

An alternation may be qualified with a deadline and code to be executed in case of a timeout.[22] We illustrate this feature with an extended example that defines the transmitter and receiver ends of an inter-JVM buffer that piggybacks "heartbeat" confirmation to the receiving end that the transmitting end is still alive.

First we define a Scala type Message whose values are of one of the forms Ping or Data(v).

```
trait Message
case object Ping              extends Message {}
case class  Data[T](data: T)  extends Message {}
```

The transmitter end repeatedly forwards data received from in to out, but intercalates Ping messages whenever it has not received anything for pulse nanoseconds.[23]

```
def transmitter[T](pulse: Nanoseconds, in: ?[T], out: ![Message]) =
  proc
  { serve (in =?⇒ {x⇒out!Data(x)} | after(pulse) ⟹ { out!Ping }) }
```

The receiver end (whose deadline should be somewhat larger than the transmitter's pulse) repeatedly reads from in, discarding Ping messages and forwarding ordinary data to out. If (in each iteration) a message has not been received before the current deadline, the receiver backs off a little more, but eventually a message is sent to the fail channel.

```
def receiver[T](pulse: Nanoseconds, in: ?[Message], out: ![T], fail: ![Unit]) =
proc
{ var backoff = 10
  serve( in =?⇒ { case Ping      ⇒ backoff = (backoff+1) % 10
                  case Data(d:T) ⇒ out!d; backoff = (backoff+1) % 10}
       | after(pulse+pulse/backoff) ⟹
                { if (backoff==1) fail!() else backoff −=1 }
       )
}
```

Though timeout is cheap and safe to implement, the technique used above may not be suitable for use in components where there is a need for more subtle interplay between timing and channel input. But such components can always be constructed (and in a way that may be more familiar to occam programmers) by using periodic timers, such as the simple one shown in program 7.

For example, program 6 shows the definition of an alternative transmitter component that "pings" if the periodic timer ticks twice without an intervening input becoming available from in, and "pongs" every two seconds regardless of what else happens.

```
def transmitter2[T](pulse: Nanoseconds, in: ?[T], out: ![Message]) =
proc
{ val tick  = periodicTimer(pulse)
  val tock  = periodicTimer(2∗Sec)
  var ticks = 0
  priserve ( tock =?⇒ { case () ⇒ out!Pong }
           | in   =?⇒ { case t  ⇒ out!Data(t); ticks = 0 }
           | tick =?⇒ { case () ⇒ ticks+=1; if (ticks>1) out!Ping }
           )
  tick·close
  tock·close
}
```

Program 6. A conventionally-programmed transmitter

In the periodic timer of program 7 the **fork** method of a process is used to start a new thread that runs concurrently with the current thread and periodically writes to the channel whose input port represents the timer. Closing the input port terminates the **repeat** the next time the interval expires, and thereby terminates the thread.

```
def periodicTimer(interval: Nanoseconds) : ?[Unit] =
{ val chan = OneOne[Unit]
  proc { repeat { sleep(interval); chan!() } } · fork
  return chan
}
```

Program 7. A simple periodic timer

5.5. *Restrictions on alternation*

For reasons of efficiency and to keep implementations simple *at most one port* of a channel may participate in an alternation construct at any one time.[24]

6. Port Type Variance

As we have seen, port types are parameterized by the types of value that are expected to be read from (written to) them. In contrast to Java, in which all parameterized type constructors are covariant in their parameter types, Scala lets us specify the variance of the port type constructors precisely. Below we argue that the InPort constructor should be covariant in its type parameter, and the OutPort constructor contravariant in its type parameter. In other words:

1. If T' is a subtype of T, then a $?[T']$ will suffice in a context that requires a $?[T]$; but not *vice-versa*.
2. If T' is a subtype of T, then a $![T]$ will suffice in a context that requires a $![T']$; but not *vice-versa*.

Our argument is, in effect, by contradiction. To take a concrete example, suppose that we have an interface Printer which has subtype BonjourPrinter that has an additional method, bonjour.

