# Implementing Generalised Alt

*A Case Study in Validated Design using CSP*

Gavin LOWE

*Department of Computer Science, University of Oxford,*
*Wolfson Building, Parks Road, Oxford, OX1 3QD, UK;*
*e-mail* gavin.lowe@cs.ox.ac.uk

**Abstract.** In this paper we describe the design and implementation of a generalised alt operator for the Communicating Scala Objects library. The alt operator provides a choice between communications on different channels. Our generalisation removes previous restrictions on the use of alts that prevented both ends of a channel from being used in an alt. The cost of the generalisation is a much more difficult implementation, but one that still gives very acceptable performance. In order to support the design, and greatly increase our confidence in its correctness, we build CSP models corresponding to our design, and use the FDR model checker to analyse them.

**Keywords.** Communicating Scala Objects, alt, CSP, FDR.

## Introduction

Communicating Scala Objects (CSO) [14] is a library of CSP-like communication primitives for the Scala programming language [12]. As a simple example, consider the following code:

```
val c = OneOne[String];
def P = proc{ c!"Hello world!"; }
def Q = proc{ println (c?); }
(P || Q)();
```

The first line defines a (synchronous) channel c that can communicate Strings (intended to be used by one sender and one receiver—hence the name OneOne; CSO also has channels whose ends can be shared); the second and third lines define processes (more accurately, threads) that, respectively, send and receive a value over the channel; the final line combines the processes in parallel, and runs them.

CSO —inspired by occam [9]— includes a construct, alt, to provide a choice between communicating on different channels. In this paper we describe the design and implementation of a generalisation of the alt operator. We begin by describing the syntax and (informal) semantics of the operator in more detail.

As an initial example, the code

```
alt ( c --> { println("c: "+(c?)); } | d --> { println("d: "+(d?)); } )
```

tests whether the environment is willing to send this process a value on either c or d, and if so fires an appropriate branch. Note that the body of each branch is responsible for performing the actual input: the alt just performs the selection, based on the communications offered by the environment. Channels may be closed, preventing further communication; each alt considers only its open channels.

Each branch of an alt may have a boolean guard. For example, in the alt:

```
alt ( (n >= 0 &&& c) −−> { println("c: "+(c?)); } | d −−> { println("d: "+(d?)); } )
```

the communication on c is enabled only if n >= 0.

An alt may also have a timeout branch, for example:

```
alt ( c −−> { println("c: "+(c?)); } | after(500) −−> { println("timeout"); } )
```

If no communication has taken place on a different branch within the indicated time (in milliseconds) then the alt times out and selects the timeout branch. Finally, an alt may have an orelse branch, for example:

```
alt ( (n >= 0 &&& c) −−> { println("c: "+(c?)); } | orelse −−> { println("orelse"); } )
```

If every other branch is disabled —that is, the guard is false or the channel is closed— then the orelse branch is selected. (By contrast, if there is no orelse branch and all the other branches are disabled, then the alt throws an Abort exception.) Each alt may have at most one timeout or orelse branch.

In the original version of CSO —as in occam— alts could perform selections only between input ports (the receiving ends of channels, known as InPorts). Later this was extended to include output ports (the sending ends of channels, known as OutPorts), for example:

```
alt ( in −?−> { println("in: "+(in?)); } | out −!−> { out!2011; } )
```

The different arrows −?−> and −!−> show whether the InPort or OutPort of the channel is to be used; the simple arrow −−> can be considered syntactic sugar for −?−>.

Being able to combine inputs and outputs in the same alt can be useful in a number of circumstances. The following example comes from the bag-of-tasks pattern [4]. A server process maintains a collection of tasks (in this case, in a stack) to be passed to worker processes on channel toWorker. Workers can return (sub-)tasks to the server on channel fromWorker. In addition, a worker can indicate that it has completed its last task on channel done; the server maintains a count, busyWorkers, of the workers who are currently busy. The main loop of the server can be defined as follows:

```
serve(
  (!stack.isEmpty &&& toWorker) −!−> { toWorker!(stack.pop) ; busyWorkers += 1; }
  | (busyWorkers>0 &&& fromWorker) −?−> { stack.push(fromWorker?); }
  | (busyWorkers>0 &&& done) −?−> { done? ; busyWorkers −= 1 }
)
```

The construct serve represents an alt that is repeatedly executed until all its branches are disabled — in this case, assuming no channels are closed, when the stack is empty and busyWorkers = 0. In the above example, it is possible to replace the output branch (the first branch) by one where the server receives a request from a worker (on channel req) before sending the task

```
  (!stack.isEmpty &&& req) −?−> { req?; toWorker!(stack.pop) ; busyWorkers += 1; }
```

However, such a solution adds complexity for the programmer; a good API should hide such complexities. Further, such a solution is not always possible.

However, the existing implementation of alt has the following restriction [15]:

*A channel's input and output ports may not both simultaneously participate in alts.*

This restriction makes the implementation of alts considerably easier. It means that at least one end of each communication will be *unconditional*, i.e. that offer to communicate will not be withdrawn once it is made.

However, the restriction can prove inconvenient in practice, preventing many natural uses of alts. For example, consider a ring topology, where each node may pass data to its clockwise neighbour or receive data from its anticlockwise neighbour; this pattern can be used to adapt the above bag-of-tasks to a distributed-bag-of-tasks as follows, where give and get are aliases for the channels connecting this node to its neighbours:[1]

```
serve(
  (!stack.isEmpty &&& toWorker) −!−> { toWorker!(stack.pop); workerBusy = true; }
  | (workerBusy &&& fromWorker) −?−> { stack.push(fromWorker?); }
  | (workerBusy &&& done) −?−> { done?; workerBusy = false; }
  | (!stack.isEmpty &&& give) −!−> { give!(stack.pop); }
  | ((!workerBusy && stack.isEmpty) &&& get) −?−> { stack.push(get?); }
)
```

However, now the InPorts and OutPorts of channels connecting nodes are both participating in alts, contrary to the above restriction.

One goal of this paper is to present a design and implementation for a generalised alt operator, that overcomes the above restriction.

McEwan [11] presents a formal model for a solution to this problem, based on a two-phase commit protocol, with the help of a centralised controller.

Welch et al. [17,18] implement a generalised alt, within the JCSP library. The implementation makes use of a single (system-wide) *Oracle* server process, which arbitrates in all alts that include an output branch or a barrier branch (which allows multi-way synchronisation); alts that use only input branches can be implemented without the Oracle. This is a pragmatic solution, but has the disadvantage of the Oracle potentially being a bottleneck.

Brown [1] adopted the same approach within the initial version of the CHP library. However, later versions of CHP built upon Software Transactional Memory [6] and so was decentralised in that alts offering to communicate on disjoint channels did not need to interact; see [3,2].

Our aim in this paper is to investigate an alternative, more scalable design. In particular, we are aiming for a design with no central controller, and that does not employ additional channels internally.

However, coming up with a correct design is far from easy. Our development strategy, described in later sections, was to build CSP [13] models of putative designs, and then to analyse them using FDR [5]. In most cases, our putative designs turned out to be incorrect: FDR revealed subtle interactions between the components that led to incorrect behaviour. Debugging CSP models using FDR is very much easier than debugging code by testing for a number of reasons:

- FDR does exhaustive state space exploration, whereas execution of code explores the state space nondeterministically, and so may not detect errors;
- The counterexamples returned by FDR are of minimal length (typically about 20 in this work), whereas counterexamples found by testing are likely to be much longer (maybe a million times longer, based on our experience of a couple of bugs that did crop up in the code);
- CSP models are more abstract and so easier to understand than code.

[1]This design ignores the problem of distributed termination; a suitable distributed termination protocol can be layered on top of this structure.

A second goal of this paper, then, is to illustrate the use of CSP in such a development.

One factor that added to the difficulty was that we were aiming for an implementation using the concurrency primitives provided by the Scala programming language, namely monitors. A third goal of this paper is an investigation of the relationship between abstract CSP processes and implementations using monitors: what CSP processes can be implemented using monitors, and what design patterns can we use?

One may use formal analysis techniques with various degrees of rigour. Our philosophy in this work has been pragmatic rather than fully rigorous. Alts and channels are *components*, and do not seem to have abstract specifications against which the designs can be verified. The best we can do is analyse systems built from the designs, and check that they act as expected. We have analysed a few such systems; this gives us a lot of confidence that other systems would be correct — but does not give us an absolute guarantee of that. Further, the translation from the CSP models to Scala code has been done informally, because, in our opinion, it is fairly obvious.

The rest of this paper is structured as follows. Below we present a brief overview of CSP and of monitors. In Section 1 we present an initial attempt at a design; this design will be incorrect, but presenting it will help to illustrate some of the ideas, and indicate some of the difficulties. In Section 2 we present a correct design, but omitting timeouts and closing of channels; we validate the design using FDR. That design, however, does not seem amenable to direct implementation using a monitor. Hence, in Section 3, we refine the design, implementing each alt as the parallel composition of two processes, each of which could be implemented as a monitor. In Section 4 we extend the design, to include timeouts and the closing of channels; this development requires the addition of a third component to each alt. In Section 5 we describe the implementation: each of the three processes in the CSP model of the alt can be implemented using a monitor. We sum up in Section 6.

## *CSP*

In this section we give a brief overview of the syntax for the fragment of CSP that we will be using in this paper. We then review the relevant aspects of CSP semantics, and the use of the model checker FDR in verification. For more details, see [7,13].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. Processes communicate via atomic events. Events often involve passing values over channels; for example, the event $\mathsf{c.3}$ represents the value $\mathsf{3}$ being passed on channel $\mathsf{c}$. Channels may be declared using the keyword **channel**; for example, **channel** $\mathsf{c : Int}$ declares $\mathsf{c}$ to be a channel that passes an $\mathsf{Int}$. The notation $\{|\mathsf{c}|\}$ represents the set of events over channel $\mathsf{c}$. In this paper we will have to talk about both CSP channels and CSO channels: we will try to make clear which we mean in each case.

The simplest process is $\mathsf{STOP}$, which represents a deadlocked process that cannot communicate with its environment.

The process $\mathsf{a} \rightarrow \mathsf{P}$ offers its environment the event $\mathsf{a}$; if the event is performed, the process then acts like $\mathsf{P}$. The process $\mathsf{c?x} \rightarrow \mathsf{P}$ is initially willing to input a value $\mathsf{x}$ on channel $\mathsf{c}$, i.e. it is willing to perform any event of the form $\mathsf{c.x}$; it then acts like $\mathsf{P}$ (which may use $\mathsf{x}$). Similarly, the process $\mathsf{c?x{:}X} \rightarrow \mathsf{P}$ is willing to input any value $\mathsf{x}$ from set $\mathsf{X}$ on channel $\mathsf{c}$, and then act like $\mathsf{P}$ (which may use $\mathsf{x}$). The process $\mathsf{c!x} \rightarrow \mathsf{P}$ outputs value $\mathsf{x}$ on channel $\mathsf{c}$. Inputs and outputs may be mixed within the same communication, for example $\mathsf{c?x!y} \rightarrow \mathsf{P}$.

The process $\mathsf{P} \,\square\, \mathsf{Q}$ can act like either $\mathsf{P}$ or $\mathsf{Q}$, the choice being made by the environment: the environment is offered the choice between the initial events of $\mathsf{P}$ and $\mathsf{Q}$; hence the alt operator in CSO is very similar to the external choice operator of CSP. By contrast, $\mathsf{P} \sqcap \mathsf{Q}$ may act like either $\mathsf{P}$ or $\mathsf{Q}$, with the choice being made internally, not under the control of the environment. $\square\, \mathsf{x{:}X} \bullet \mathsf{P(x)}$ and $\bigsqcap \mathsf{x{:}X} \bullet \mathsf{P(x)}$ are indexed versions of these operators, with the choice being made over the processes $\mathsf{P(x)}$ for $\mathsf{x}$ in $\mathsf{X}$. The process $\mathsf{P} \triangleright \mathsf{Q}$ represents a sliding choice or timeout: it initially acts like $\mathsf{P}$, but if no event is performed then it can internally change state to act like $\mathsf{Q}$.

