Communicating Process Architectures 2008
Susan Stepney, Fiona Polack, Alistair McEwan, Peter Welch, and Wilson Ifill (Eds.)
IOS Press, 2008

Communicating Scala Objects (Revised)

Bernard SUFRIN

Oxford University Computing Laboratory
and
Worcester College,
Oxford OX1 2HB, England

Bernard.Sufrin@comlab.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. Initially inspired by an early release of JCSP, CSO goes beyond JCSP expressively in some respects, including the provision of a unitary extended rendezvous notation and appropriate treatment of subtype variance in channels and ports. Similarities with recent versions of JCSP include the treatment of channel ends (we call them *ports*) as parameterized types. Ports and channels may be transmitted on channels (including inter-JVM channels), provided that an obvious design rule – the *ownership* rule – is obeyed. Significant differences with recent versions of JCSP include a treatment of network termination that is significantly simpler than the "poisoning" approach (perhaps at the cost of reduced programming convenience), and the provision of a family of type-parameterized channel implementations with performance that obviates the need for the special-purpose scalar-typed channel implementations provided by JCSP. On standard benchmarks such as Commstime, CSO communication performance is close to or better than that of JCSP and Scala's Actors library.

This version revises section 5 of the paper published in the conference proceedings.

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 JavaPP, JCSP, and CTJ [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 1.5 library that bore a strong resemblance to the current JCSP 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 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 some familiarity with the conceptual and notational basis of occam and JCSP, but only a little familiarity with Scala.

Readers familiar with JCSP and Scala may be able to get a quick initial impression of the relative notational weights of Scala+CSO and Java+JCSP by inspecting the definitions of FairPlex multiplexer components defined on pages 12 and 20 respectively.

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 values, denoted by one of:

[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 modern JCSP 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.

proc $\{ \; expr \; \}$	A simple process ($expr$ must be a command, *i.e.* have type Unit)
$p_1 \; \| \| \; p_2 \; \| \| \; ... \; \| \| \; p_n$	Parallel composition of n processes (each p_i must have type PROC)
$\| \| \; collection$	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: $p(0) \; || \; p(1) \; || \; ... \; || \; p(n-1)$,

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:

1. p is **proc** $\{ \; expr \; \}$
   - $p()$ causes $\{ \; expr \; \}$ to be evaluated in the current thread.
   - 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}.
2. p is $p_1 \; || \; p_2 \; || \; ... \; || \; p_n$
   - $p()$ causes all the processes $p_1 ... p_n$ to be run concurrently.
   - All but one of the processes is run in a new thread; the remaining process is run in the current thread.
   - The process as a whole terminates only when *all* the component p_i have terminated. But if one or more of the component p_i 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* cso.Stop; in which case a single cso.Stop is thrown.[6]

2. Ports and Channels

2.1. Introduction

Following ECSP [5,6], CSO ports (akin to JCSP channel ends) are generically parameterized, 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 important methods of an ?[T] are its read method

```
? () : T
```

[5]A process also has a fork method that runs it in a new thread concurrent with the thread that invoked its fork method. The new thread is recycled when the process terminates.

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

and its *extended rendezvous* method

```
? [U] (body: T => U) : U
```

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 delivers the data written at its output port to its input port in the order in which the data was 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.

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.
  - OneOne[T] – No more than one process at a time may access its output port or its input port.[7] This is the classic occam-style point to point channel.
  - 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.[8]
  - 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.
- Buf[T](n) – a many-to-many buffer of capacity n.[9]

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.[10]
- 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.

---

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

[8] 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.

[9] We expect that history will soon give way to logic: at that point each form of synchronous channel will be supplemented by an aptly-named form of buffered channel.

