Implementing Generalised Alt

Gavin Lowe

CSO for dummies

Communicating Scala Objects (CSO) is a library of CSP-like communication primitives for the Scala programming language, implemented by Bernard Sufrin.

Here's a simple example:

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

Alternation

CSO —inspired by occam— includes a construct, alt, to provide a choice between communicating on different channels. Here's a simple example

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

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.

In the original version of CSO alts could perform selections only between input ports (InPorts). Later this was extended to include output ports (OutPorts), for example:

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

Alternation

However, the implementation of alt had the following restriction:

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

This restriction makes the implementation of alts considerably easier. But it can be inconvenient in a number of settings. Our aim is to remove this restriction.

We are aiming for an implementation in terms of monitors, avoiding using channels internally, or a centralised controller.

Using CSP

Our development strategy was to build CSP models of putative designs, and then to analyse them using FDR. 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, whereas counterexamples found by testing are likely to be much longer;
- CSP models are more abstract and so easier to understand than code.

Overview

- An incorrect design;
- A correct design — but that can't be implemented directly by a monitor;
- A compound design;
- Adding timeouts and channels closing;
- Code.

First design

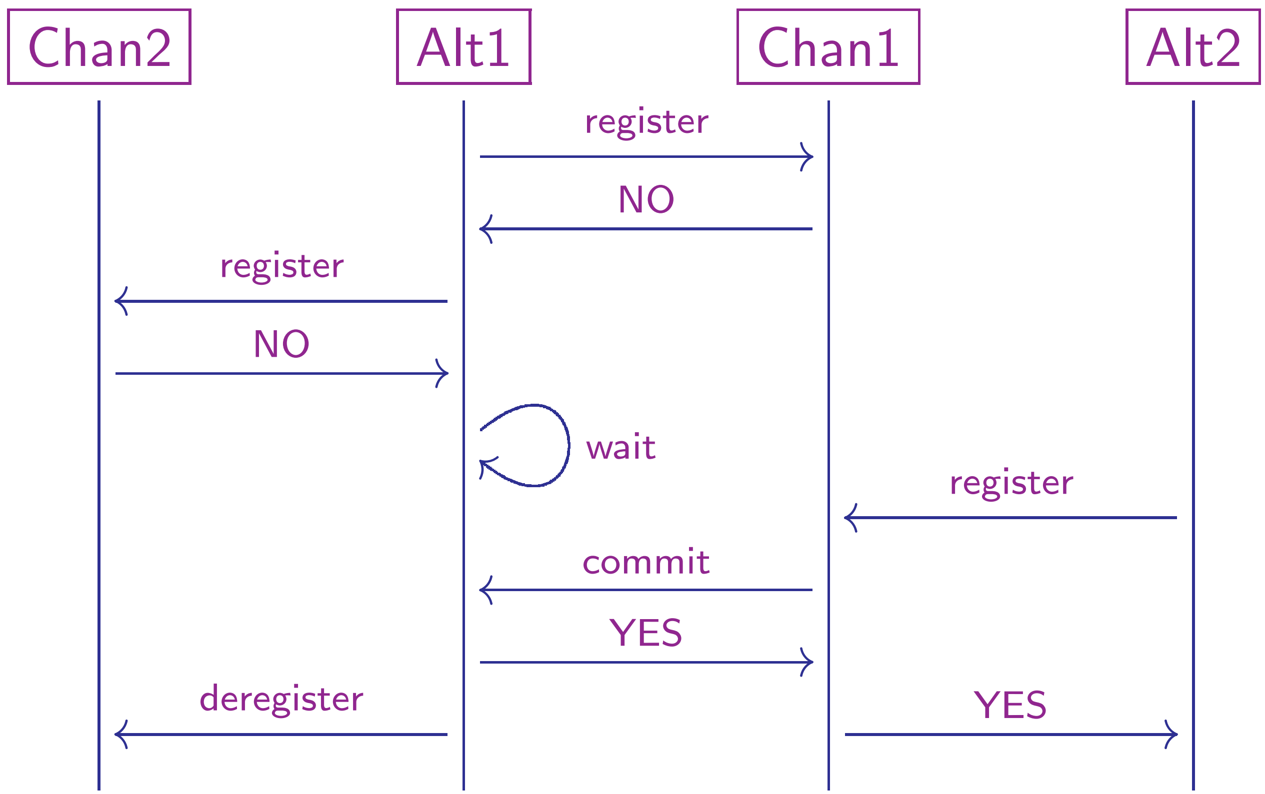
An alt registers, in turn, with each of its channels.

- If the channel is immediately ready to communicate, it returns a response of YES, and the communication goes ahead;
- Otherwise, the channel returns a response of NO.

If the alt receives a response of NO from each of its channels, it waits for one to become ready.