The process **if** b **then** P **else** Q represents a conditional. It will prove convenient to write assertions in our CSP models, similar in style to assertions in code. We define Assert(b)(P) as shorthand for **if** b **then** P **else** error $\rightarrow$ STOP; we will later check that the event error cannot occur, ensuring that all assertions are true.

The process $\mathsf{P} \ [| \ \mathsf{A} \ |] \ \mathsf{Q}$ runs P and Q in parallel, synchronising on events from A. The process $\mathsf{P} \ ||| \ \mathsf{Q}$ interleaves P and Q, i.e. runs them in parallel with no synchronisation. The process $|||\,\mathsf{x{:}X} \bullet \mathsf{P(x)}$ represents an indexed interleaving.

The process $\mathsf{P} \setminus \mathsf{A}$ acts like P, except the events from A are hidden, i.e. turned into internal, invisible events.

Prefixing ($\rightarrow$) binds tighter than each of the binary choice operators, which in turn bind tighter than the parallel operators.

A *trace* of a process is a sequence of (visible) events that a process can perform. We say that P is refined by Q in the traces model, written $\mathsf{P} \sqsubseteq_T \mathsf{Q}$, if every trace of Q is also a trace of P. FDR can test such refinements automatically, for finite-state processes. Typically, P is a specification process, describing what traces are acceptable; this test checks whether Q has only such acceptable traces.

Traces refinement tests can only ensure that no “bad” traces can occur: they cannot ensure that anything “good” actually happens; for this we need the stable failures or failures-divergences models. A *stable failure* of a process P is a pair $(tr, X)$, which represents that P can perform the trace $tr$ to reach a stable state (i.e. where no internal events are possible) where $X$ can be refused, i.e., where none of the events of $X$ is available. We say that P is refined by Q in the stable failures model, written $\mathsf{P} \sqsubseteq_F \mathsf{Q}$, if every trace of Q is also a trace of P, and every stable failure of Q is also a stable failure of Q.

We say that a process *diverges* if it can perform an infinite number of internal (hidden) events without any intervening visible events. In this paper, we will restrict ourselves to specification processes that cannot diverge. If P is such a process then we say that P is refined by Q in the failures-divergences model, written $\mathsf{P} \sqsubseteq_{FD} \mathsf{Q}$, if Q also cannot diverge, and every stable failure of Q is also a stable failure of P (which together imply that every trace of Q is also a trace of P). This test ensures that if P can stably offer an event a, then so can Q; hence such tests can be used to ensure Q makes useful progress. Again, such tests can be performed using FDR.

*Monitors*

A monitor is a program module —in Scala, an object— with a number of procedures that are intended to be executed under mutual exclusion. A simple monitor in Scala typically has a shape as below.

```
object Monitor{
  private var x ,...;  // private variables
  def procedure_1(arg_1 : T_1) = synchronized{...};
  ...
  def procedure_n(arg_n : T_n) = synchronized{...};
}
```

The keyword **synchronized** indicates a synchronized block: before a thread can enter the block, it must acquire the lock on the object; when it leaves the block, it releases the lock; hence at most one thread at a time can be executing within the code of the monitor.

It is sometimes necessary for a thread to suspend part way through a procedure, to wait for some condition to become true. It can do this by performing the command **wait**(); it releases the object's lock at this point. Another thread can wake it up by performing the command **notify**(); this latter thread retains the object's lock at this point, and the awoken thread must wait to re-obtain the lock.

The following producer-consumer example illustrates this technique. Procedures are available to put a piece of data into a shared slot, and to remove that data; each procedure might have to suspend, to wait for the slot to be emptied or filled, respectively.

```
object Slot{
  private var value = 0;   // the value in the slot
  private var empty = true; // is the slot empty?

  def put(v : Int) = synchronized{
    while(!empty) wait();   // wait until space is available
    value = v; empty = false; // store data
    notify(); // wake up consumer
  }

  def get : Int = synchronized{
    while(empty) wait(); // wait until value is available
    val result = value; empty = true; // get and clear value
    notify(); // wake up producer
    return result;
  }
}
```

An unfortunate feature of the implementation of **wait** within the Java Virtual Machine (upon which Scala is implemented) is that sometimes a process will wake up even if no other process has performed a **notify**, a so-called spurious wake-up. It is therefore recommended that all **wait**s are guarded by a boolean condition that is unset by the awakening thread; for example:

```
waiting = true; while(waiting) wait();
```

with awakening code:

```
waiting = false; notify();
```

## 1. Initial Design

In this section we present our initial design for the generalised alt. The design is not correct; however, our aims in presenting it are:

- to act as a stepping-stone towards a correct design;
- to illustrate some of the difficulties in producing a correct design;
- to introduce some features of the CSP models;
- to illustrate how model checking can discover flaws in a design.

For simplicity, we do not consider timeouts or the closing of channels within this model. We begin by describing the idea of the design informally, before presenting the CSP model and the analysis.

In order for an alt to fire a particular branch, say the branch for channel c, there must be another process —either another alt or not— willing to communicate on the other port of c. In order to ascertain this, an alt will register with the channel for each of its branches.

- If another process is already registered with channel c's other port, and ready to communicate, then c will respond to the registration request with YES, and the alt will select that branch. The act of registration represents a promise by the alt, that if it receives an immediate response of YES it will communicate.
- However, if no other process is registered with c's other port and ready to communicate, then c responds with NO, and the alt will continue to register with its other channels. In this case, the registration does not represent a firm promise to communicate, since it may select a different branch: it is merely an expression of interest.

If an alt has registered with each of its channels without receiving a positive response, then it waits to hear back from one of them. This process is illustrated in the first few steps of Figure 1: Alt1 registers with Chan1 and Chan2, receiving back a response of NO, before waiting.

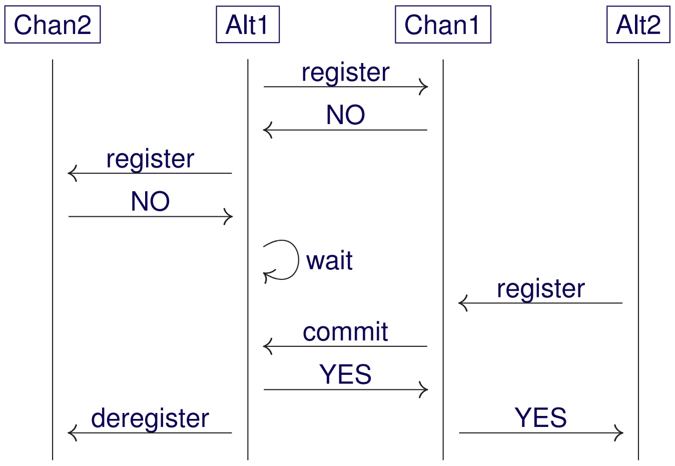

**Figure 1.** First sequence diagram

When a channel receives another registration attempt, it checks whether any of the alts already registered on its other port is able to commit to a communication. If any such alt agrees, the channel returns a positive response to the registering alt; at this point, both alts deregister from all other channels, and the communication goes ahead. However, if none of the registered alts is able to commit, then the channel returns a negative result to the registering alt. This process is illustrated in the last few steps of Figure 1. Alt2 registers with Chan1; Chan1 checks whether Alt1 can commit, and receives a positive answer, which is passed on to Alt2.

In the Scala implementation, our aim will be to implement the messages between components as procedure calls and returns. For example, the commit messages will be implemented by a procedure in the alt, also called commit; the responses will be implemented by the values returned from that procedure. A difference between the two types of components is that each alt will be *thread-like*: a thread will be executing the code of the alt (although at times that thread will be within procedure calls to other components); by contrast, channels will be *object-like*: they will be mostly passive, but willing to receive procedure calls from active threads.

## *1.1. CSP Model*

Each CSP model will be defined in two parts: a definition of a (generic) alt and channel; and the combination of several alts and channels into a system. The definition of each system will include two integer values, numAlts and numChannels, giving the number of alts and CSO channels, respectively. Given these, we can define the identities of alts and channels:

```
AltId = {1..numAlts} –– IDs of Alts
ChannelId = {1..numChannels} –– IDs of channels
```

We can further define a datatype of ports, and a datatype of responses:

```
datatype Port = InPort.ChannelId | OutPort.ChannelId
datatype Resp = YES | NO
```

We can now declare the CSP channels used in the model. The register, commit and deregister channels, and response channels for the former two, are declared as follows[2].

```
channel register : AltId . Port
channel registerResp : Port . AltId . Resp
channel commit : Port . AltId
channel commitResp : AltId . Port . Resp
channel deregister : AltId . Port
```

We also include a CSP channel on which each alt can signal that it thinks that it is executing a branch corresponding to a particular CSO channel; this will be used for specification purposes.

```
channel signal : AltId.ChannelId
```

The process Alt(me, ps) represents an alt with identity me with branches corresponding to the ports ps. It starts by registering with each of its ports. Below, reged is the set of ports with which it has registered, and toReg is the set of ports with which it still needs to register. It chooses (nondeterministically, at this level of abstraction) a port with which to register, and receives back a response; this is repeated until either it receives a positive response, or has registered with all the ports.

```
Alt(me,ps) = AltReg(me, ps, {}, ps)

AltReg(me, ps, reged, toReg) =
  if toReg=={} then AltWait(me, ps, reged)
  else
    ⊓ p : toReg •
        register.me.p → registerResp?p'!me?resp → Assert(p'==p)(
          if resp==YES then AltDereg(me, ps, remove(reged,p), p)
          else AltReg(me, ps, add(reged,p), remove(toReg,p))
        )
```

Here we use two helper functions, to remove an element from a set, and to add an element to a set:[3]

```
remove(xs,x) = diff(xs,{x})
add(xs,x) = union(xs,{x})
```

[2] deregister does not return a result, and can be treated as atomic, so we do not need a response channel

[3] diff and union are the machine-readable CSP functions for set difference and union.

If the alt registers unsuccessfully with each of its ports, then it waits to receive a commit message from a port, which it accepts.

```
AltWait(me, ps, reged) =
  commit?p:reged!me → commitResp.me.p!YES →
  AltDereg(me, ps, remove(reged,p), p)
```

Once an alt has committed to a particular port, p, it deregisters with each of the other ports, and then signals, before returning to its initial state. During the same time, if the alt receives a commit event, it responds negatively.

```
AltDereg(me, ps, toDereg, p) =
  if toDereg=={} then signal.me.chanOf(p) → Alt(me,ps)
  else (
    ( ⊓ p1:toDereg •
        deregister.me.p1 → AltDereg(me, ps, remove(toDereg,p1), p) )
    □
    commit?p1:aports(me)!me → commitResp.me.p1!NO →
    AltDereg(me, ps, toDereg, p)
  )
```

Here chanOf returns the channel corresponding to a port:

```
chanOf(InPort.c) = c
chanOf(OutPort.c) = c
```

We now consider the definition of a channel. The process Channel(me, reged) represents a channel with identity me, where reged is a set of (port, alt) pairs, showing which alts have registered at its two ports.

```
Channel(me, reged) =
  register?a?port:ports(me) → (
    let toTry = {(p,a1) | (p,a1) ← reged, p==otherP(port)} within
    ChannelCommit(me, a, port, reged, toTry)
  )
  □
  deregister?a?p:ports(me) → Channel(me,remove(reged,(p,a)))
```

Here, ports(me) gives the ports corresponding to this channel:

```
ports(me) = {InPort.me, OutPort.me}
```

The set toTry, above, represents all the previous registrations with which this new registration might be matched; otherP(port) returns this channel's other port.

```
otherP(InPort.me) = OutPort.me
otherP(OutPort.me) = InPort.me
```