Suppose also that we have process generators:

```
def printServer   (printers: ![Printer])  : PROC = ···
def bonjourClient(printers: ?[BonjourPrinter]) : PROC = ···
```

Then under the uniformly covariant regime of Java the following program would be type valid, but it would be unsound:

```
val connector = new OneOne[BonjourPrinter]
(printServer(connector) || bonjourClient(connector))()
```

The problem is that the server could legitimately write a non-bonjour printer that would be of little use to a client that expects to read and use bonjour printers. This would, of course, be trapped as a runtime error by the JVM, but it is, surely, bad engineering practice to rely on this lifeboat if we can avoid launching a doomed ship in the first place![25] And we can: for under CSO's contravariant typing of outports, the type of connector is no longer a subtype of ![Printer], and the expression printServer(connector) would, therefore, be ill-typed.

Discussions of the variance of type constructors are easier to understand if we think of the subtype relation as capturing "satisfies all the assumptions we can make about". In what follows we write $T' \geq T$ to mean "a value of type T' satisfies all the assumptions we can make about a value of type T", or alternatively, "a T' has all the methods of a T."

It is a general principle that if a variable var has type T, then it is acceptable to associate var (by binding or assignment) with a value whose type is $\geq T$, for any *uses* of var in its scope can require no more of it than is required of a T, and any $T' \geq T$ provides this.

We rationalize the variances we have assigned to these constructors as follows:

1. If $T' \geq T$ then $?[T'] \;\geq\; ?[T]$
   Rationale: a process that reads from an input port associated with a variable declared by $var\ :\ ?[T]$ expects to receive objects of type $\geq T$. Thus any port of type $?[T']$ (where $T' \geq T$) can be associated with var.
2. If $T' \geq T$ then $![T] \;\geq\; ![T']$
   Rationale: a process that writes to an output port associated with a variable declared by $var\ :\ ![T]$ must send objects of a type that is $\geq T$. Thus any port of type $![T']$ (where $T \geq T'$) can be associated with var.

7. Prospects

We remain committed to the challenge of developing Scala+CSO both as a pedagogical tool and in the implementation of realistic and efficient programs. Several small-scale and a few medium-scale case studies on networked multicore machines have given us some confidence that our implementation is sound, though we have neither proofs of this nor a body of successful (*i.e.* non-failed) model checks. The techniques pioneered by Welch and Martin in [10] show the way this could be done.

In the first version of this paper, we wrote:

> "The open nature of the Scala compiler permits, at least in principle, a variety of compile-time checks on a range of design rules to be enforced. It remains to be seen whether there are any combinations of expressively useful Scala sublanguage and "CSO design rule" that are worth taking the trouble to enforce. We have started our search with an open mind but in some trepidation that the plethora of possibilities for aliasing might render it fruitless – save as an exercise in theory."

We regret to report that the very rapid evolution of the Scala compiler has made it impractical for us to attempt to use it to enforce design rules, or (more importantly) to provide a practical way of using CSO-specific laws to guide compile-time optimizations. On the other hand, the rapid advance of the functionality of IDEs for Scala now offers the intriguing prospect of incorporating rules of this kind in an IDE.

A few years ago Andrew Bate implemented a very high performance variant of CSO, in which huge numbers of running processes can be multiplexed (as "fibres") across smaller numbers of threads. We would very much like to make Andrew's dialect and the dialect described here compatible at the source-code level, but Andrew's implementation depended on extensive post-processing of the JVM code generated by Scala/Java, and keeping it up to date would require the evolution of the Scala compiler to be tracked: something we don't have the resources to commit to.

8. Distributed Programming with CSO

The prototype CSO library **eieio** (Extensible Interface to External I/O) provides components that can be used to implement distributed and networked programs. It maps external sockets to internal channels and handles the details of serializing and deserialising data.

Acknowledgements

We are grateful to Peter Welch, Gerald Hilderink and their collaborators whose early demonstration of the feasibility of using occam-like concurrency in a Java-like language inspired us in the first place. Also to Martin Odersky and his collaborators in the design and implementation of the Scala language.