If the channel subsequently becomes ready to communicate, it sends a message to the alt asking if it can commit.

First design



CSP model of an alt

```
-- Alt with identity me and ports ps
Alt(me,ps) = AltReg(me, ps, {}, ps)

-- Register with the ports in toReg
AltReg(me, ps, reged, toReg) =
  if toReg=={} then AltWait(me, ps, reged)
  else
    ⊓ p : toReg •
        register.me.p → registerResp?p!me?resp →
        if resp==YES then AltDereg(me, ps, reged, p)
        else AltReg(me, ps, add(reged,p), remove(toReg,p))
```

CSP model of an alt

```
-- Wait for a port to become ready
AltWait(me, ps, reged) =
  commit?p:reged!me → commitResp.me.p!YES →
  AltDereg(me, ps, remove(reged,p), p)

-- Deregister from the ports in toDereg
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:ps → commitResp.me.p1!NO → AltDereg(me, ps, toDereg, p)
  )
```

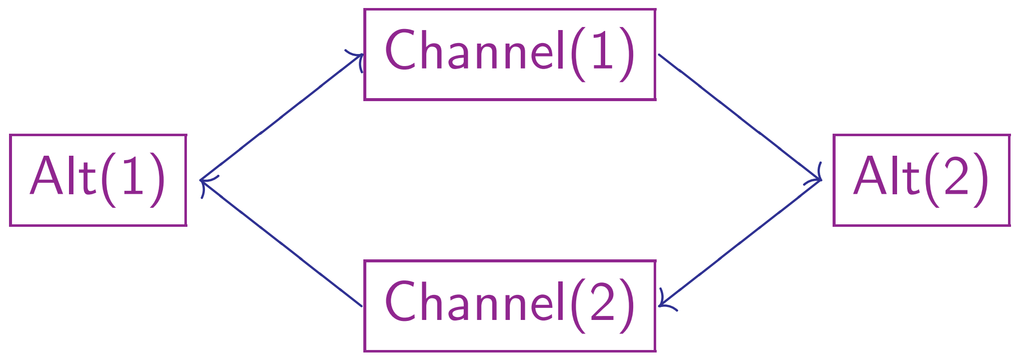
CSP model of a channel

```
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)))

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'))
```

Testing with FDR

We can use FDR to test whether this configuration:

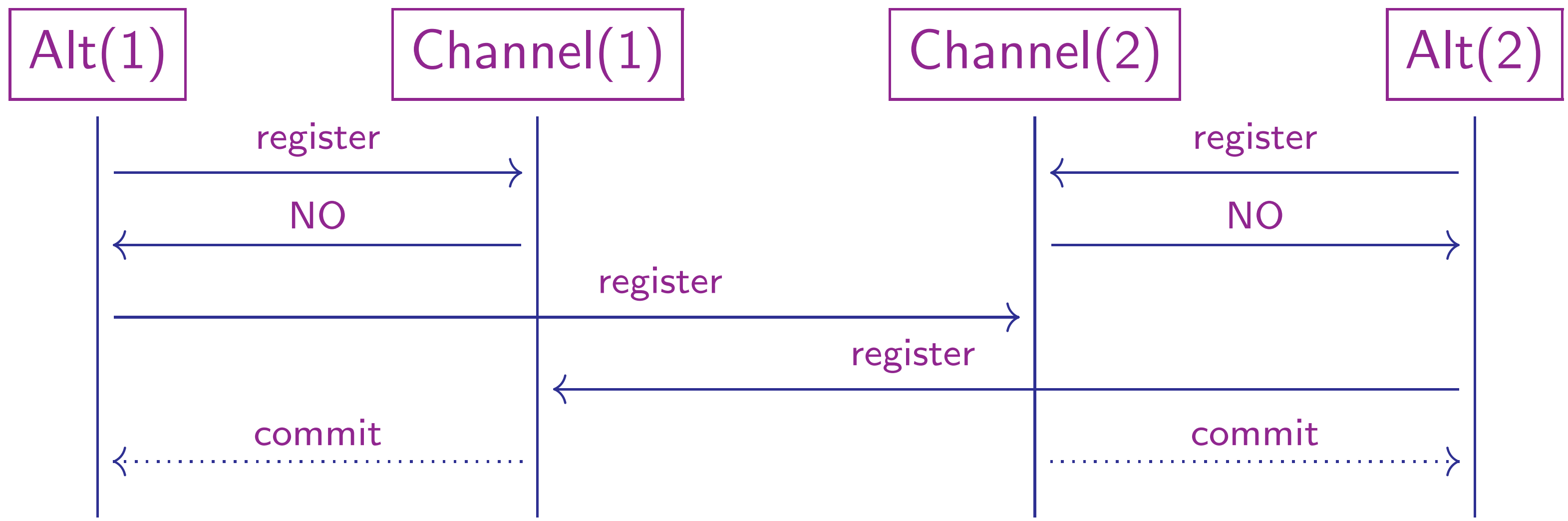


with all events other than signals hidden, refines the following specification in the stable failures model:

$$\begin{aligned}
&\mathsf{Spec} = \\
&\quad \sqcap\ \mathsf{c : ChannelId} \bullet \\
&\qquad \mathsf{signal.1.c} \rightarrow \mathsf{signal.2.c} \rightarrow \mathsf{Spec} \\
&\qquad \Box\ \mathsf{signal.2.c} \rightarrow \mathsf{signal.1.c} \rightarrow \mathsf{Spec}
\end{aligned}$$

Deadlock

FDR finds the following behaviour leads to deadlock.



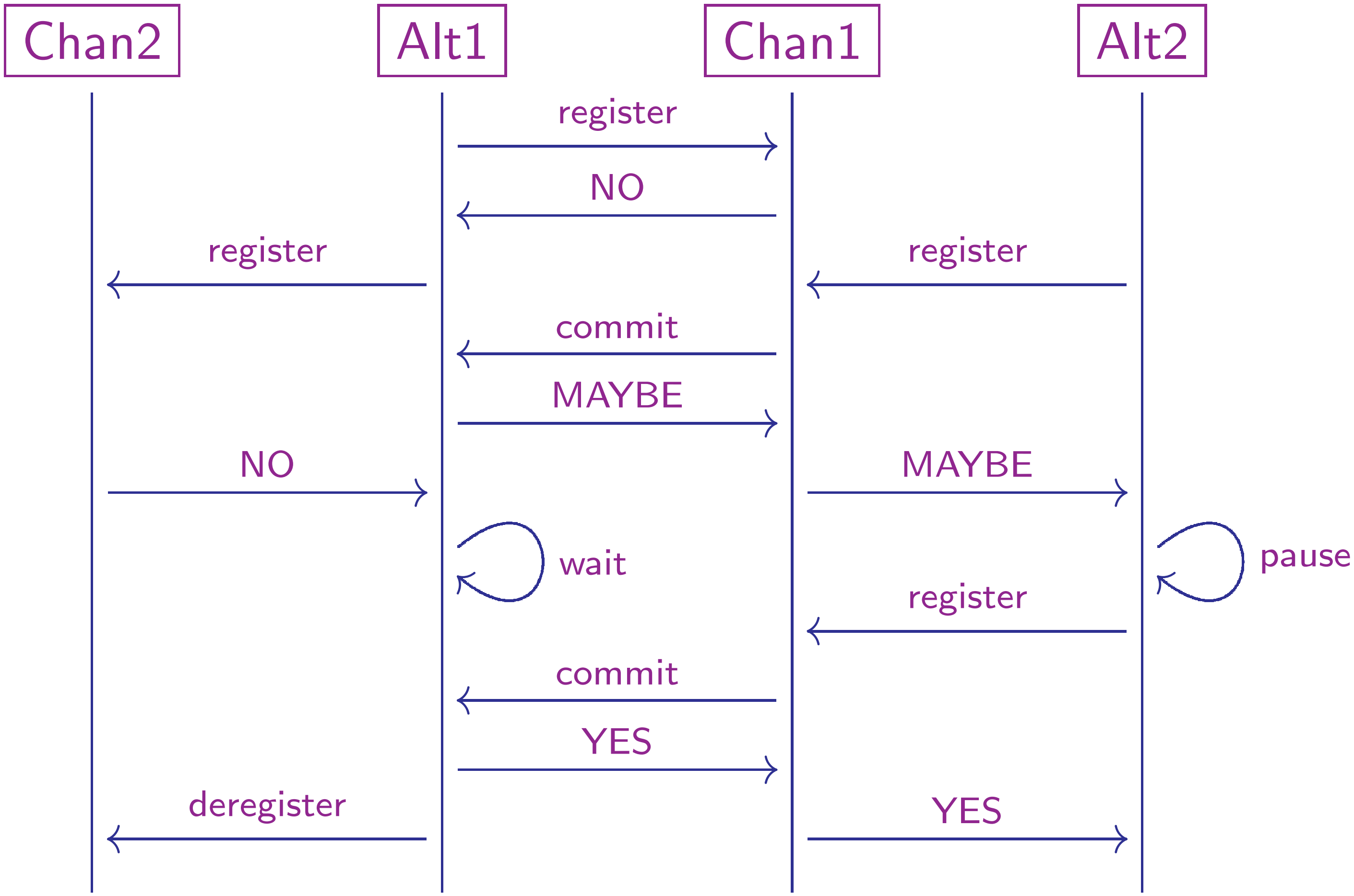
Improved design

The counterexample 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 (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.
- Delaying replying to the commit until after a response to the register has been received would again lead to a deadlock.