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

```
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 = OneOne[T](n) // 2 arrays of n unshared channels
val mid = OneOne[(int, T)]     // an unshared 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 = OneOne[T](n)    // an array of n unshared channels

(  || (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 read is attempted from a channel with an unshared input port before an earlier read has terminated, then an illegal state exception is thrown.
- ∗ 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 still 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

```
def mux1[T] (ins: Seq[?[T]], out: ![(int, T)]) : PROC =
{ val mid = ManyOne[(int, T)]
  (  proc { while(true) { out!(mid?) } }
  || (for (i<-0 until ins.length) yield proc { mid!(i, ins(i)?) })
  )
}

def mux2[T] (ins: Seq[?[T]], out: SharedOutPort[(int, T)]) : PROC =
    || (for (i<-0 until ins.length) yield proc { 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

the connecting channels as parameters, and that the components take the required *view*s of them. This means we needn't name the ports explicitly, and significantly reduces the degree of *formal clutter* in the network description.[11]

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.

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 a ManyOne[(**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 it we must assume that it 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 terminates once the data is transferred between the writer and the reader process, an *extended rendezvous read* permits a computation on the transferred data to take place *in the reader process*, and 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 }
```

[11]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.

[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.

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 = Chan[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 = Chan[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 is also used to good effect in the implementation of synchronized inter-JVM or cross-network connections, where it keeps 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) )()
```

In reality the CSO networking components deliver their functionality at a higher level of abstraction than this, namely bidirectional client/server connections, and the synchronous implementations piggy-back acknowledgements to client requests on top of server responses.

4. Closing Ports and Channels

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.

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.

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 at a *non-shared* OutPort by invoking its closeout method, or if it has been closed at a *non-shared* InPort by invoking its closein method, or if it has been closed by invoking its close method.[13]

This means that closing a shared *port* has no effect. 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 result in the right to do so being denied to others.[14]

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.
  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.
  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

$$\textbf{repeat}\ (expr_{guard})\ \{\ expr_{body}\ \}$$

behaves in exactly the same way as

$$\textbf{while}\ (expr_{guard})\ \{\ expr_{body}\ \}$$

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 humble 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

[13] In the case of buffered (non-synchronized) channels, the effect of invoking close is immediate at the InPort, but is delayed at the OutPort until any buffered data has been consumed.

[14] 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.

```
def copy[T](in: ?[T], out: ![T]) =
proc {
  repeat { out!(in?) }                              // copying
  (proc { out.closeout } || proc { in.closein })()  // close-down
}
```

Program 4. A terminating copy component

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".[15]

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 below 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.

```
def tee[T](in: ?[T], outs: Seq[![T]]) =
proc
{ var data      = _  // unspecified initial value
  val broadcast = || for (out<-outs) yield proc { out!data }
  repeat { in ? { d => { data=d; broadcast() }}}
  (|| (for (out<-outs) yield proc { out.closeout }) || in.closein)()
}
```

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.

---

[15] *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.open) { out!(in?) } *but this is not so*, because the out.open 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.

[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.

5. Input Alternation

5.1. Introduction

The simplest form of an **alt** consists of a collection of *guarded events*:[17]

```
alt ( port₁ (guard₁) --> { cmd₁ }
    | ...
    | portₙ (guardₙ) --> { cmdₙ }
    )
```

An event of the form $port\ (guard)$ `-->` $\{\ cmd\ \}$

- 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 executing its cmd (which *must* read $port$)

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 ( l   (!r.open || diff < 5 ) --> { out!(0, l?); diff+=1 }
        | r   (!l.open || diff > -5) --> { out!(1, r?); diff-=1 }
        )
  ( proc {l.closein} || proc {r.closein} || proc {out.closeout} )()
}
```

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. Sugared notation for guarded events

CSO has a supplementary notation for guarded events that ensures that there is no possiblity of the programmer forgetting to read from the port. The notation

$$port\ (guard)\ \ \texttt{==>}\ \ function$$

[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].

is syntactic sugar for the notation:

$$port\ (guard)\ \ \texttt{-->}\ \ \{\ function(port?)\ \}.$$

The business end of the tagger described above is expressed as follows in this notation