Several of our students on the Concurrent and Distributed Programming course at Oxford helped us by testing the present work and by questioning the work that preceded and precipitated it. Itay Neeman drew Scala to our attention and participated in the implementation of a prototype of CSO. Dalizo Zuse and Xie He helped us refine our ideas by building families of Web-servers – using a JCSP-like library and the CSO prototype respectively.

Our colleagues Michael Goldsmith and Gavin Lowe (and more recently Andrew Bate) have been a constant source of technical advice and friendly skepticism; and Quentin Miller joined us in research that led to the design and prototype implementation of ECSP [5,6].

Last, but not least, we are grateful to Carsten Heinz and his collaborators for their powerful and versatile LaTeX `listings` package [15].

Appendix: Thumbnail Scala and the Coding of CSO

In many respects Scala is a conventional object oriented language semantically very similar to Java, though notationally somewhat different.[26] It has a number of features that have led some to describe it as a *hybrid* functional and object-oriented language, notably

- *Case classes* make it easy to represent free datatypes and to program with them.
- *Functions are first-class values*. The type expression T⇒U denotes the type of functions that map values of type T into values of type U. One way of denoting such a function anonymously is { bv ⇒ body } (providing body has type U).[27]

The principal novel features of Scala we used in making CSO notationally palatable were:

- *Syntactic extensibility*: objects may have methods whose names are symbolic operators; and an object with an apply method may be "applied" to an argument as if it were a function.
- *Call by Name*: a Scala function or method may have have one or more parameters of type ⇒ T, in which case they are given "call by name" semantics and the actual parameter expression is evaluated anew whenever the formal parameter name is mentioned.
- *Code blocks*: an expression of the form {...} may appear as the actual parameter corresponding to a formal parameter of type ⇒ T.

The following extracts from the CSO implementation show these features used in the implementation of unguarded repetition and **proc**.

```
// Implementing unguarded repetition
def repeat (cmd: ⇒ Unit) : Unit =
{ var go = true
  while (go)
    try   { cmd }
    catch { case io.threadcso.process.Stopped(_,_) ⇒ go=false }
}

// Definition of proc syntax
def proc (body: ⇒ Unit) : PROC =
    new Process.Simple(() ⇒body).withName(Process.genName)
```

Implementation of the guarded event notation of section 5 is more complex. For example, the formation of an input event from the Scala expression $(guard\&\&port)$=?=> rhs takes place in two stages: first the evaluation of $(guard\&\&port)$ yields an intermediate GuardedInPort object, ev; then the evaluation of ev=?=> rhs yields the InPortEvent that will be a candidate for selection and execution. An unguarded event is constructed as an InPortEvent in a simple step.

Appendix: JCSP Fair Multiplexer

Program 8 shows the JCSP implementation of a fair multiplexer component (taken from [3]) for comparison with the CSO implementation of the component with the same functionality in section 5.3.

```
public final class FairPlex implements CSProcess {
    private final AltingChannelInput[] in;
    private final ChannelOutput        out;
    public FairPlex ( AltingChannelInput[] in, ChannelOutput out)
    { this.in = in; this.out = out; }

    public void run () {
        final Alternative alt = new Alternative (in);
        while (true) {  final int i = alt.fairSelect ();
                        out.write (in[i].read ());
        }
    }
}
```

Program 8. Fair Multiplexer Component using JCSP

Notes

[1] This was derived from an earlier library, written in Generic Java, whose development had been inspired by the appearance of the first public edition of JCSP. The principal differences between that library and the JCSP library were the generically parameterized interfaces, InPort and OutPort akin to what JCSP called "channel ends."

[2] Although Scala interoperates with Java, and we could easily have constructed Scala "wrappers" for the JCSP library and for our own derivative library, we wanted to have a pure Scala implementation both to use as part of our instructional material, and to ensure portability to the .NET platform when the Scala .NET compiler became available.