We therefore introduce a different response, MAYBE, that an alt can send in response to a commit; informally, MAYBE means “I'm busy right now; please call back later”.

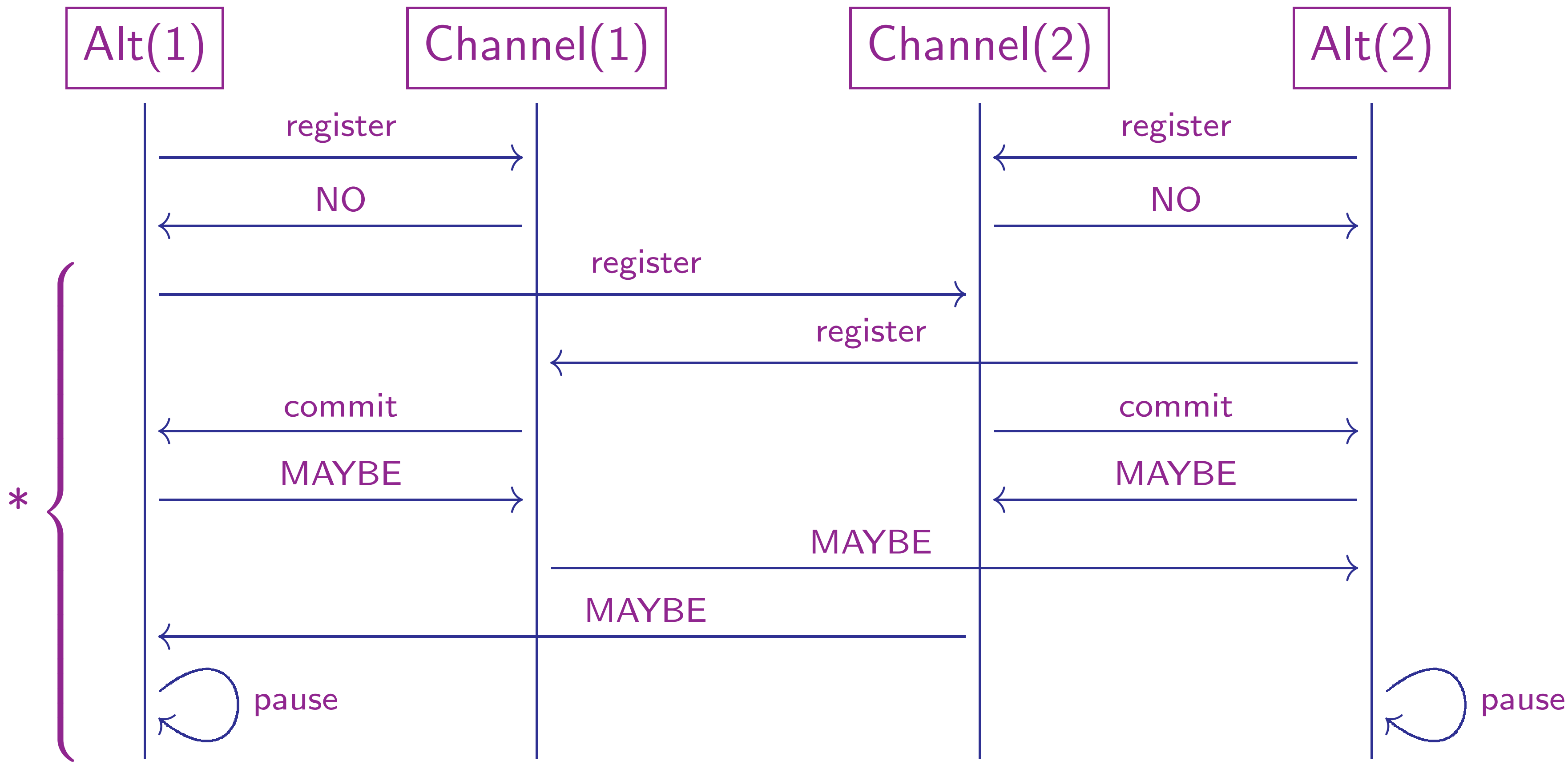
Using MAYBE



We can adapt the CSP model to capture this new protocol. (See paper.)

Analysing the new design

FDR finds that the system satisfies the earlier stable failures refinement, but that it can diverge.



Analysing the new design

We can perform a different refinement test to find that the only way that the system can diverge is through repeated pauses and retries.

In the implementation, the pause will be of a random amount of time, to ensure the symmetry is eventually broken (with probability 1).

I've analysed various other configurations, and got appropriate results.

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.