```
serve ( l   (!r.open || diff < 5 ) ==> { t => out!(0, t); diff+=1 }
      | r   (!l.open || diff > -5) ==> { t => out!(1, t); diff-=1 }
      )
```

5.3. Collections of guards

Alternations can be composed of collections of guards, as illustrated by the fair multiplexer defined below.[19]

```
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 than 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 ==> { min = MIN? }
           | MAX ==> { max = MAX? }
           | | (for (i <- 0 until ins.length) yield
                  ins(i) (max>=i && i>=min) ==> { 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.[20] 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 milliseconds.

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

[19] It is perhaps worthwhile comparing this construction with that of the analogous JCSP component shown in program 11 (page 20).

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

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: long, 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 and straighforward one shown in program 6.

For example, program 5 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: long, in: ?[T], out: ![Message]) =
proc
{ val tick  = periodicTimer(pulse)
  val tock  = periodicTimer(2000)
  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 5. A conventionally-programmed transmitter

In the periodic timer of program 6 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: long) : ?[Unit] =
{ val chan = OneOne[Unit]
  proc { repeat { sleep(interval); chan!() } } . fork
  return chan
}
```

Program 6. A simple periodic timer

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, as it were, 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) || printClient(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![21] 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.

7. Bidirectional Connections

In order to permit client-server forms of interaction to be described conveniently CSO defines two additional interface traits:

```
trait Connection.Client[Request,Reply] extends OutPort[Reply]
                                       with    InPort[Request] { ... }

trait Connection.Server[Request,Reply] extends OutPort[Request]
                                       with    InPort[Reply]   { ... }
```

Thus a Server interface is something to which requests are written and from which replies are read, while a Client interface is something from which requests are read and to which replies are written.

A Connection[Request,Reply] has a client interface and a server interface:

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

```
trait Connection[Request, Reply]
{ def client : Connection.Client[Request,Reply]
  def server : Connection.Server[Request,Reply]
}
```

The implicit contract of a connection implementation is that requests written to its server interface by the code of a client should eventually be readable by the code of the corresponding server in the order in which they were written; likewise responses written to its client interface by the code of a server should eventually be readable by the code of the corresponding client in the order they were written. Different connection implementations implement "eventually" in different ways. The simplest of these is a

```
Connection.OneOne[Request,Reply]
```

which connects both directions synchronously.

It is worth noticing that both Client and Server interfaces can be viewed as both an InPort and an OutPort. This lends an air of verisimilitude to the wrong idea that "a connection is a bidirectional channel", but nevertheless contributes to the lack of formal clutter in the programming of clients and servers.

For example, program 7 shows a process farmer component that acquires requests from its in port, and farms them out to servers from which it eventually forwards replies to its out port. This implementation is a little inefficient because we enable *all* the server guards when *any* server is busy.

```
def farmer[Req,Rep]( in:       ?[Req]
                   , out:      ![Rep]
                   , servers: Seq[Server[Req,Rep]]
                   ) =
proc
{ var busy = 0                      // number of busy servers
  val free = new Queue[![Req]]     // queue of free server connections
  free ++= servers                  // initially all are free
  // INVARIANT: busy+free.length=servers.length
  serve(| (for (server <- servers) yield
              server (busy>0) ==>
              { res => out ! res
                       free += server
                       busy = busy-1
              }
          )
        | in (free.length>0) ==>
        { req => { val server = free.dequeue
                   busy = busy+1
                   server ! req
                 }
        }
       )
}
```

Program 7. A Process Farmer

8. Performance

The Commstime benchmark has been used as a measure of communication and thread context-swap efficiency for a number of implementations of occam and occam-like lan-