[3] The (admirably ingenious) Actor library implementation is complicated; its performance appears to scale well only for certain styles of use; and it depends for correct functioning on a global timestamp ([14] p183).

[4] The present pool implementation acquires new worker threads from the underlying JVM when necessary and "retires" threads that have remained dormant in the pool for more than a certain period.

[5] The expression run(p) has exactly the same effect as $p()$. The expression fork(p) runs p in a new thread concurrent with the thread that invoked fork, and returns a *handle* on the running process. The new thread is recycled when the process terminates.

[6] This is because io.threadcso.process.Stopped exceptions signify anticipated failure, whereas other types of exception signify unexpected failure, and must be propagated rather than silently ignored. One useful consequence of the special treatment of io.threadcso.process.Stopped exceptions is explained in section 4: *Closing Ports and Channels*.

[7] Other forms of synchronous channel, mostly now obsolete, are:

- ManyOne[T] – No more than one process at a time may access its input port; processes attempting to access its output port get access in nondeterministic order. The name is a contraction of "From Many possible writer processes to One reader process." The other forms of synchronous channel are named using the same contraction convention.
- OneMany[T] – No more than one process at a time may access its output port; processes attempting to access its input port get access in nondeterministic order.

- ManyMany[T] – Any number of processes may attempt to access either port. Writing processes get access in nondeterministic order, as do reading processes.

[8] The name is a contraction of "From One writer process to One reader process."

[9] See, for example, the component mux2 defined in program 3.

[10] The reduction of formal clutter comes at the cost of forcing readers to refer back to the component signatures to ascertain which ports they actually use. The JCSP designers made the tradeoff in the other direction.

[11] We have used the plain form of read $(mid?())$ and its read-and-evaluate form $(ins(i)?v \Rightarrow mid!(i, v))$ simply to give an example of the latter.

[12] The most general form of extended rendezvous read is in??f where f denotes a function of type T⇒U. The type of in??f is then U.

[13] This is a deliberate choice, designed to keep shared channel semantics simple. More complex channel-like abstractions – such as one in which a non-shared end is informed when all subscribers to the shared end have disappeared – can always be layered on top of it.

[14] *i.e.* from the out direction. On the face of it it looks like this could be avoided by reprogramming the component with a stronger guard to the iteration, *viz* as: **repeat** (out·canOutput) { out!(in ?()) } *but this is not so*, because the out·canOutput test and the out! action are not joined atomically, so the channel associated with the output port could be closed between being polled in the guard and being written to in the body of the loop.

[15] We observe, without much enthusiasm, that there is no particular constraint on the order in which the ports are closed, so they could be closed in parallel by the more complicated:

(|| (**for** (out←outs) **yield** **proc** out·closeOut) || in·closeIn)()

[16] Although it is incidental to the theme of this example, it is worth noticing that we construct the concurrent process broadcast before starting the iteration. While this is not strictly necessary, it provides an improvement in efficiency over: **repeat** { in ? { d ⇒ {|| (**for** (out←outs) **yield** **proc** { out!d })() }}}. This is because the expression: ||(**for** (out←outs) ...) that constructs the concurrent broadcast process is evaluated only once, rather than being evaluated once per broadcast.

[17] Guard expressions must be free of side-effects, and a $(guard)$ that is literally (true) may be omitted.

[18] We say "in principle" because we wish to retain the freedom to use a much more efficient implementation than is described here, namely an adaptation of that described in [13].

[19] An early version of CSO provided an even more general form, in which the epilogue was a function, and the value of the expression was passed to this function once it had been transmitted. This proved incompatible with the Scala type system.

[20] The operator $\Longrightarrow$ that introduces the epilogue is pronounced "and then".

[21] It is perhaps worthwhile comparing this construction with that of the analogous JCSP component shown in program 8 (page 19).

[22] The implementation of this feature employs a nonzero timeout for the wait in phase 2, and is not subject to any potential race conditions.

