Analysing a Library of Concurrency Primitives using CSP

Gavin Lowe

June 19, 2023

Abstract

...

1 Introduction

Scala Concurrency Library (SCL) is a library of concurrency primitives for the Scala programming language. It was developed for teaching concurrent programming to students, and includes support for message-passing concurrency, monitors and semaphores. In this paper we analyse the message-passing primitives: we build CSP models of them, and then use the model checker FDR to test for correctness. To our surprise, the analysis revealed a bug in the implementation.

We start by describing relevant aspects of SCL. Program threads can send and receive messages using *channels*. If c is a channel, then the command c!x sends the value x on c; the expression c?() receives and returns a value. We consider just *synchronous* channels in this paper: the sending thread much wait until there is a thread willing to receive. Channels are typed: the type SyncChan[A] represents synchronous channels that send data of type A. A channel is composed of an *outport*, where values are sent, and an *inport*, where values are received.

Channels also have timed operations. The operation sendWithin(delay)(x) attempts to send x, but if it is unable to synchronise with a receiving thread within delay ms, it times out; it returns a boolean to indicate whether sending was successful. Similarly, the operation receiveWithin(delay) attempts to receive, but if it is unable to synchronise with a sending thread within delay ms, it times out; it returns a value Some(x) to indicate that it successfully received x, or None to indicate that it timed out.

Ports may be shared: multiple threads may try to send or receive on the same port concurrently; the channel is responsible for pairing off a sender with a receiver.

A channel can be closed (by the close operation): subsequently, an attempt to send or receive on the channel will throw a Closed exception.

An *alt* allows a thread to communicate on one of several channels, whichever is available for communication first. The following example illustrates the usage.

```
alt(
  in =?=> { x => println(x) }
  | out =!=> { 42 } ==> { println("42 sent") }
)
```

An alt consists of a number of branches, separated by "|". An *inport branch* is denoted "in =?=> f" where in is a channel (or inport), and f is a function that takes an argument of the type of in; we call f a *continuation* (above, "x => println(x)" denotes the function that takes argument x and executes println(x)). An *outport branch* is denoted "out =!=> e ==> cont", where out is a channel (or outport), e is an expression which evaluates to a value of the type of out, and c is a computation, which we again call a continuation (this continuation is optional). The alt waits until there is another thread ready to communicate at the other end of the channel corresponding to one of the branches. In the case of an inport branch, the continuation is applied to the value received. In the case of an outport branch, the value of the expression is sent, and the continuation (if present) is executed.

A branch may have a boolean *guard*: a branch may be selected only if the guard evaluates to true (however, we do not model guards in our analysis). As an example, the following code implements a bounded buffer, with maximum capacity Bound.

```
val queue = new Queue[Int]
while(true){
  alt(
    queue.length < Bound & in =?=> { x => q.enqueue(x) }
    | queue.nonEmpty & out =!=> { q.dequeue() }
  )
}
```

We say that a branch is *feasible* if the port has not been closed and the guard is true. Above, the inport branch is feasible only if the buffer is not full; the outport branch is feasible only if the buffer is not empty. If no branch of an alt is feasible, it throws an AltAbort exception.

There are two restrictions on the usage of alts: a port may not be simultaneously feasible in two alts (although a non-alt thread may use a port concurrently to an alt); and both ports of a channel may not simultaneously be feasible in alts. The implementation throws an exception if these restrictions are not respected.

In this paper, we build CSP models of the implementations of channels and alts. We then use the model checker FDR to analyse them against appropriate specifications. The implementations of channels and alts are tricky: each has multiple modes of operation, and can be used concurrently by multiple threads. These factors also provide a challenge to the analysis. We do not include the Scala implementation here, because there is so much code; but it can be obtained from ???. Likewise, we do not include the CSP model of the implementation, but instead concentrate on the specification, which we consider more interesting; all the CSP can be obtained from ???.

The rest of the paper is structured as follows. In Section 2 we give a brief overview of the syntax and semantics of CSP. In Section 3 we consider synchronous channels. We describe different aspects of channels incrementally, in the interests of clarity. We start by considering just the (untimed) send and receive operations: we give an overview of the implementation, and of the corresponding part of the CSP model. We then describe the correctness condition for these operations, namely *synchronisation linearisation* [**?**]. We also describe a related progress property, *synchronisation progressibility*. We present the corresponding CSP specification and refinement check for each property. extend our analysis to consider the closing of channels: this is of particular interest, because the analysis revealed an error in an earlier implementation. Fixing this error, required fairly substantial changes to the implementation. Performing this analysis helps to clarify what the correctness condition should be, and so helps to focus on the critical point. We then extend the analysis to the timed send and receive operations (but ignoring the interactions with alts at this point).