```
val a,b,c,d       = OneOne[int]
val Prefix        = proc { a!0; repeat { a!(b?) }}
val Succ          = proc { repeat { b!(c?+1) } }
val SeqDelta      = proc { repeat { val n=a?; c!n; d!n } }
val SeqCommstime  = (Prefix || SeqDelta || Succ || Consumer)

val ParDelta      = proc { var n   = a?;
                           val out = proc{c!n} || proc{d!n}
                           repeat { out(); n=a? }
                         }
val ParCommstime  = (Prefix || ParDelta || Succ || Consumer)
```

Program 8. Parallel and Sequential variants of the Commstime network defined with CSO

```
type Node   = OutputChannel[int]
val  Succ   : Node =
       actor { loop { receive { case n:int => Prefix!(1+n)}}}
val  Prefix : Node =
       actor { Delta!(0); loop { receive { case n:int => Delta!n}}}
val  Delta  : Node =
       actor { loop { receive { case n:int => {Succ!n; Consume!n}}}}
```

Program 9. The Commstime network defined with Actors

guages and library packages. Its core consists of a cyclic network of three processes around which an integer value, initially zero, is circulated. On each cycle the integer is replaced by its successor, and output to a fourth process, Consumer, that reads integers in batches of ten thousand, and records the time per cycle averaged over each batch.

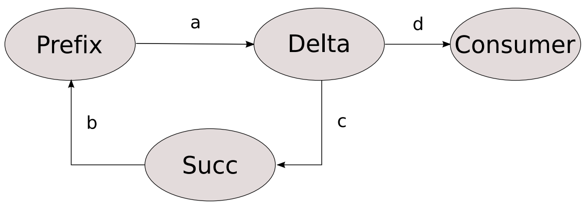

Figure 1. The Commstime network

The network is shown diagrammatically in figure 1. Its core components are defined (in two variants) with CSO in program 8, and with Actors in program 9. The SeqCommstime variant writes to Succ and Consumer sequentially. The ParCommstime variant writes to Consumer and Succ concurrently, thereby providing a useful measure of the overhead of starting the additional thread per cycle needed to implement ParDelta.

In table 1 we present the results of running the benchmark for the current releases of Scala Actors, CSO and JCSP using the latest available Sun JVM on each of a range of host types. The JCSP code we used is a direct analogue of the CSO code: it uses the specialized integer channels provided by the JCSP library. Each entry shows the range of average times per cycle over 10 runs of 10k cycles each.

Host	JVM	Actors	CSO Seq	JCSP Seq	CSO Par	JCSP Par
4 × 2.66GHz Xeon, OS/X 10.4	1.5	28-32	31-34	44-45	59-66	54-56
2 × 2.4GHz Athlon 64X2 Linux	1.6	25-32	26-39	32-41	24-46	27-46
1 × 1.83GHz Core Duo, OS/X	1.5	62-71	64-66	66-69	90-94	80-89
1 × 1.4GHz Centrino, Linux	1.6	42-46	30-31	28-32	49-58	36-40

Table 1. Commstime performance of Actors, CSO and JCSP (Range of Avg. μs per cycle)