The channel now tries to find a previous registration with which this new one can be paired. The parameter toTry represents those previous registrations with which the channel still needs to check. The channel chooses (nondeterministically) a previous registration to try, and sends a commit message. It repeats until either (a) it receives back a positive response, in which case it sends a positive response to the registering alt a, or (b) it has exhausted all possibilities, in which case it sends back a negative response.[4]

[4]The notation ⊓pa' @@(port',a') : toTry binds the identifier pa' to an element of toTry, and also binds the identifiers port' and a' to the two components of pa'.

```
ChannelCommit(me, a, port, reged, toTry) =
  if toTry=={} then -- None can commit
    registerResp.port.a!NO → Channel(me, add(reged,(port,a)))
  else(
    ⊓ pa'@@(port',a') : toTry •
      commit.port'.a' → commitResp.a'.port'?resp →
      if resp==YES then
        registerResp.port.a!YES → Channel(me, remove(reged,pa'))
      else
        ChannelCommit(me, a, port, remove(reged,pa'), remove(toTry,pa'))
  )
```

## *1.2. Analysing the Design*

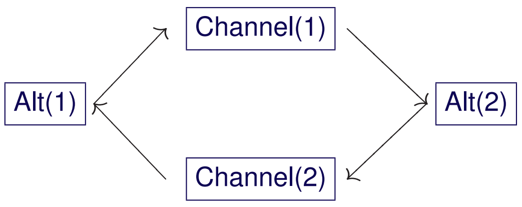

**Figure 2.** A simple configuration

We consider a simple configuration of two alts and two channels, as in Figure 2 (where the arrows indicate the direction of dataflow, so Alt(1) accesses Channel(1)'s inport and Channel(2)'s outport, for example). This system can be defined as follows.

```
numAlts = 2
numChannels = 2

Channels = ||| me : ChannelId • Channel(me, {})

aports(1) = {InPort.1, OutPort.2}
aports(2) = {InPort.2, OutPort.1}

Procs = ||| me : AltId • Alt(me, aports(me))

System =
  let internals = {|register,registerResp,commit,commitResp,deregister|}
  within (Channels [|internals|] Procs) \ internals
```

The two processes should agree upon which channel to communicate; that is, they should (repeatedly) signal success on the same channel. Further, no error events should occur. This requirement is captured by the following CSP specification.

```
Spec =
  ⊓ c:ChannelId •
     signal.1.c → signal.2.c → Spec □ signal.2.c → signal.1.c → Spec
```

When we use FDR to test if System refines Spec in the traces model, the test succeeds. However, when we do the corresponding test in the stable failures model, the test fails, because System deadlocks. Using the FDR debugger shows that the deadlock occurs after the system (without the hiding) has performed

```
<register.2.InPort.2, register.1.InPort.1, registerResp.InPort.2.2.NO,
 registerResp.InPort.1.1.NO, register.1.OutPort.2, register.2.OutPort.1>
```

This is illustrated in Figure 3. Each alt has registered with one channel, and is trying to

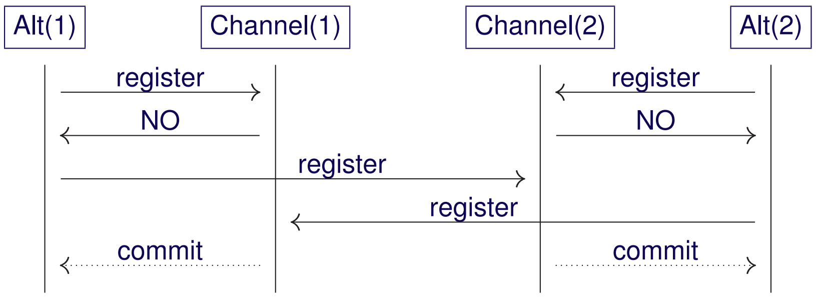

**Figure 3.** The behaviour leading to deadlock

register with its other channel. In the deadlocked state, Channel(1) is trying to send a commit message to Alt(1), but Alt(1) refuses this because it is waiting for a response to its last register event; Channel(2) and Alt(2) are behaving similarly.

The following section investigates how to overcome this problem.

## 2. Improved Design

The counterexample in the previous section shows that alts should be able to accept commit messages while waiting for a response to a register. But how should an alt deal with such a commit? It would be wrong to respond with YES, for then it would be unable to deal with a response of YES to the register message (recall that an alt must respect a response of YES to a register message). It would also be wrong to respond NO to the commit, for then the chance to communicate on this channel would be missed. Further, a little thought shows that delaying replying to the commit until after a response to the register has been received would also be wrong: in the example of the last section, this would again lead to a deadlock.

Our solution is to introduce a different response, MAYBE, that an alt can send in response to a commit; informally, the response of MAYBE means “I’m busy right now; please call back later”. The sequence diagram in Figure 4 illustrates the idea. Alt1 receives a commit from Chan1 while waiting for a response to a register. It sends back a response of MAYBE, which gets passed back to the initiating Alt2. Alt2 pauses for a short while (to give Alt1 a chance to finish what it’s doing), before again trying to register with Chan1. Note that it is the alt’s responsibility to retry, rather than the channel’s, because we are aiming for an implementation where the alt is thread-like, but the channel is object-like.

### *2.1. CSP Model*

We now adapt the CSP model from the previous section to capture this idea. First, we expand the type of responses to include MAYBE:

```
datatype Resp = YES | NO | MAYBE
```

When a channel pauses before retrying, it will signal on the channel pause; we will later use this for specification purposes.

```
channel pause : AltId
```

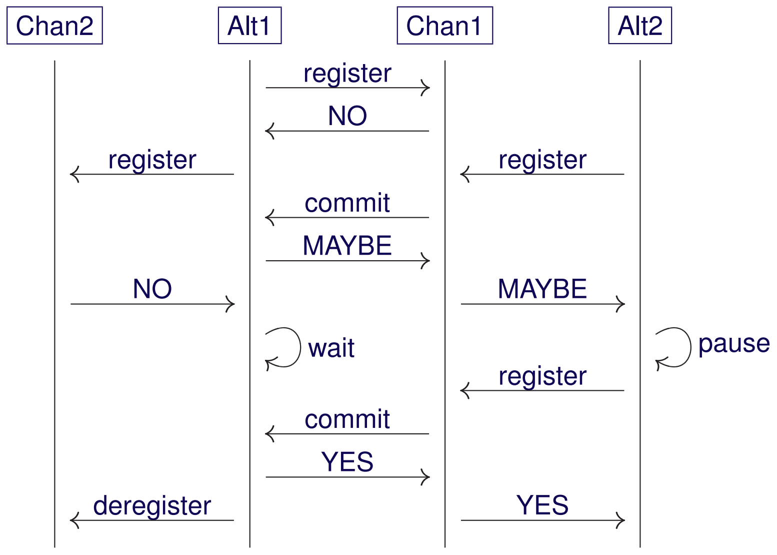

**Figure 4.** Using MAYBE

An alt again starts by registering with each of its channels. It may now receive a response of MAYBE; the parameter maybes below stores those ports for which it has received such a response. Further, it is willing to receive a commit message during this period, in which case it responds with MAYBE.

```
Alt(me,ps) = AltReg(me, ps, {}, ps, {})

AltReg(me, ps, reged, toReg, maybes) =
  if toReg=={} then
    if maybes=={} then AltWait(me, ps, reged)
    else
      pause.me → AltPause(me, ps, reged, maybes)
      □
      commit?p:aports(me)!me → commitResp.me.p!MAYBE →
      AltReg(me, ps, reged, toReg, maybes)
  else
    ( ⊓ p : toReg •
        register.me.p → AltReg'(me, ps, reged, toReg, maybes, p) )
    □
    commit?p:aports(me)!me → commitResp.me.p!MAYBE →
    AltReg(me, ps, reged, toReg, maybes)

-- Waiting for response from p
AltReg'(me, ps, reged, toReg, maybes, p) =
  registerResp?p'!me?resp → Assert(p'==p)(
    if resp==YES then AltDereg(me, ps, remove(reged,p), p)
    else if resp==NO then
      AltReg(me, ps, add(reged,p), remove(toReg,p), maybes)
    else -- resp==MAYBE
      AltReg(me, ps, reged, remove(toReg,p), add(maybes,p))
  )
  □
  commit?p1:aports(me)!me → commitResp.me.p1!MAYBE →
  AltReg'(me, ps, reged, toReg, maybes, p)
```

If an alt receives no positive response, and at least one MAYBE, it pauses for a short while before retrying. However, it accepts any commit request it receives in the mean time.[5]

```
AltPause(me, ps, reged, maybes) =
  ( STOP ▷ AltReg(me, ps, reged, maybes, {}) )
  □
  commit?p:aports(me)!me → commitResp.me.p!YES →
  AltDereg(me, ps, remove(reged,p), p)
```

If an alt receives only negative responses to its register messages, it again waits.

```
AltWait(me, ps, reged) =
  commit?p:aports(me)!me → commitResp.me.p!YES →
  AltDereg(me, ps, remove(reged,p), p)
```

Once the alt has committed, it deregisters the other ports, and signals, as in the previous model.

```
AltDereg(me, ps, toDereg, p) =
  if toDereg=={} then signal.me.chanOf(p) → Alt(me,ps)
  else (
    ( ⊓ p1:toDereg •
        deregister.me.p1 → AltDereg(me, ps, remove(toDereg,p1), p) )
    □
    commit?p1:aports(me)!me → commitResp.me.p1!NO →
    AltDereg(me, ps, toDereg, p)
  )
```

The definition of a channel is a fairly straightforward adaptation from the previous model. In the second process below, the parameter maybeFlag is true if any alt has responded MAYBE. The port is registered at the channel only if each register message received a response of NO.

```
Channel(me, reged) =
  register?a?port:ports(me) → (
    let toTry = {(p,a1) | (p,a1) ← reged, p==otherP(port)} within
    ChannelCommit(me, a, port, reged, toTry, false)
  )
  □
  deregister?a.p → Channel(me,remove(reged,(p,a)))

ChannelCommit(me, a, port, reged, toTry, maybeFlag) =
  if toTry=={} then -- None can commit
    if maybeFlag then
      registerResp.port.a!MAYBE → Channel(me, reged)
    else
      registerResp.port.a!NO → Channel(me, add(reged,(port,a)))
  else(
    ⊓ pa' @@ (port',a') : toTry •
      commit.port'.a' → commitResp.a'.port'?resp →
      if resp==YES then
        registerResp.port.a!YES → Channel(me, remove(reged,pa'))
      else if resp==MAYBE then
        ChannelCommit(me, a, port, reged, remove(toTry,pa'), true)
      else -- resp==NO
```

[5]CSP-cognoscenti may point out that the “STOP ▷ ” does not affect the behaviour of the process; we include it merely to illustrate the desired behaviour of our later Scala implementation.

```
        ChannelCommit(me, a, port, remove(reged,pa'),
                      remove(toTry,pa'), maybeFlag)
  )
```

## 2.2. *Analysing the Design*

We can again combine these alts and channels into various configurations. First, we consider the configuration in Figure 2; this is defined as earlier, but also hiding the pause events. FDR can then be used to verify that this system refines the specification Spec, in both the traces and the stable failures model.