In Section 4 we consider alts. We describe the high-level design, concerning the interactions (via operation calls) between alts and channels, sketch some implementation details, and describe aspects of the CSP model. We then describe a direct analysis: we consider a system constructed from an alt with a fixed number of branches, and associated channels; we construct a corresponding CSP specification for synchronisation linearisability and progressibility. This analysis was useful in helping to develop a correct implementation: it revealed various flaws with earlier versions. However, the analysis suffers from a state-space explosion, and so it's possible to analyse only rather small systems.

In Section 5, we perform an alternative, compositional, verification. We

build a more abstract CSP description of a synchronous channel, describing the way it reacts to operation calls and interacts with alts, but abstracting away from details of the implementation: we call this an *idealised channel*. We show that the CSP model of the channel implementation refines this idealised channel. Likewise, we build an idealised model of an alt, and show that it is refined by the model of the implementation. Finally, we combine the idealised alt with a fixed number of idealised channels, use FDR to analyse the combination, and argue that this implies correctness for the corresponding combination of the implementation models. This approach scales much better than the direct analysis. A challenge of this approach is that the implementations of a channel and an alt each assumes that other components act correctly, i.e. follows the protocol that defines interactions between them: we describe how to deal with this challenge.

$\boxed{\cdots}$

We employed various techniques in our CSP modelling. We present some of these in an appendix: they are rather orthogonal to the main focus of this paper, but we believe they would be useful elsewhere.

We consider our main contributions to be the following:

- The modelling of a fairly large body of code, and the development of related modelling techniques;
- The specification of synchronisation linearisation and progressibility, and the illustration of how they can be tested using model checking;
- The demonstration that this technique can discover real bugs on code;
- The demonstration of compositional verification, in particular where each component makes assumptions about the correct behaviour of other components.

1.1 TO DO

Alt sent values are calculated only when sent.

2 Overview of CSP

$\boxed{\cdots}$

3 Modelling and analysing a synchronous channel

In this section, we consider the operation of a single synchronous channel, without considering interaction with alts.

We start by considering just the send ("!") and receive ("?") operations. We outline the implementation for these operations, how we model them in CSP, and then how we analyse their correctness. In later subsections, we extend to include the timed operations and the closing of channels; for the moment we elide details related to those operations or to the use of alts.

The implementation is based on a monitor (more precisely, using an implementation of monitors within the SCL library). All operations are carried out while holding the monitor's lock. In addition, the implementation uses a number of *conditions*: one thread can wait on a condition until another thread signals on the same condition.

The implementation uses a variable value to store the value that a thread is currently trying to send (if any). Further, it uses a variable status to store the status of the current exchange, one of the following:

Empty: No sender has deposited a value;

Filled: A sender has deposited a value, but no receiver has yet read it;

Read: A receiver has read the current value, but the corresponding sender has not yet returned.

The implementation also uses a variable receiversWaiting, which records how many receivers are currently waiting to receive a value.

A sender waits (on a condition slotEmptied) until the status is Empty. It then stores its value in value, sets the status to Filled, and signals (on a condition slotFull) to a receiver. It next waits (on a condition continue) until the status is Read. Finally, it sets the status back to Empty, and signals (on slotEmptied) to the next sender.

A receiver waits (on slotFull) until the status is Filled (updating receiversWaiting before and after). It then sets the status to Read, signals to the sender (on continue), and returns the value stored in value.

We now outline the CSP model. The model uses small fixed types representing the type of data communicated by the channel, and the type of thread identities. We discuss the choices for these types later.

The monitor is modelled by a CSP module, encapsulating a process that maintains the state of the monitor, specifically: (1) which thread, if any, holds the lock on the monitor; and (2) which threads are waiting on which

conditions. When no thread holds the lock, the monitor process allows a thread, other than any of the waiting threads, to acquire the lock. The thread that currently holds the lock can:

- Release the lock;
- Wait on a condition, at which point it also releases the lock;
- Signal on a condition, at which point a thread that it waiting on that condition is recorded as no longer waiting (but needs to reacquire the lock before it can continue).

The monitor module provides an interface to client processes corresponding to the different monitor operations.

Each variable in the implementation of the channel is modelled by a CSP process, with CSP channels to allow it to be read or written.

Each of the send and receive operations can then be straightforwardly translated into a CSP process that performs the operations on the monitor and variables, following the Scala code.

The Scala code uses several assertions. We model these by having the CSP process diverge if the property does not hold. Later, we use FDR to verify that such divergences do not occur.

Each operation is framed by begin and end events:

- The event beginSend.t.x represents a thread with identity t calling !x on the channel, and the event endSend.t.SendSuccess represents it successfully returning (later we extend the channel to model unsuccessful calls that find the channel has been closed).
- Likewise the events beginReceive.t and endReceive.t.ReceiveSuccess.x represent thread t calling and returning from a call of the receive operation that successfully receives the value x.

Inside the module is a process corresponding to each thread, which accepts the begin events, simulates the operation, and then performs the appropriate end event. The module encapsulates this so only the begin and end events are visible. A process outside the module can simulate calling the operation by synchronising on those events.