Compound alts

The previous model captures the desired behaviour of an alt. However, it does not seem possible to implement this behaviour using a single monitor, with messages implemented by procedure calls and their returns.

We want to:

- implement the main execution of the alt as a procedure apply, and
- implement the commit and commitResp events as a procedure commit and its return.

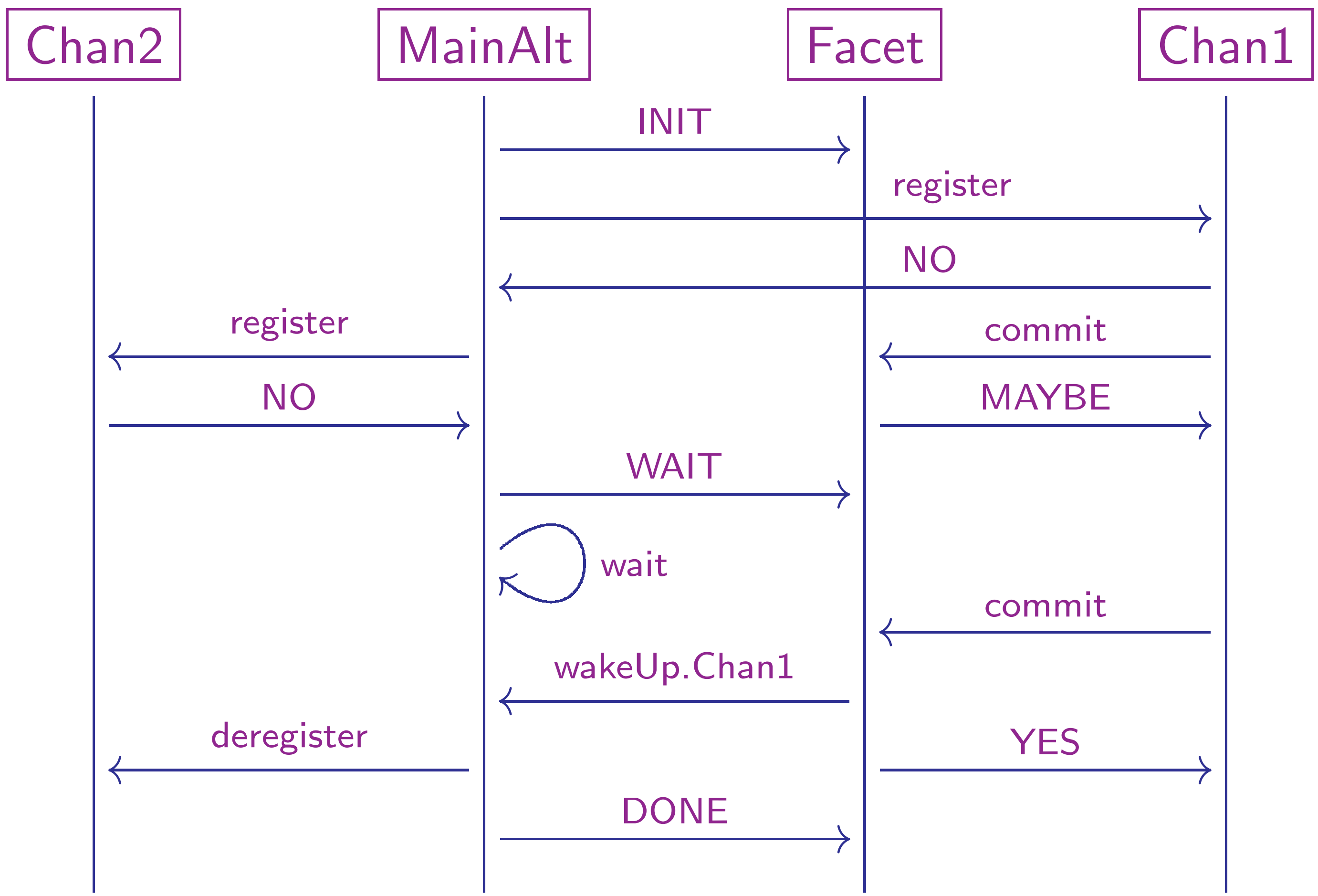
However, these two procedures will need to be able to run concurrently, so cannot be implemented in a single monitor.

Compound alts

Instead we implement the alt using two monitors.