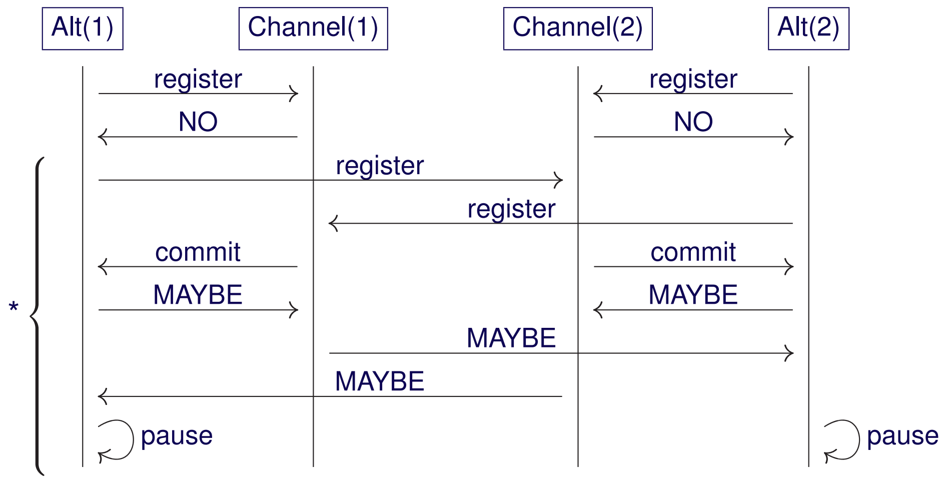

**Figure 5.** Behaviour causing divergence

However, the refinement does not hold in the failures-divergences model, since the system can diverge. The divergence can happen in a number of different ways; one possibility is shown in Figure 5[6]. Initially, each alt registers with one channel. When each alt tries to register with the other channel, a commit message is sent to the other alt, receiving a response of MAYBE; each alt then pauses. These attempts to register (marked "$*$" in the diagram) can be repeated arbitrarily many times, causing a divergence. The problem is that the two alts are behaving symmetrically, each sending its register events at about the same time: if one alt were to send its register while the other is pausing, it would receive back a response of YES, and the symmetry would be broken. In the implementation, the pause will be of a random amount of time, to ensure the symmetry is eventually broken (with probability 1).

We can check that the *only* way that the system can diverge is through repeated pauses and retries. We can show that the system without the pause events hidden refines the following specification: each alt keeps on pausing until both signal.

```
SpecR =
  ( ⊓ p:ChannelId • signal.1.p → SpecR_1(p) □ signal.2.p → SpecR_2(p))
  ⊓ pause.1 → SpecR ⊓ pause.2 → SpecR
SpecR_1(p) = signal.2.p → SpecR ⊓ pause.1 → SpecR_1(p)
SpecR_2(p) = signal.1.p → SpecR ⊓ pause.2 → SpecR_2(p)
```

We have built other configurations, including those in Figure 6. For each, we have used

[6]In fact, FDR finds a slightly simpler divergence, where only one alt repeatedly tries to register; in the implementation, this would correspond to the other alt being starved of the processor; we consider the example in the figure to be more realistic.

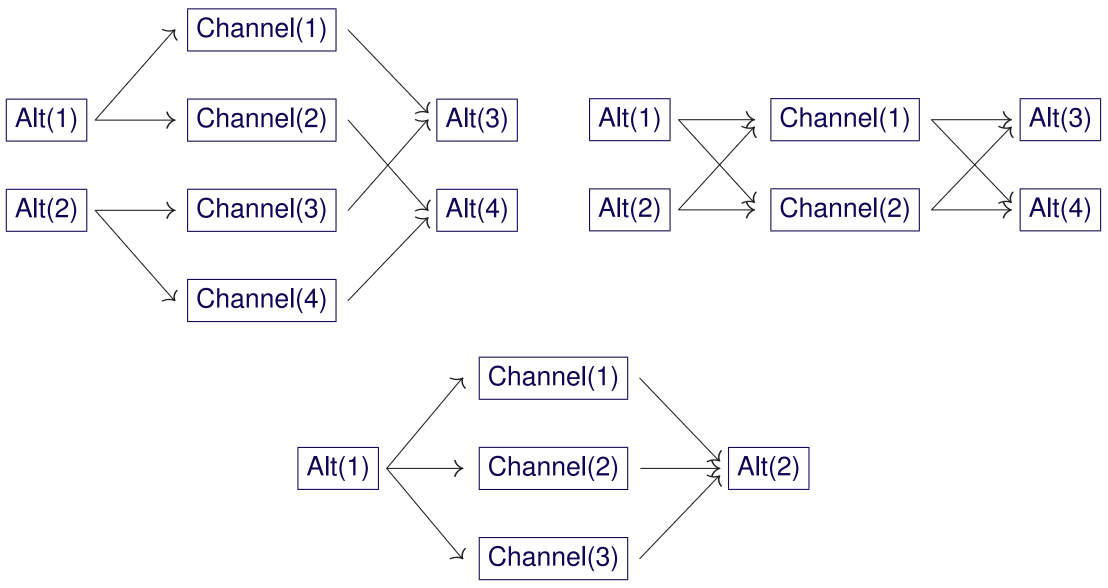

**Figure 6.** Three test configurations

FDR to check that it refines a suitable specification that ensures that suitable signal events are available, in particular that if an alt signals at one port of a channel then another signals at the other port. We omit the details in the interests of brevity.

But as the alts and channels are *components*, we would really like to analyse *all* systems built from them: this seems a particularly difficult case of the parameterised model checking problem, beyond the capability of existing techniques.

## 3. Compound Alts

The model in the previous section captures the desired behaviour of an alt. However, it does not seem possible to implement this behaviour using a single monitor. We would like to implement the main execution of the alt as a procedure apply, and to implement the commit and commitResp events as a procedure commit and its return. However, these two procedures will need to be able to run concurrently, so cannot be implemented in a single monitor.

Instead we implement the alt using two monitors.

- The MainAlt will implement the apply procedure, to register with the channels, deregister at the end, execute the appropriate branch of the alt, and generally control the execution.
- The Facet will provide the commit procedure, responding appropriately; it will receive messages from the MainAlt, informing it of its progress; if the Facet receives a call to commit while the MainAlt is waiting, the Facet will wake up the MainAlt.

The definition of a channel remains the same as in the previous section.

Figure 7 illustrates a typical scenario, illustrating how the two components cooperate together to achieve the behaviour of Alt1 from Figure 1. The MainAlt starts by initialising the Facet, and then registers with Chan1. When the Facet receives a commit message from Chan1, it replies with MAYBE, since it knows the MainAlt is still registering with channels. When the MainAlt finishes registering, it informs the Facet, and then waits. When the Facet subsequently receives another commit message, it wakes up the MainAlt, passing the identity of Chan1, and returns YES to Chan1. The MainAlt deregisters the other channels, and informs the Facet. In addition, if the Facet had received another commit message after sending YES to Chan1, it would have replied with NO.

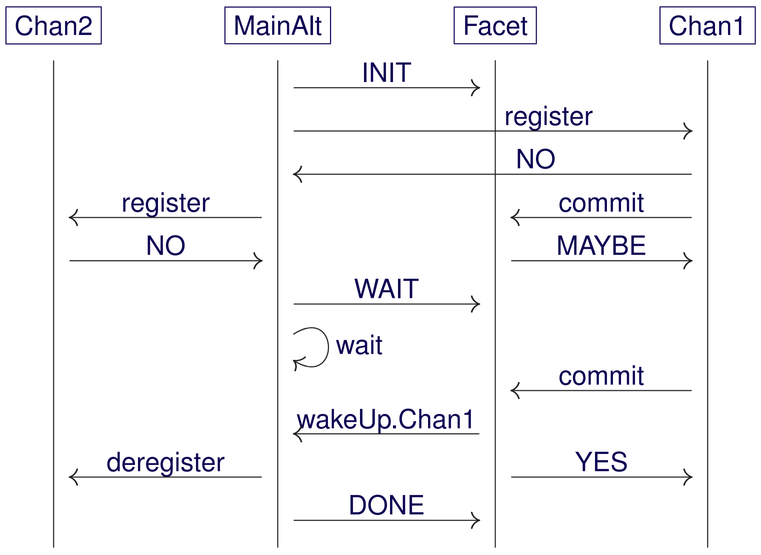

**Figure 7.** Expanding the alt

As noted earlier, if the MainAlt receives any reply of MAYBE when trying to register with channels, it pauses for a short while, before retrying; Figures 8 and 9 illustrate this for the compound alt (starting from the point where the alt tries to register with Chan2). Before pausing, the MainAlt informs the Facet. If the Facet receives a commit in the meantime, it replies YES (and would reply NO to subsequent commits). When the MainAlt finishes pausing, it checks back with the Facet to find out if any commit was received, getting a positive answer in Figure 8, and a negative one in Figure 9.

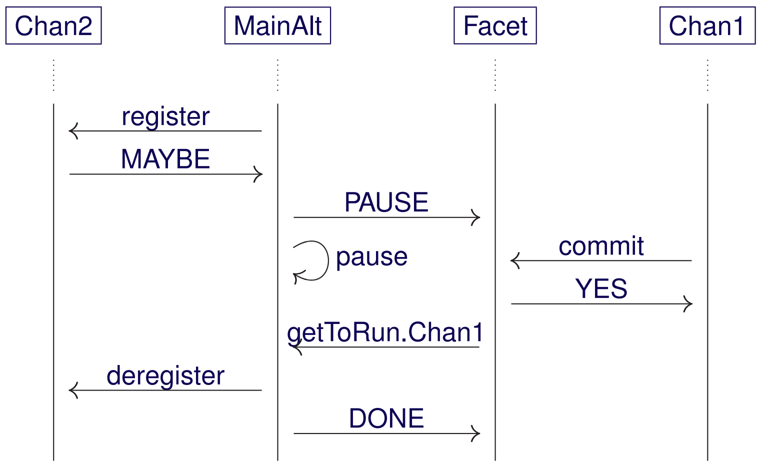

**Figure 8.** A commit received while pausing

## 3.1. *CSP Model*

We now describe a CSP model that captures the behaviour described informally above. We define a datatype and channel by which the MainAlt informs the Facet of changes of status.

```
datatype Status = Init | Pause | Wait | Dereg | Done
channel changeStatus : Status
```

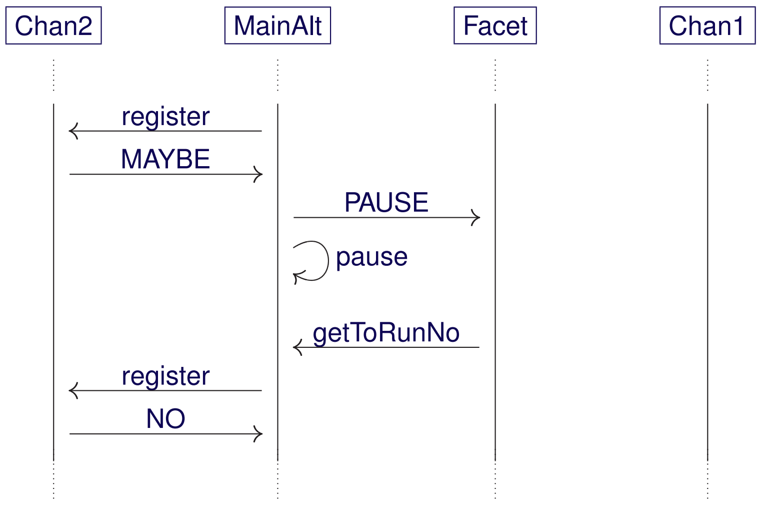

**Figure 9.** Pausing before retrying

When the Facet wakes up the MainAlt, it sends the identity of the port whose branch should be run, on channel wakeUp.

```
channel wakeUp : Port
```

When the MainAlt finishes pausing, it either receives from the Facet on channel getToRun the identity of a port from whom a commit has been received, or receives a signal getToRunNo that indicates that no commit has been received.

```
channel getToRun : Port
channel getToRunNo
```

The alt is constructed from the two components, synchronising on and hiding the internal communications:

```
Alt(me, ps) =
  let A = {| wakeUp, changeStatus, getToRun, getToRunNo |} within
  (MainAlt(me, ps) [| A |] Facet(me)) \ A
```