[23] Nanosecond is now the unit of resolution of CSO time. It's not yet realistic to measure delays or timeouts in small numbers of nanoseconds, so appropriate multipliers, such as microSec, milliSec, Sec, Min, Hour, Day are provided as part of the CSO package.

[24] Gavin Lowe [16] worked with an earlier version of CSO to remove this restriction.

[25] This difficulty is analogous to the well-known difficulty in Java caused by the covariance of the array constructor.

[26] The main distributed Scala implementation translates directly into the JVM; though another compiler translates into the `.net` CLR. The existence of the latter compiler encouraged us to build a pure Scala CSO library rather than simply providing wrappers for the longer-established JCSP library.

[27] In some contexts fuller type information has to be given, as in: { **case** bv: T ⇒ body }. Functions may also be defined by cases over free types; for an example see the match expression within receiver in section 5.4

References

[1] Hilderink G., Broenink J., Vervoort W. and Bakkers A. Communicating Java Threads. In *Proceedings of WoTUG-20: ISBN 90 5199 336 6* (1997) 48–76.

[2] Hilderink G., Bakkers A. and Broenink J. A Distributed Real-Time Java System Based on CSP. In *Proceedings of the Third IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, (2000), 400–407.

[3] Website for Communicating Sequential Processes for Java, (2008) `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`

[4] Welch P. *et. al.* Integrating and Extending JCSP. In *Communicating Process Architectures* (2007) 48–76.

[5] Miller Q. and Sufrin B. Eclectic CSP: a language of concurrent processes. In *Proceedings of 2000 ACM symposium on Applied computing*, (2000) 840–842.

[6] Sufrin B. and Miller Q. Eclectic CSP. *OUCL Technical Report*, (1998) `http://users.comlab.ox.ac.uk/bernard.sufrin/ECSP/ecsp.pdf`

[7] Welch P. *et. al.* Letter to the Editor. *IEEE Computer*, (1997) `http://www.cs.bris.ac.uk/~alan/Java/ieeelet.html`

[8] Muller H. and Walrath K. Threads and Swing. *Sun Developer Network*, (2000) `http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html`

[9] Odersky M. *et. al.* An Overview of the Scala Programming Language. *Technical Report LAMP-REPORT-2006-001, EPFL, 1015 Lausanne, Switzerland* (2006) (also at `http://www.scala-lang.org/`)

[10] Welch P. and Martin J. Formal Analysis of Concurrent Java Systems. In *Proceedings of CPA 2000 (WoTUG-23): ISBN 1 58603 077 9* (2000) 275–301.

[11] Welch P. CSP Networking for Java (JCSP.net) `http://www.cs.kent.ac.uk/projects/ofa/jcsp/jcsp-net-slides-6up.pdf`

[12] Haller P. and Odersky M. Event-based Programming without Inversion of Control. In: Lightfoot D.E. and Szyperski C.A. (Eds) JMLC 2006. LNCS 4228, 4–22. Springer-Verlag, Heidelberg (2006) (also at `http://www.scala-lang.org/`)

[13] Geoff Barrett, Michael Goldsmith, Geraint Jones, and Andrew Kay. The meaning and implementation of PRI ALT in **occam**. In **occam** and the Transputer, research and applications (OUG-9), ed. Charlie Askew, IOS, (September 1988). (also at `ftp://ftp.comlab.ox.ac.uk/pub/Documents/techpapers/Geraint.Jones/OCCAM-1-88.ps.Z`)

[14] Haller P. and Odersky M. Actors that unify Threads and Events. In: Murphy A.L. and Vitek J. (Eds) COORDINATION 2007. LNCS 4467, 171–190. Springer-Verlag, Heidelberg (2006) (also at `http://www.scala-lang.org/`)

[15] Heinz C. The **`listings`** Package. (`http://www.ifi.uio.no/it/latex-links/listings-1.3.pdf`)

[16] Lowe, G. Implementing Generalised Alt. In *Proceedings of Concurrent Process Architectures* (2011) 1–34. (`http://www.cs.ox.ac.uk/gavin.lowe/Papers/alt.pdf`)