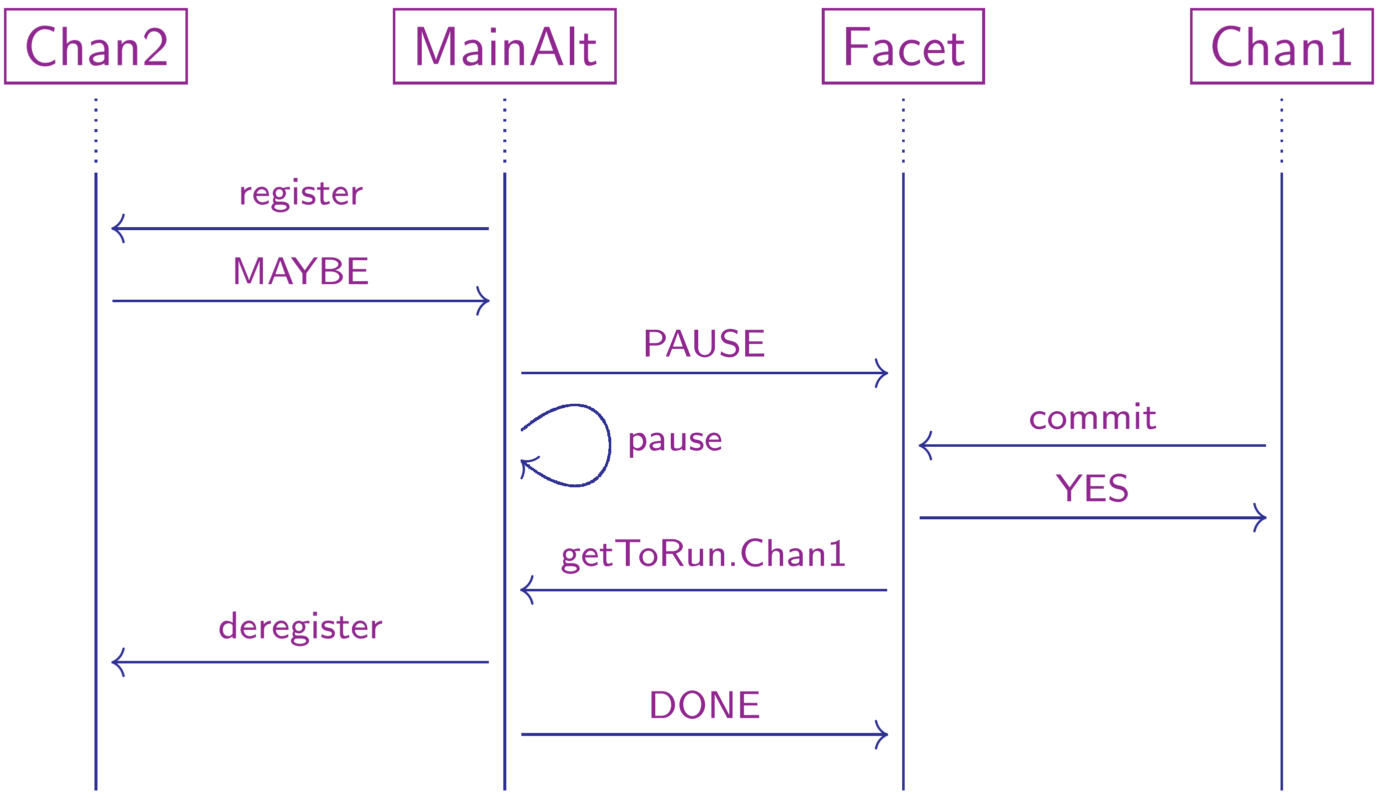
- The MainAlt will implement the apply procedure, to register with the channels, deregister at the end, execute the appropriate branch of the alt, and generally control the execution.
- The Facet will provide the commit procedure, responding appropriately. It will receive messages from the MainAlt, informing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