The definition of the MainAlt is mostly similar to the definition of the alt in Section 2, so we just outline the differences here. The MainAlt does not receive the commit messages, but instead receives notifications from the Facet. When it finishes pausing (state MainAltPause below), it either receives from the Facet the identity of the branch to run on channel getToRun, or receives on channel getToRunNo an indication that no commit event has been received. When it is waiting (state MainAltWait), it waits until it receives a message from the Facet on channel wakeUp, including the identity of the process to run.

```
MainAlt(me, ps) = changeStatus!Init → MainAltReg(me, ps, {}, ps, {})

MainAltReg(me, ps, reged, toReg, maybes) =
  if toReg=={} then
    if maybes=={} then MainAltWait(me, ps, reged)
    else
      pause.me → changeStatus!Pause → MainAltPause(me, ps, reged, maybes)
  else
    ⊓ p : toReg •
      register.me.p → registerResp?p'!me?resp → Assert(p'==p)(
      if resp==YES then
```

```
        changeStatus!Dereg → MainAltDereg(me, ps, remove(reged,p), p)
      else if resp==NO then
        MainAltReg(me, ps, add(reged,p), remove(toReg,p), maybes)
      else -- resp==MAYBE
        MainAltReg(me, ps, reged, remove(toReg,p), add(maybes,p))
      )

MainAltPause(me, ps, reged, maybes) =
  STOP ▷ (
    getToRunNo → MainAltReg(me, ps, reged, maybes, {})
    □
    getToRun?p → MainAltDereg(me, ps, remove(reged,p), p)
  )

MainAltWait(me, ps, reged) =
  changeStatus!Wait → wakeUp?p:reged →
  MainAltDereg(me, ps, remove(reged,p), p)

MainAltDereg(me, ps, toDereg, p) =
  if toDereg=={} then
    changeStatus!Done → signal.me.chanOf(p) → MainAlt(me,ps)
  else
    ⊓ p1:toDereg •
      deregister.me.p1 → MainAltDereg(me, ps, remove(toDereg,p1), p)
```

The Facet tracks the state of the MainAlt; below we use similar names for the states of the Facet as for the corresponding states of MainAlt. When the MainAlt is pausing, the Facet responds YES to the first commit it receives (state FacetPause), and NO to subsequent ones (state FacetPause'); it passes on this information on getToRun or getToRunNo. When the MainAlt is waiting, if the Facet receives a commit message, it wakes up the MainAlt (state FacetWait).

```
Facet(me) = changeStatus.Init → FacetReg(me)

FacetReg(me) =
  commit?p:aports(me)!me → commitResp.me.p!MAYBE → FacetReg(me)
  □
  changeStatus?s →
  if s==Wait then FacetWait(me)
  else if s==Dereg then FacetDereg(me)
  else Assert(s==Pause)(FacetPause(me))

FacetPause(me) =
  commit?p:aports(me)!me → commitResp.me.p!YES → FacetPause'(me,p)
  □
  getToRunNo → FacetReg(me)

FacetPause'(me,p) =
  commit?p1:aports(me)!me → commitResp.me.p1!NO → FacetPause'(me,p)
  □
  getToRun!p → FacetDereg(me)

FacetWait(me) =
  commit?p:aports(me)!me → wakeUp!p →
  commitResp.me.p!YES → FacetDereg(me)

FacetDereg(me) =
  commit?p:aports(me)!me → commitResp.me.p!NO → FacetDereg(me)
  □
  changeStatus?s → Assert(s==Done)(Facet(me))
```

*3.2. Analysing the Design*

We have built configurations, using this compound alt, as in Figures 2 and 6. We have again used FDR to check that each refines a suitable specification.

In fact, the compound alt defined in this section is not equivalent to, or even a refinement of, the sequential alt defined in the previous section. The compound alt has a number of behaviours that the sequential alt does not, caused by the fact that it takes some time for information to propagate through the former. For example, the compound alt can register with each of its ports, receiving NO in each case, and then return MAYBE in response to a commit message (whereas the sequential alt would return YES), because the (internal) changeStatus.Wait event has not yet happened. We see the progression from the sequential to the compound alt as being a step of *development* rather than formal refinement: such (typically small) changes in behaviour are common in software development.

## 4. Adding Timeouts and Closing of Channels

We now extend our compound model from the previous section to capture two additional features of alts, namely timeouts and the closing of channels. We describe these features separately from the main operation of alts, since they are rather orthogonal. Further, this follows the way we developed the implementation, and how we would recommend similar developments are carried out: get the main functionality right, then add the bells and whistles.

We describe the treatment of timeouts first. If the alt has a timeout branch, then the waiting stage from the previous design is replaced by a timed wait. If the Facet receives a commit during the wait, it can wake up the MainAlt, much as in Figure 7. Alternatively, if the timeout time is reached, the alt can run the timeout branch. However, there is a complication: the Facet may receive a commit at almost exactly the same time as the timeout is reached — a race condition. In order to resolve this race, we introduce a third component into the compound alt: the Arbitrator will arbitrate in the event of such a race, so that the Facet and MainAlt proceed in a consistent way.

Figure 10 corresponds to the earlier Figure 7. The WAIT message informs the Facet that the MainAlt is performing a wait with a timeout. When the Facet subsequently receives a commit message, it checks with the Arbitrator that this commit has not been preempted by a timeout. In the figure, it receives a returned value of true, indicating that there was no race, and so the commit request can be accepted.

Figure 11 considers the case where the timeout is reached without a commit message being received in the meantime. The MainAlt checks with the Arbitrator that indeed no commit message has been received, and then deregisters all channels before running the timeout branch.

Figures 12 and 13 consider cases where the timeout happens at about the same time as a commit is received. The MainAlt and the Facet both contact the Arbitrator; whichever does so first “wins” the race, so the action it is dealing with is the one whose branch will be executed. If the Facet wins, then the MainAlt waits for the Facet to wake it up (Figure 12). If the MainAlt wins, then the Facet replies NO to the commit, and waits for the MainAlt to finish deregistering channels (Figure 13).

We now consider the treatment of channels closing. Recall that if there is no timeout branch and all the channels close, then the alt should run its orelse branch, if there is one, or throw an Abort exception. However, if there is a timeout branch, then it doesn’t matter if all the branches are closed: the timeout branch will eventually be selected.

When a channel closes, it sends a chanClosed message to each alt that is registered with it; this message is received by the Facet, which keeps track of the number of channels that have

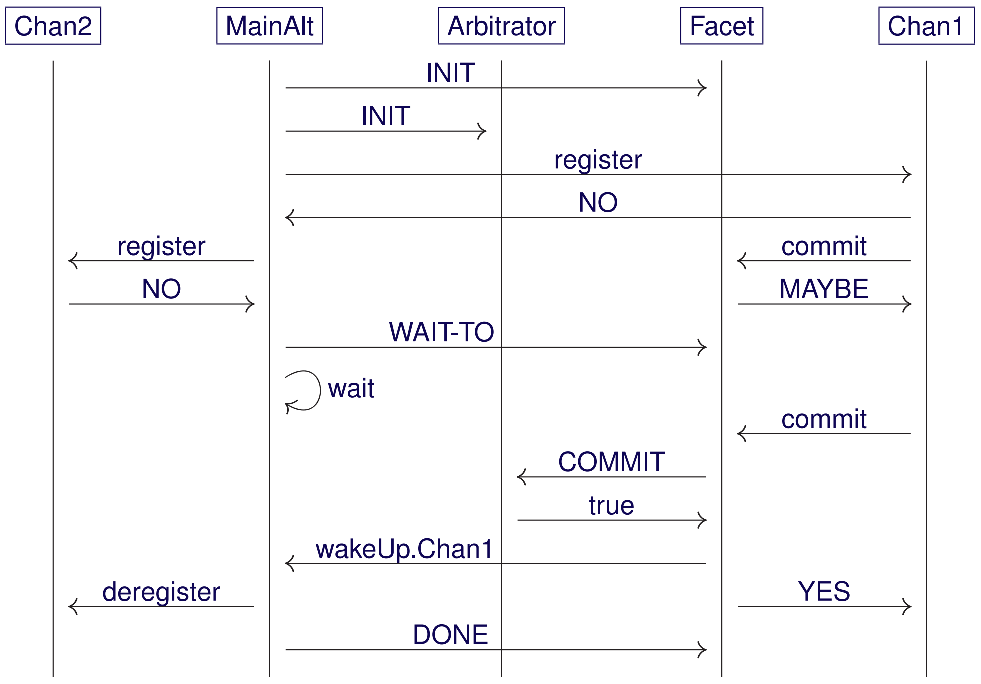

**Figure 10.** Expanding the alt

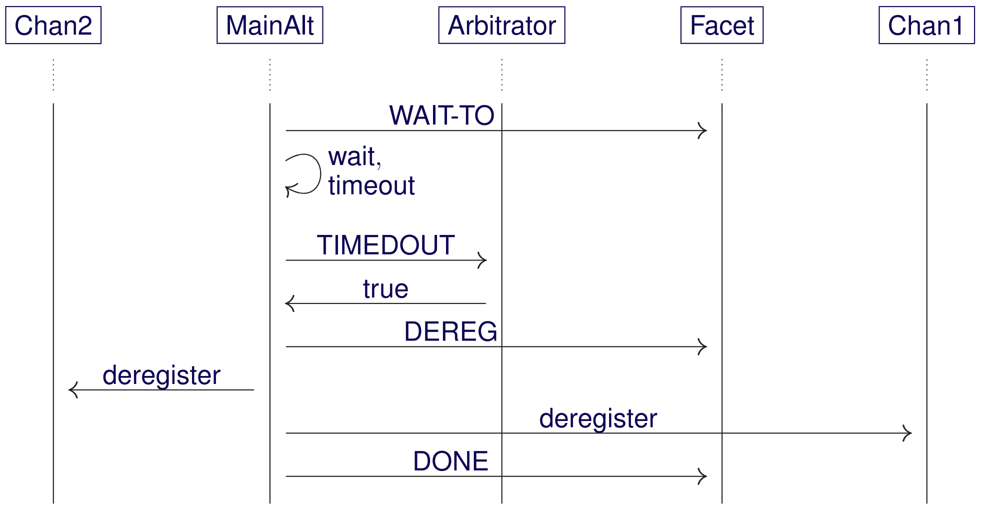

**Figure 11.** After a timeout

closed. If an alt subsequently tries to register with the closed channel, it returns a response of CLOSED.

When the MainAlt is about to do a non-timed wait, it sends the Facet a setReged message (replacing the WAIT message in Figure 7), including a count of the number of channels with which it has registered. The Facet returns a boolean that indicates whether all the channels have closed. If so, the MainAlt runs its orelse branch or throws an Abort exception. Otherwise, if subsequently the Facet receives sufficient chanClosed messages such that all channels are closed, it wakes up the MainAlt by sending it an allClosed message; again, the MainAlt either runs its orelse branch or throws an Abort exception.