```
val a,b,c,d        = Buf[int](4)
val BuffPrefix     = proc { a!0; a!0; a!0; a!0; repeat { a!(b?) } }
...
val BuffCommstime = (BuffPrefix || SeqDelta || Succ || Consumer)
```

Program 10. Buffered Commstime for Actors *v.* CSO benchmark

8.1. Performance Analysis: CSO v. JCSP

It is worth noting that communication performance of CSO is sufficiently close to that of JCSP that there can be no substantial performance disadvantage to using completely generic component definitions.

It is also worth noting that process startup overhead of CSO is somewhat higher than that of JCSP. This may well reflect the fact that the JCSP Parallel construct caches the threads used in its first execution, whereas the analogous CSO construct re-acquires threads from its pool on every execution of the parallel construct.

8.2. Performance Analysis: Actors v. Buffered CSO

At first sight it appears that performance of the Actors code is better than that of CSO and JCSP: but this probably reflects the fact that Actors communications are buffered, and communication does not force a context switch. So in order to make a like-for-like comparison of the relative communication efficiency of the Actors and CSO libraries we ran a modified benchmark in which the CSO channels are 4-buffered, and 4 zeros are injected into the network by Prefix to start off each batch. The CSO modifications were to the channel declarations and to Prefix – as shown in program 10; the Actors version of Prefix was modified analogously. The results of running the modified benchmark are provided in table 2.

Host	JVM	Actors	CSO
4 × 2.66 GHz Xeon, OS/X 10.4	1.5	10-13	10-14
2 × 2.4 GHz Athlon 64X2 Linux	1.6	16-27	4-11
1 × 1.83 GHz Core Duo, OS/X 10.4	1.5	27-32	14-21
1 × 1.4 Ghz Centrino, Linux	1.6	42-45	17-19

Table 2. Buffered Commstime performance of Actors and CSO (Range of Avg. μs per cycle)

Space limitations preclude our presenting the detailed results of the further experiments we conducted, but we noted that even when using an *event-based* variant of the Actors code, the performance of the modified CSO code remains better than that of the Actors code, and becomes increasingly better as the number of initially injected zeros increases. The account of the Actors design and implementation given in [12,14] suggests to us that this may be a consequence of the fact that the network is cyclic.[22]

9. Prospects

We remain committed to the challenge of developing Scala+CSO both as a pedagogical tool and in the implementation of realistic 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.

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.

Finally, we continue to be inspired and challenged by the work of the JCSP team. We hope that new communication and synchronization components similar to some of those they describe in [4] and a networking framework such as that described in [11] will soon find their way into CSO; and if this happens then the credit will be nearly all theirs.

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 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].

The anonymous referees' comments on the first draft of this paper were both challenging and constructive; we are grateful for their help.

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

[22]This result reinforces our feeling that the only solution of the scaleability problem addressed by the Actors library is a reduction in the cost of "principled" threading. We are convinced that this reduction could be achieved by (re-)introducing a form of lighter-weight (green) threads, and by providing OS-kernel/JVM collaboration for processor-scheduling.

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.[23] 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).[24]

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**.

```
// From the CSO module: implementing unguarded repetition
def repeat (cmd: => Unit) : Unit =
{ var go = true;
  while (go) try { cmd } catch { case ox.cso.Stop(_,_) => go=false }
}

// From the CSO module: definition of proc syntax
def proc (body: => Unit) : PROC = new Process (null) (()=>body)
```

Implementation of the guarded event notation of section 5 is more complex. The formation of an InPort.Event (the main form of Alt.Event) from the Scala expression $port(guard) \Longrightarrow fun$ takes place in two stages: first the evaluation of $port(guard)$ yields an intermediate InPort.GuardedEvent object, ev; then the evaluation of $ev \Longrightarrow fun$ yields the required event. An unguarded event is constructed in a simple step.[25]

```
// From def'n of InPort[T]: a regular guarded event
def apply (guard: =>boolean) =
    new GuardedInPortEvent(this, ()=>this.open && guard)

// From def'n of GuardedInPortEvent[T](port:InPort[T], guard: ()=>boolean)
def ==> (fun: T=>Unit) = new InPortEvent[T](port, ()=>fun(port?), guard)

// From def'n of InPort[T]: implementing a true-guarded event
def ==> (fun: T=>Unit) =
    new InPortEvent[T](this, ()=>fun(this?), ()=>this.open)
```

[23] 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.

[24] 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

[25] Notice that openness of the port becomes an implicit part of the guard.

Appendix: JCSP Fair Multiplexer

Program 11 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 11. Fair Multiplexer Component using JCSP

Appendix: Availability

Release announcements are made at:

`http://users.comlab.ox.ac.uk/bernard.sufrin/cso.html.`

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`)

`(Id: cpa2008-cso.tex 557 2008-08-21 15:29:48Z sufrin )`