Compound alts



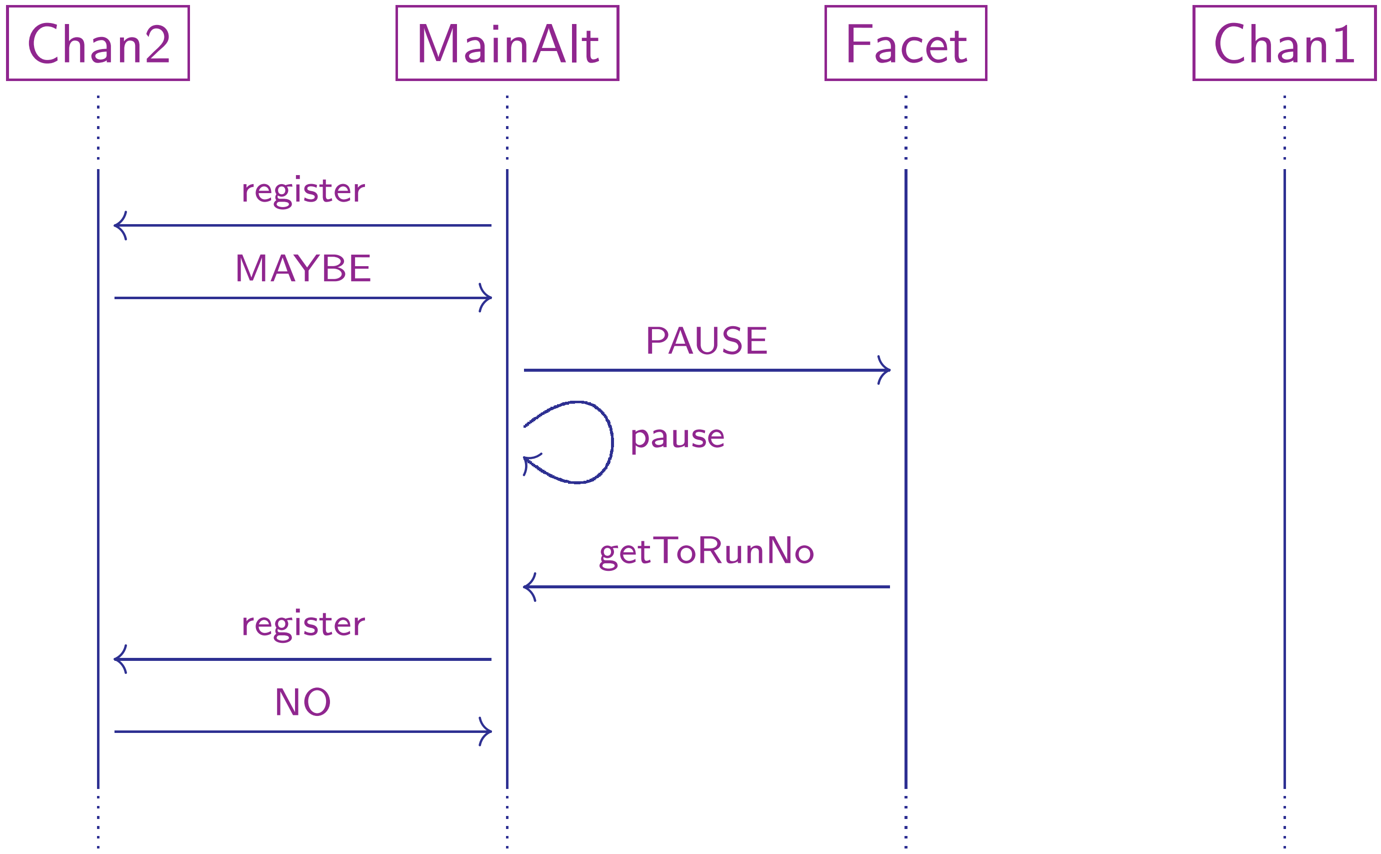
A commit received while pausing

Recall that if the MainAlt receives a reply of MAYBE when trying to register with channels, it pauses for a short while, before retrying. Here's what happens if a commit is received during the pause.



Pausing before retrying

And here's what happens if no commit is received during the pause.



Analysing the compound design

We can build CSP models for this compound design: each alt is formed as a parallel composition of MainAlt and Facet processes.

I have tested various configurations built from compound alts.

Adding timeouts

Alts may have timeout branches, for example

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

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, and respond YES, as before.
- If the timeout time is reached, the alt can run the timeout branch.