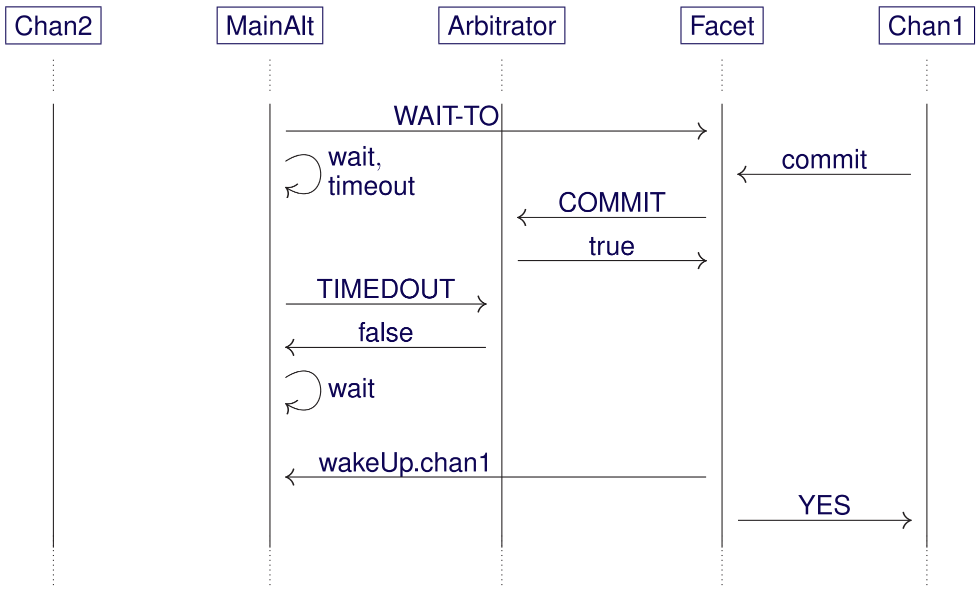

**Figure 12.** A commit beating a timeout in a race

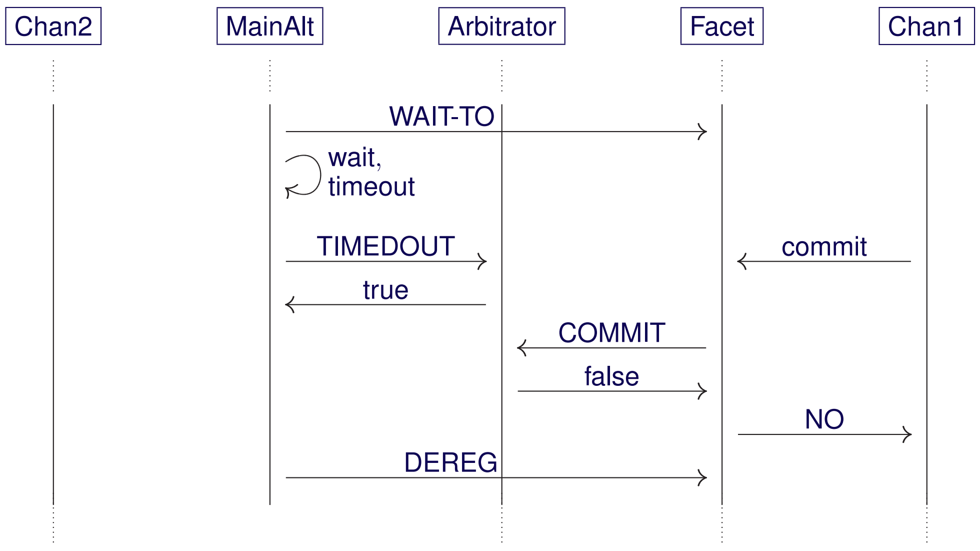

**Figure 13.** A timeout beating a commit in a race

### *4.1. CSP Model*

We now describe a CSP model that capture the behaviour described informally above. We extend the types of ports, responses and status values appropriately.

```
datatype Port = InPort.ChannelId | OutPort.ChannelId | TIMEOUT | ORELSE
datatype Resp = YES | NO | MAYBE | CLOSED
datatype Status = Init | Pause | WaitTO | Done |
                  Dereg | Commit | Timedout
```

We extend the type of signals to include timeouts and orelse. We also include events to indicate that a process has aborted, and (to help interpret debugging traces) that a process has timed out.

```
TIMEOUTSIG = 0
ORELSESIG = −1
```

```
channel signal : AltId . union(ChannelId,{TIMEOUTSIG,ORELSESIG})
channel abort : AltId
channel timeout : AltId
```

Finally we add channels for a (CSO) channel to inform an alt that it has closed, and for communications with the Arbitrator (for simplicity, the latter channel captures communications in both directions using a single event).

```
channel chanClosed : Port . AltId
channel checkRace : Status . Bool
```

The alt is constructed from the three components, synchronising on and hiding the internal communications:

```
Alt(me, ps) =
  let A = {| wakeUp, changeStatus, getToRun, getToRunNo |} within
  ((MainAlt(me, ps) [| A |] Facet(me)) [| {|checkRace|} |] Arbitrator(Init))
     \ union(A,{|checkRace|})
```

The definition of the MainAlt is mostly similar to as in Section 3, so we just describe the main differences here. It starts by initialising the other two components, before registering with channels as earlier.

```
MainAlt(me, ps) =
  changeStatus!Init → checkRace.Init?b → MainAltReg(me, ps, {}, ps, {})

MainAltReg(me, ps, reged, toReg, maybes) =
  if toReg=={} then
    if maybes=={} then
      if member(TIMEOUT,ps) then MainAltWaitTimeout(me, ps, reged)
      else MainAltWait(me, ps, reged)
    else
      retry.me → changeStatus!Pause → MainAltPause(me, ps, reged, maybes)
  else
    ⊓ p : toReg •
      if p==TIMEOUT or p==ORELSE then
        MainAltReg(me, ps, reged, remove(toReg,p), maybes)
      else
        register.me.p → registerResp?p'!me?resp → Assert(p'==p)(
        if resp==YES then
          changeStatus!Dereg → MainAltDereg(me, ps, remove(reged,p), p)
        else if resp==NO then
          MainAltReg(me, ps, add(reged,p), remove(toReg,p), maybes)
        else  -- resp==MAYBE
          MainAltReg(me, ps, reged, remove(toReg,p), add(maybes,p))
      )

MainAltPause(me, ps, reged, maybes) =
  STOP ▷ (
    getToRunNo → MainAltReg(me, ps, reged, maybes, {})
    □ getToRun?p → MainAltDereg(me, ps, remove(reged,p), p)
  )
```

Before doing an untimed wait, the MainAlt sends a message to the Facet on setReged, giving the number of registered channels, and receiving back a boolean indicating whether all branches are closed. If so (state MainAltAllClosed) it runs the orelse branch if there is one, or aborts. If not all branches are closed, it waits to receive either a wakeUp or allClosed message.

```
MainAltWait(me, ps, reged) =
  setReged!card(reged)?allBranchesClosed →
  if allBranchesClosed then MainAltAllClosed(me, ps, reged)
  else -- wait for signal from Facet
    wakeUp?p:reged → MainAltDereg(me, ps, remove(reged,p), p)
    □ allClosed → MainAltAllClosed(me, ps, reged)

MainAltAllClosed(me, ps, reged) =
  if member(ORELSE,ps) then
    changeStatus!Dereg → MainAltDereg(me, ps, reged, ORELSE)
  else abort.me → STOP
```

The state MainAltWaitTimeout describes the behaviour of waiting with the possibility of selecting a timeout branch. The MainAlt can again be woken up by a wakeUp event; we also model the possibility of an allClosed event, but signal an error if one occurs (subsequent analysis with FDR verifies that they can't occur). We signal a timeout on the timeout channel. The MainAlt then checks with the Arbitrator whether it has lost a race with a commit; if not (**then** branch) it runs the timeout branch; otherwise (**else** branch) it waits to be woken by the Facet.

```
MainAltWaitTimeout(me, ps, reged) =
  changeStatus!WaitTO → (
    ( wakeUp?p:reged → MainAltDereg(me, ps, remove(reged,p), p)
      □ allClosed → error → STOP
    )
    ▷
    timeout.me → checkRace.Timedout?resp →
      if resp then
        changeStatus!Dereg → MainAltDereg(me, ps, reged, TIMEOUT)
      else wakeUp?p:reged → MainAltDereg(me, ps, remove(reged,p), p)
  )

MainAltDereg(me, ps, toDereg, p) =
  if toDereg=={} then
    changeStatus!Done → signal.me.chanOf(p) → MainAlt(me,ps)
  else
    ⊓ p1:toDereg •
      deregister.me.p1 → MainAltDereg(me, ps, remove(toDereg,p1), p)
```

The model of the Facet is a fairly straightforward extension of that in Section 3, dealing with the closing of channels and communications with the Arbitrator as described above.

```
Facet(me) = changeStatus?s → Assert(s==Init)(FacetReg(me,0))

FacetReg(me,closed) =
  commit?p:aports(me)!me → commitResp.me.p!MAYBE → FacetReg(me,closed)
  □
  changeStatus?s → (
    if s==WaitTO then FacetWaitTimeout(me,closed)
    else if s==Dereg then FacetDereg(me)
    else Assert(s==Pause)(FacetPause(me,closed))
  )
  □
  chanClosed?p:aports(me)!me →
  Assert(closed<numChannels)(FacetReg(me,closed+1))
  □
  setReged?nreged!(nreged==closed) → FacetWait(me,closed,nreged)

FacetPause(me,closed) =
  commit?p:aports(me)!me → commitResp.me.p!YES → FacetPause'(me, closed ,p)
  □
  getToRunNo → FacetReg(me, closed)
  □
  chanClosed?p:aports(me)!me →
  Assert(closed<numChannels)(FacetPause(me, closed+1))

FacetPause'(me,p) =
  commit?p1:aports(me)!me → commitResp.me.p1!NO → FacetPause'(me,p)
  □
  getToRun!p → FacetDereg(me)
  □
  chanClosed?p1:aports(me)!me → FacetPause'(me,p)

FacetWait(me, closed ,nreged) =
  commit?p:aports(me)!me → checkRace.Commit?resp → (
     if resp then  -- wake up body of MainAlt
       wakeUp!p → commitResp.me.p!YES → FacetDereg(me)
    else commitResp.me.p!NO → FacetWait(me, closed ,nreged)
  )
  □
  changeStatus?s → Assert(s==Dereg)(FacetDereg(me))
  □
  chanClosed?p:aports(me)!me →
  Assert(closed<numChannels)(
     if nreged==closed+1 then allClosed → FacetWait(me, closed+1,nreged)
    else FacetWait(me, closed+1,nreged)
  )

FacetWaitTimeout(me, closed) =
  commit?p:aports(me)!me → checkRace.Commit?resp → (
     if resp then  -- wake up body of MainAlt
       wakeUp!p → commitResp.me.p!YES → FacetDereg(me)
    else commitResp.me.p!NO → FacetWaitTimeout(me, closed)
  )
  □
  changeStatus?s → Assert(s==Dereg)(FacetDereg(me))
  □
  chanClosed?p:aports(me)!me →
  Assert(closed<numChannels)(FacetWaitTimeout(me, closed+1))

FacetDereg(me) =
  commit?p:aports(me)!me → commitResp.me.p!NO → FacetDereg(me)
  □
  changeStatus?s → Assert(s==Done)(Facet(me))
  □
  chanClosed?p:aports(me)!me → FacetDereg(me)
```

The Arbitrator keeps track of whether it has received a Timedout or Commit message, or neither, on this round; it replies true to the first Timedout or Commit, indicating no race, and false to any subsequent one.

```
Arbitrator(status) =
  checkRace.Init!false → Arbitrator(Init)
  □
  checkRace.Timedout!(status==Init) →
  Assert(status==Init or status==Commit)(Arbitrator(Timedout))
  □
  checkRace.Commit!(status==Init) →
  Assert(status==Init or status==Timedout)(
    if(status==Init) then Arbitrator(Commit) else Arbitrator(Timedout)
  )
```

The definition of a channel is very similar to as before, except in the initial state it may receive a command to close; it then sends a chanClosed message to each of the alts registered with it, and then sends a CLOSED response to any subsequent attempt to register.

```
Channel(me, reged) =
  ... -- as before
  □
  close.me → ChannelClosing(me, reged)

ChannelClosing(me, reged) =
  if reged=={} then ChannelClosed(me)
  else
    ⊓ pa'@@(port',a') : reged •
      chanClosed!port'.a' → ChannelClosing(me, remove(reged,pa'))

ChannelClosed(me) =
  register?a:calts(me)?port:inter(ports(me),aports(a)) →
  registerResp.port.a!CLOSED → ChannelClosed(me)
  □
  deregister?a?p:ports(me) → ChannelClosed(me)
```

### *4.2. Analysing the Design*

We have built configurations, using this extended compount alt, as in Figures 2 and 6. We have again used FDR to check that each refines a suitable specification, both including and excluding the possibility of timeouts and of channels closing. This gives us very strong confidence that the design is also correct in other configurations.

## 5. Code

In this section we outline the Scala implementation of the alt. We omit the full code for reasons of space constraints, and because most of it is a fairly straightforward implementation of the CSP model. We give the code for the MainAlt in the appendix. Most of the CSP events are implemented by a call to a procedure with the same name, or the return from a procedure.

The implementation needs to deal with a number of issues that we have ignored in the CSP models.

- In the CSP models, each alt registered with its channels in a *nondeterministic* order. Our analysis has, therefore, considered all possible orders for registering. The implementation supports two different orders: if the alt is a prialt, then the channels are registered in the order given in the program; otherwise, on the first execution of the alt, the channels are registered in the order given in the program, and on subsequent executions, they are registered starting from the one after the one registered last on the previous execution. (We discuss in the Conclusions how alts with different priorities interact.)
- The alt has to evaluate the guards. Our CSP analysis ignored guards; this abstraction is sound, as it is equivalent to the branches whose guards are false being filtered out prior to registration.

- The implementation of standard reading and writing on channels has to be adapted to interact with alts: if a thread attempts to do a standard read or write on a channel, the channel checks whether any alt registered at the other port is able to commit.
- The implementation also supports buffered channels. This part of the implementation is very similar to that for synchronous channels, except the OutPort will always reply YES to a register if the buffer is not full, and the InPort will always reply YES to a register if the buffer is not empty.

## *5.1. The Alt Class*

The structure of the Alt class is as in Figure 14. The class is parameterised by a sequence of Events, defining its branches. In the case of standard branches, the Event acts as a wrapper around the port, the guard and the command; the Events for timeout and orelse branches are similar; see the class diagram in Figure 15. The Alt class also has a boolean parameter priAlt indicating whether it is a prialt. It provides a default constructor, initialising to a non-prialt.

```
class Alt(events: Seq[Alt.Event], priAlt : Boolean){
  def this(events: Seq[Alt.Event]) = this(events, false)
  def apply(): Unit = MainAlt.apply();
  def repeat = CSO.repeat { this(); }
  private [cso] def commit(n:Int) : Int = Facet.commit(n);
  private [cso] def chanClosed(n:Int) = Facet.chanClosed(n);

  private object MainAlt extends Pausable{
    def apply(): Unit = synchronized {...}
    def wakeUp(n:Int) = synchronized {...}
    def allClosed = synchronized{...}
  }

  private object Facet {
    private var status = INIT;
    def commit(n:Int) : Int = synchronized{...}
    def chanClosed(n:Int) = synchronized{...}
    def changeStatus(s:Int) = synchronized {...}
    def setReged(nReged:Int) : Boolean = synchronized{...}
    def getToRun : Int = synchronized{...}
  }

  private object Arbitrator{
    def checkRace(s:Int) : Boolean = synchronized{...}
  }
}
```

**Figure 14.** The structure of the Alt class

At the top level the class provides two public procedures: apply, which corresponds to executing the alt, and which is implemented within the MainAlt; and repeat, which simply repeatedly executes the alt. It also has two procedures that are private to the CSO package and called by channels, commit and chanClosed; both correspond to the CSP events with the same name, and are implemented within the Facet.

Each Event has procedures

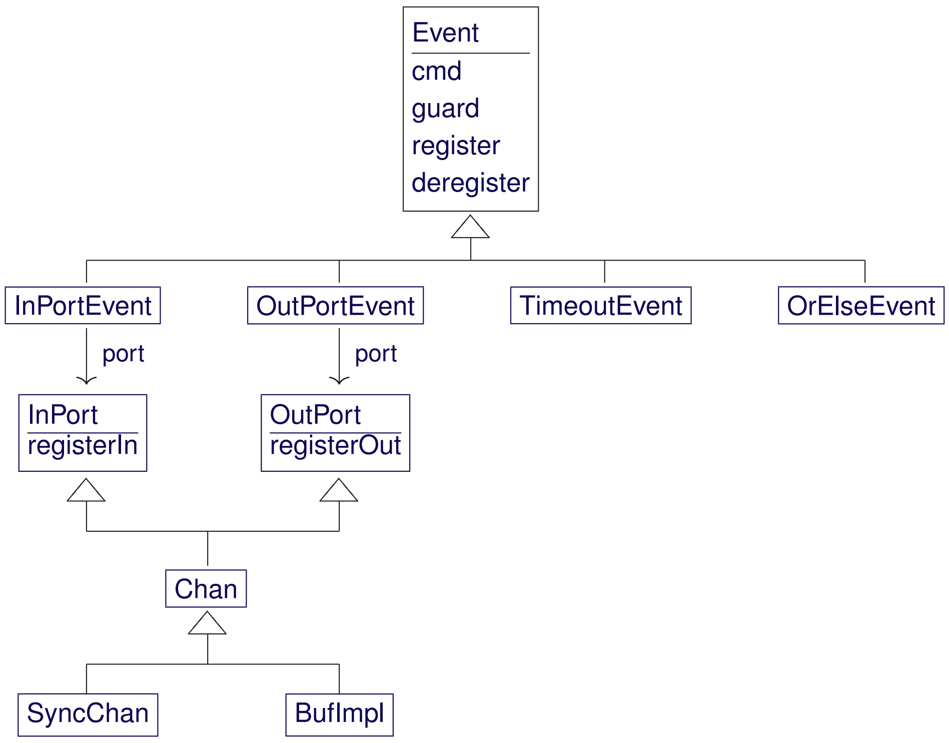

**Figure 15.** Class diagram for Events

```
def register(a:Alt, n:Int) : Int = {...}
def deregister(a:Alt, n:Int) = {...}
```

to allow an alt to register and deregister with the underlying channel. When the alt calls register, it passes the index n of the corresponding branch. For standard Events (i.e. excluding timeout and orelse Events), these calls are forwarded on to the appropriate channels. The index n is returned in subsequent calls to the commit and chanClosed procedures.

The Facet has a procedure changeStatus, which the MainAlt uses to inform it when it moves between the different stages of its operation: registering with channels, pausing, waiting, waiting with a timeout, deregistering with channels, and executing the appropriate branch; the status variable records the current status. We explain the other procedures of the Facet and Arbitrator below.

When MainAlt.apply is called, it calls register on each of its Events for which the guard is true (cf. state MainAltReg in the CSP model). If any call of register returns YES, that Event's command can be executed; see below. If none returns YES, and at least one returns MAYBE, then it enters the pausing phase.

In the pausing phase (cf. state MainAltPause), the MainAlt calls a procedure pause, within the Pausable trait. This performs a binary exponential back-off algorithm, inspired by the IEEE 802.3 Ethernet Protocol (see [8] or, e.g., [16]). The call to pause sleeps for a random amount of time. The maximum possible length of pause doubles on each call, to increase the chance of two alts in contention getting out of sync. (A final call to a procedure resetPause resets the maximum delay to a starting value of 1ns.)

After the call to pause completes, the MainAlt finds out from the Facet whether it has received any commit call in the meantime, by calling the getToRun procedure (this plays the role of both the getToRun and getToRunNo events; a negative result represents the latter); if that returns a positive result, the appropriate Event's command can be executed; see below. Otherwise, the MainAlt again tries to register with the remaining channels.

If and when the MainAlt receives a reply of NO from each of its standard Events, it needs to wait for one to become ready. Consider, first, the case where there is no timeout Event (cf. state MainAltWait). The MainAlt first calls Facet.setReged, passing in the number of registered Events. A boolean is returned that indicates whether all the channels have closed in the meantime. If so, then no Event is enabled, so the alt executes the orelse branch if there is one, or throws an Abort exception. If at least one Event is enabled, the MainAlt sets a boolean flag waiting, and executes

```
while(waiting) wait()
```

waiting for one of two things to happen.

- If the Facet receives a call to commit, it wakes up the MainAlt by calling the following procedure, passing in the index of the Event:

  ```
  def wakeUp(n:Int) = synchronized {
    assert(waiting); toRun = n; waiting = false; notify();
  }
  ```

- If the Facet finds that all the channels have been closed, it wakes up the MainAlt by calling the following procedure;

  ```
  def allClosed = synchronized{
    assert(waiting); allBranchesClosed = true; waiting = false; notify();
  }
  ```

When the MainAlt is awoken, it checks the value of allBranchesClosed; if it's true (so allClosed was called), it executes the orelse branch if there is one, or throws an Abort exception; otherwise (so wakeUp was called) it can run the process indicated by toRun.

If there is a timeout branch, then things proceed much as above, except the MainAlt performs a timed **wait**, and the Facet will not call allClosed. When the MainAlt wakes up, it can tell, by inspecting the waiting variable, whether the timeout was reached, or if it was awoken by the Facet calling wakeUp. In the latter case, things proceed as above. If the timeout was reached, then the MainAlt checks whether there was a race with a received commit by calling Arbitrator.checkRace; if there was no race, then it proceeds as above; if there was a race, it performs another (untimed) **wait**, waiting to be woken by the Facet. (Likewise, if the Facet receives a commit while the MainAlt is waiting, it checks whether there was a race with a timeout by calling Arbitrator.checkRace.)

Finally, once the Event to execute has been selected, the MainAlt calls deregister on each of the other registered Events, and executes the command of the selected Event.

### *5.2. Testing*

We have tested the implementation on configurations corresponding to Figures 2 and 6. The implementation seems robust and efficient. For example, the configuration in Figure 2 achieves over 40,000 communications per second (over 20,000 on each channel) on a standard quad-core PC. We have also used it in our Concurrent Programming course.

The binary back-off algorithm seems to work well, giving rather small delays. We performed informal testing of a simple configuration based on Figure 2; in this configuration, the two alts are likely to register with channels at the same time, leading to more pausing than in most other configurations. On average, only about 0.1% of the total time was spent in the pause procedure.

### *5.3. Restrictions*

The current implementation is subject to the following restrictions on usage:

1. If a shared port (e.g. of a ManyMany channel, whose ports can be shared by several senders and by several receivers) is involved in an alt, it must not simultaneously be read or written by a non-alt process;
2. An alt may not have two simultaneously enabled branches using the same channel (although it may have two branches using the same channel with disjoint guards);
3. The current implementation does not cover network channels.

The reason for the first restriction is that —as mentioned in the introduction— the alt itself does not perform a read or write: the body of the selected branch is responsible for this. Consider, for example, P || Q || R where

```
def P = proc{ alt ( c −?−> { val x = c?; ... } |  ...  ) }
def Q = proc{ val y = c?;  ...  }
def R = proc{ c!3;  ...  }
```

There is a danger that the alt in P selects its first branch, but then the read in that branch is pre-empted by the read in Q. It seems impossible to avoid this, without the alt performing the read or write.

The reason for the second restriction is that without it the system can livelock in certain circumstances. Consider, for example, P || Q where

```
def P = proc{ alt ( c −?−> { ... } | c −?−> { ... } ) }
def Q = proc{ c!3;  ...  }
```

Suppose P has already registered with c corresponding to the first branch. Now suppose Q executes c!3, and so locks c. At this point, P tries to register with c corresponding to the second branch, but is blocked, because it cannot get the lock on c. Meanwhile, within Q, c will repeatedly call P's commit method, but repeatedly get a response of MAYBE, because P has not finished registering. The system is stuck, since Q's write on c will never complete, so P will never get the lock on c to finish registering.

We believe the extension to network channels would be reasonably straightforward: essentially the same design can be used, although with register and commit messages being sent across the network.

## 6. Conclusions

In this paper we have described the development of a generalised alt operator, using CSP models and FDR analysis to develop a correct design. We consider the use of CSP and FDR to have been invaluable in this work: we believe we would not have ended up with a working implementation without such an approach.

Recall that we initially (Section 2) used a sequential model of an alt, and then (Section 3) developed this into a parallel model, corresponding to the composition of several objects. We consider this approach to have been useful: first considering the *inter*-component protocol, and then considering the *intra*-component protocol. (In fact, some of the latter stage proved inconsistent with the former, so we subsequently revised the models from the former stage, so as to tell a coherent story in this paper.) The next development steps were to add the features of channels closing and timeouts (Section 4); separating these features from the main line of development helped to clarify the ideas.