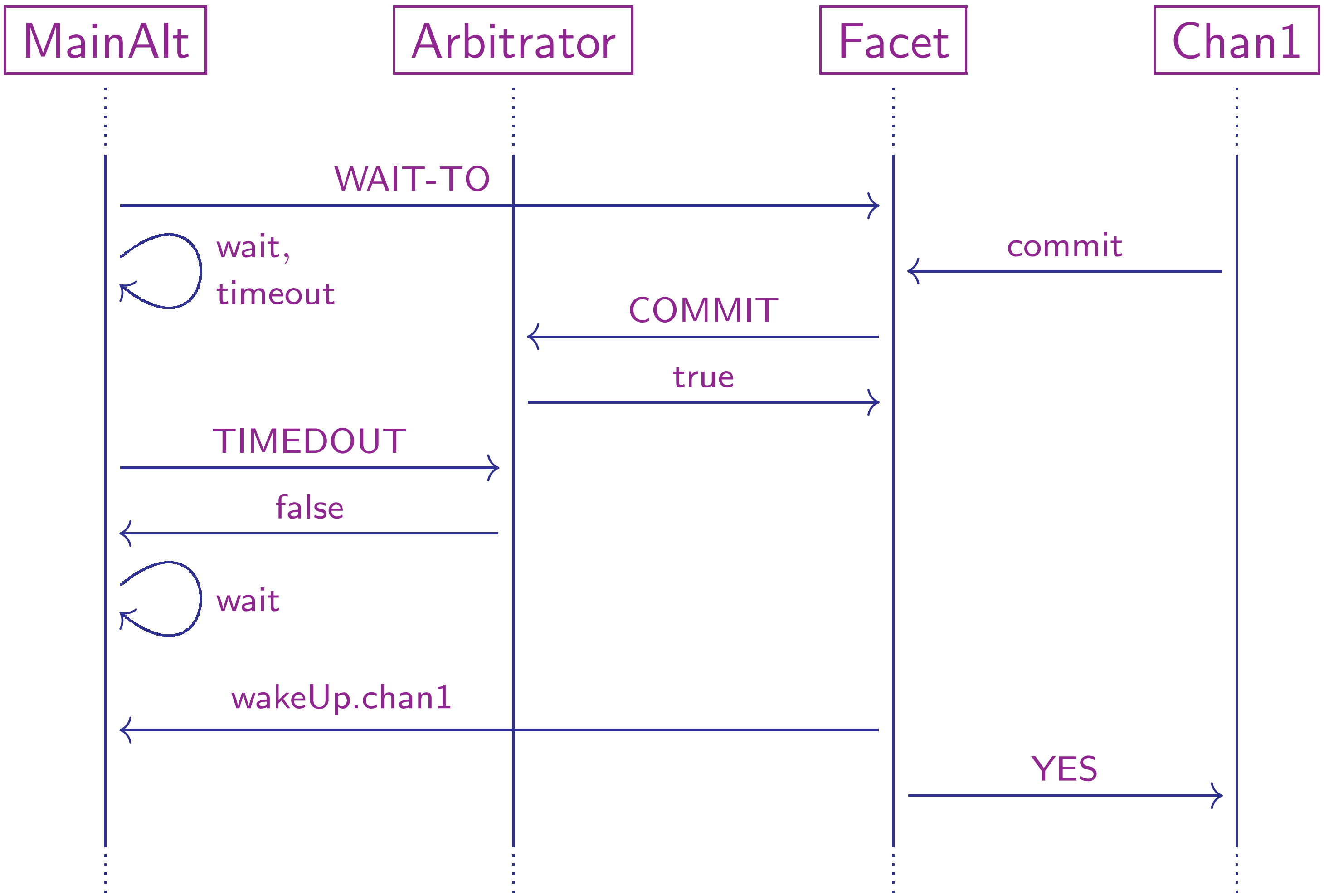
Adding timeouts

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.

When the Facet receives a commit, it contacts the Arbitrator to see if there was a race. Likewise, when a timeout is reached, the MainAlt contacts the Arbitrator to see if there was a race. Whichever component calls the Arbitrator first "wins" the race.

A commit beating a timeout in a race



The case of a timeout beating the commit is similar.

Closing channels

Channels may be closed. If all of an alt's branches are disabled (i.e., for each, the guard is false or the channel is closed), then it throws an Abort exception.

However, if there is an orelse branch, e.g.

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

and all other branches are disabled, then the orelse branch is executed.

Closing channels

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 closed.

If the Facet receives sufficient chanClosed messages such that all channels are closed, it wakes up the MainAlt by sending it an allClosed message.

We can extend the CSP models to include both timeouts and the closing of channels.

Code overview

```
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{...}
  }

   ...
}
```

Code overview

```
class Alt(events: Seq[Alt.Event], priAlt : Boolean){
  ...

  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{...}
  }
}
```

Implementation

Most of the implementation is a straightforward translation of the CSP model.

Recall that if the MainAlt receives a response of MAYBE (and no response of YES), it pauses before retrying.

In the implementation, the MainAlt calls a procedure pause. This performs a binary exponential back-off algorithm, inspired by the IEEE 802.3 Ethernet Protocol. The call to pause sleeps for a random amount of time. The maximum possible length of pause doubles on each successive call, to increase the chance of two alts in contention getting out of sync.

Implementing waiting

Recall that if the MainAlt receives a reply of NO from each of its channels, it waits for one to become ready, or for all the channels to be closed; each of these is signalled by a message from the Facet. In the CSP model:

```
  wakeUp?p:reged → MainAltDereg(me, ps, remove(reged,p), p)
□ allClosed → MainAltAllClosed(me, ps, reged)
```

Implementing waiting

In the implementation, the MainAlt sets a boolean flag waiting, and executes

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

The Facet wakes up the MainAlt by calling one of the following procedures (in MainAlt).

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

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

When the MainAlt wakes up, it can determine which procedure was called by testing allBranchesClosed.

Conclusions

- We've built a useful component for message-passing concurrency. The implementation seems fast. It has survived beta-testing by students.
- Building CSP models, and performing analysis using FDR, can help with developing working code.
- What CSP processes can be implemented directly as monitors?
- Can we automate the translation from CSP to code?
- What design patterns can we use to aid such a development?