The final step was to to refine the models to code (Section 5) and test; as noted in Section 5.2, the implementation gives very acceptable performance. Despite the formal development, this step was not completely straightforward. Besides the inevitable small errors, one issue proved difficult to resolve. A feature of the Scala implementation is that when each object of a class has an inner object —like the Facet inside each Alt— the inner object is created at the point of first use. Unfortunately, if the object is multi-threaded, it is possible for two threads to create this object at almost the same time, meaning the inner object is not unique! An earlier implementation of Alt fell foul of this, so that sometimes (once every few million iterations) an Alt ended up with two Facets, leading to incorrect behaviour. This problem is now avoided by MainAlt.apply initialising the Facet (and Arbitrator) initially, before any other thread can call a procedure.

## *6.1. CSP and Monitors*

One of the goals of this paper was to investigate the relationship between CSP processes and implementations using monitors; we discuss this here. Recall that our aim is to avoid using internal channels (or anything that is channel-like, or has the same overheads in terms of context switches). Some may object to this restriction, and argue for using channels; we would agree with this point of view in many applications; but sometimes it is necessary to code at a lower level of abstraction in order to achieve greater efficiency.

A simple monitor that does not perform **wait** and has procedures $f_1, \ldots, f_n$ corresponds to a process with shape

```
Proc(state) =
  f₁?arg₁ → ... → fResp₁!resp₁ → Proc(state₁)
  □ ...
  □ fₙ?argₙ → ... → fRespₙ!respₙ → Proc(stateₙ)
```

Here $arg_1, \ldots, arg_n$ correspond to the arguments of the procedures, and $resp_1, \ldots, resp_n$ correspond to the returned values; state captures the state of the monitor, and $state_1, \ldots, state_n$ capture how the state is updated. The elided part mostly corresponds to calls to and returns from other monitors, the call and return events being consecutive; the elided part may also include signal events that are used within the specification for the FDR analysis.

Some CSP processes cannot be written in the above form, but can still be implemented as a monitor by using **wait**. For example, suppose we have a CSP process with a state of the following form, as an intermediate state within the part of the process corresponding to a procedure call.

```
e₁?x₁ → P₁ □ ... □ eₙ?xₙ → Pₙ
```

This process is waiting to receive a message from another process. The process can often be implemented by the code

```
waiting = true; while(waiting) wait(); // wait to be woken up
wakeUpType match { // which wake−up event happened?
  case 1 => ... // code for P₁
  ...
  case n => ... // code for Pₙ
}
```

and by providing procedures of the following form, for k = 1,...,n (corresponding to events of the form $e_k$!arg in the other process).

```
def eₖ(arg : Tₖ) = synchronized{
  assert(waiting); wakeUpType = k; xₖ = arg; // pass data
  waiting = false; notify();                  // wake up waiting process
}
```

Here waiting, wakeUpType and $x_k$ ($k = 1,...,n$) are private variables of the monitor. In order for this to work, we need to ensure that other processes try to perform one of $e_1,\ldots,e_n$ only when this process is in this waiting state. Further, we need to be sure that no other process calls one of the main procedures $f_1,\ldots,f_n$ while this process is in this state. We can test for both of these requirements within our CSP models.

The restrictions in the previous paragraph prevent many processes from being directly implemented as monitors. In such cases we believe that we can often follow the pattern corresponding to the use of the Facet: having one monitor that performs most of the functionality, and a second monitor (like the Facet) that keeps track of the state of the main monitor, receives procedure calls, and passes data on to the main monitor where appropriate. In some such cases, it will also be necessary to follow the pattern corresponding to the use of the Arbitrator, to arbitrate in the case of race conditions.

We leave further investigation of the relationship between CSP and monitors for future work.

## *6.2. Priorities*

An interesting question concerns the behaviour of a system built as the parallel composition of prialts with differing priorities, such as P || Q where:

```
def P = proc{ prialt ( c1 -!-> { c1!1; } | c2 -!-> { c2!2; } ) }
def Q = proc{ prialt ( c2 -?-> { println(c2?); } | c1 -?-> { println(c1?); } ) }
```

It is clear to us that such a system should be able to communicate on either c1 or c2, since both components are; but we should be happy whichever way the choice between the channels is resolved.

Consider the implementation in this paper. Suppose that P runs first, and registers with both of its channels before Q runs; then when Q tries to register with c2, it will receive a response of YES, so that branch will run: in other words, Q's priority will be followed. Similarly, if Q runs first, then P's priority will be followed. If both run at the same time, so they both receive a response of MAYBE to their second registration attempt, then they will both pause; which channel is chosen depends upon the relative length of their pauses.

## *6.3. Future Plans*

Finally, we have plans for developing the implementation of alts further. We would like to change the semantics of alt, so that the alt operator is responsible for performing the read or write of the branch it selects. This will remove the first restriction discussed at the end of Section 5. (This would also remove a possible source of bugs, where the programmer forgets to read or write the channel in question.) This would not change the basic protocol described in this paper.

A barrier synchronisation [10] allows n processes to synchronise together, for arbitrary n. It would be useful to extend alts to allow branches to be guarded by barrier synchronisations, as is allowed in JCSP [17].

## *Acknowledgements*

We would like to thank Bernard Sufrin for implementing CSO and so interesting us in the subject, and also for numerous discussions involving the intended semantics for alts. We would also like to thank the anonymous referees for a number of useful comments and suggestions.

## References

[1] Neil Brown. Communicating Haskell Processes: Composable explicit concurrency using monads. In *Communicating Process Architectures (CPA 2008)*, pages 67–83, 2008.

[2] Neil Brown. Choice over events using STM. `http://chplib.wordpress.com/2010/03/04/choice-over-events-using-stm/`, 2010.

[3] Neil Brown. Conjoined events. In *Proceedings of the Advances in Message Passing Workshop*, 2010. `http://twistedsquare.com/Conjoined.pdf`.

[4] N. Carriero, D. Gelernter, and J. Leichter. Distributed data structures in Linda. In *Proc. Thirteenth ACM Symposium on Principles of Programming Languages*, pages 236–242, 1986.

[5] Formal Systems (Europe) Ltd. *Failures-Divergence Refinement—FDR 2 User Manual*, 1997. Available via URL `http://www.formal.demon.co.uk/FDR2.html`.

[6] Tim Harris, Simon Marlow, Simon Peyton Jones, and Maurice Herlihy. Composable memory transactions. In *PPoPP '05*, pages 48–60, 2005.

[7] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[8] IEEE 802.3 Ethernet Working Group website, `http://www.ieee802.org/3/`.

[9] INMOS Ltd. *The occam Programming Language*. Prentice Hall, 1984.

[10] H. F. Jordan. A special purpose architecture for finite element analysis. In *Proc. 1978 Int. Conf. on Parallel Processing*, pages 263–6, 1978.

[11] Alistair A. McEwan. *Concurrent Program Development*. DPhil, Oxford University, 2006.

[12] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima Press, 2008.

[13] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

[14] Bernard Sufrin. Communicating Scala Objects. In *Proceedings of Communicating Process Architectures (CPA 2008)*, 2008.

[15] Bernard Sufrin. CSO API documentation. `http://users.comlab.ox.ac.uk/bernard.sufrin/CSO/doc/`, 2010.

[16] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall, 1996.

[17] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and extending JCSP. In *Communicating Process Architectures (CPA 2007)*, 2007.

[18] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting barriers: synchronisation with choice in Java using CSP. *Concurrency and Computation: Practice and Experience*, 22:1049–1062, 2010.

## A. Code Listing

We give here the code for the MainAlt.

```
private object MainAlt extends Pausable{
  private var waiting = false ; // flag to indicate the alt is waiting
  private var toRun = −1; // branch that should be run
  private var allBranchesClosed = false; // are all branches closed?
  private var n = 0; // index of current event

  /* Execute the alt */
  def apply (): Unit = synchronized {
    Facet.changeStatus(INIT); Arbitrator.checkRace(INIT);

    var enabled = new Array[Boolean](eventCount); // values of guards
    var reged = new Array[Boolean](eventCount); // is event registered ?
    var nReged = 0;                             // number of registered events

    var done = false; // Have we registered all ports or found a match?
    var success = false; // Have we found a match?
    var maybes = false; // have we received a MAYBE?
    var timeoutMS : Long = 0; // delay for timeout
    var timeoutBranch = −1; // index of timeout branch
    var orElseBranch = −1; // index of orelse branch
     if ( priAlt ) n=0;
    toRun = −1; allBranchesClosed = false;
```

```
// Evaluate guards; this must happen before registering with channels
for(i <- 0 until eventCount) enabled(i) = events(i).guard();

while(!done){
  var count=0; // number of events considered so far
  while(count<eventCount && !done){
    if (!reged(n)){ // if event(n) not already registered
      val event = events(n);
      if (enabled(n)){
        event match {
          case Alt.TimeoutEvent(tf, _) =>
            if (timeoutBranch>=0 || orElseBranch>=0)
              throw new RuntimeException("Multiple timeout/orelse branches in alt");
            else{ timeoutMS = tf(); timeoutBranch = n; reged(n) = true; }
          case Alt.OrElseEvent(_,_) =>
            if (timeoutBranch>=0 || orElseBranch>=0)
              throw new RuntimeException("Multiple timeout/orelse branches in alt");
            else{ orElseBranch = n; reged(n) = true; }
          case _ => { // InPortEvent or OutPortEvent
            event.register(theAlt,n) match{
               case YES => { Facet.changeStatus(DEREG); toRun = n; done=true; success=true; }
               case NO => { reged(n) = true; nReged += 1; }
               case MAYBE => maybes = true;
               case CLOSED => enabled(n) = false; // channel has just closed
            } // end of event.register(theAlt,n) match
          } // end of case _
        } // end of event match
      } // end of if(enabled(n))
    } // end of if(!reged(n))

    n = (n+1)%eventCount; count += 1;
  } // end of inner while

  if (!done) // All registered, without finding a match
    if (maybes){
      // Random length pause to break symmetry
      Facet.changeStatus(PAUSE); pause;
      // see if a commit has come in
      toRun = Facet.getToRun;
      if (toRun<0) maybes = false; // No, so reset variables for next round
      else{ done = true; success = true; } // done
    } // end of if(maybes)
    else done=true;
} // end of outer while
resetPause;

// All events now registered with their channels

if (!success){ // No registration returned YES
  if (timeoutMS==0){ // no timeout
    if (nReged==0) // no event enabled
      if (orElseBranch>=0) toRun = orElseBranch else throw new Abort;
    // Need to wait for a channel to become ready
    waiting=true; allBranchesClosed = Facet.setReged(nReged);
    if (!allBranchesClosed) while(waiting) wait(); // wait to be awoken
  }
  else{ // with timeout
    Facet.changeStatus(WAITTO); waiting=true;
    wait(timeoutMS); // wait to be awoken or for timeout
    if (waiting){
      // assume timeout was reached (this could be a spurious wakeup)
      if (Arbitrator.checkRace(TIMEDOUT)){ waiting = false; toRun = timeoutBranch; }
      else // A commit was received just before the timeout.
        while(waiting) wait()   // Wait to be woken
        } // end of if(waiting)
      } // end of else (with timeout)
    } // end of if(! success)

    // Can now run branch toRun, unless allBranchesClosed
    if (allBranchesClosed)
      if (orElseBranch>=0) toRun = orElseBranch else throw new Abort;
    // Deregister events
    Facet.changeStatus(DEREG);
    for(n <- 0 until eventCount) if(n != toRun && reged(n)) events(n).deregister(theAlt,n)

    // Finally, run the selected branch
    Facet.changeStatus(DONE); events(toRun).cmd();
  } // end of apply

  /* Implementation of WakeUp events; called by Facet to wake up the MainAlt */
  def wakeUp(n:Int) = synchronized {
    assert(waiting); // use of Arbitrator should ensure this
    toRun = n; waiting = false; notify(); // wake up MainAlt
  }

  /* Receive notification from Facet that all branches have been closed */
  def allClosed = synchronized{ assert(waiting); allBranchesClosed=true; waiting=false; notify(); }
}
```