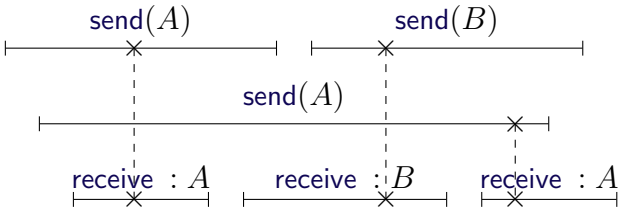
3.1 Analysing the basic channel

We now analyse the basic model of the channel. We start by describing abstractly the required property, and then describe how to capture it using CSP.

Each invocation of the send operation should synchronise with a corresponding invocation of receive: the two invocations should overlap in time, and the receive should return the value of the send's parameter.

In [**?**], we introduced a correctness condition, *synchronisation linearisation*, corresponding to synchronisation objects like this. The idea is that each synchronisation should appear to take place between the begin and end of the two corresponding invocations; different synchronisations should occur in a one-at-a-time order. We call the points at which the synchronisations appear to take place *synchronisation points*. (The definition is based on *linearisation*, the standard correctness property for concurrent datatypes, where invocations of operations should appear to take place in a one-at-a-time order, with each between the begin and end of that invocation.)

The diagram below illustrates the idea, which illustrates a particular history of a channel (or a trace of the CSP model).



Time goes from left to right in the diagram. Each horizontal line represents an invocation, with the end points representing the call and return (or begin and end events). The "×"s in the diagram illustrate the synchronisation points of the corresponding invocations, linked by a dashed vertical line.

Thus a history (or trace) is synchronisation linearisable if it is possible to identify synchronisation points with the desired properties. This corresponds to identifying which invocations synchronise with one another. A synchronisation object is synchronisation linearisable if all of its histories are synchronisation linearisable [**?**].

We now describe how we use model checking to test synchronisation linearisability of the channel. We create an instance C of the channel module. Below, expressions of the form C::v represent a value v from this instance. We create a process System that allows threads to call the send and receive operations.

We build a CSP specification process that allows precisely the traces that are synchronisation linearisable. We use events of the form $\mathsf{sync.t_1.t_2.x}$ to represent a synchronisation point between sender $\mathsf{t_1}$ and receiver $\mathsf{t_2}$, passing data value x. We build a *lineariser* process for each thread as follows, which ensures that the sync events occur between the corresponding begin and end events.

```
Lin(t) =
  C::beginSend.t?x → sync.t?other!x → C::endSend.t.SendSuccess → Lin(t)
  □ C::beginReceive.t → sync?other!t?x → C::endReceive.t.ReceiveSuccess.x → Lin(t)
```

When sending, the thread can synchronise with any other thread other, passing its value x. When receiving, the thread can synchronise with any other thread, accepting its value x; this x is subsequently returned by the invocation.

We combine the Lin processes in parallel, with their natural alphabets, so the linearisers for threads t_1 and t_2 synchronise on events of $\mathsf{sync}.t_1.t_2$ and $\mathsf{sync}.t_2.t_1$.

```
alphaLin(t) =
  {|C::beginSend.t, C::endSend.t, C::beginReceive.t, C::endReceive.t|} ∪
  {|sync.t.other, sync.other.t | other ← ThreadID−{t}|}
Spec0 = ||| t ← ThreadID • [alphaLin(t)] Lin(t)
```

Thus each trace of Spec_0 represents a history that is synchronisation linearisable, together with the corresponding synchronisation points. By hiding the synchronisation points, we obtain a process whose traces represent precisely those histories that are synchronisation linearisable. We can then use FDR to check that the system is synchronisation linearisable.

```
Spec = Spec0 \ {|sync|}
assert Spec ⊑T System
```

We can also test the same refinement in the stable failures model (which implies the refinement in the traces model).

```
assert Spec ⊑F System
```

This captures a useful progress property, which says that if a synchronisation is possible (i.e. there is at least one send and one receive operation that have been called but not yet returned), then some such synchronisation must happen, and so the relevant threads reach a state where they can return. (If several different synchronisations are possible, then the refinement test allows any to happen.) We call this property *synchronisation progressiblity*.

Finally, as discussed above, we can test that System does not diverge, to verify that no thread in the implementation throws an exception. Further, this verifies that threads cannot perform an infinite amount of internal activity without any thread returning. These tests can be combined by testing for refinement in the failures-divergences model.

```
assert Spec ⊑FD System
```

**] Number of threads.

3.2 Closing channels

We now consider the closing of channels.

Implementing the close operation correctly proved harder than expected. An earlier version of the implementation suffered from a bug involving three threads acting concurrently: thread A calls send(x), thread B calls receive, and thread C calls close. Under certain conditions, it was possible for thread B to return successfully, having received x, but for thread A to see the channel closed, and so throw a Closed exception. We consider this behaviour to be incorrect: either both A and B should think the communication has succeeded, or both should throw Closed exceptions.

The implementation uses a boolean variable isChanClosed that records whether the channel is closed. The close operation sets this variable, and signals to all the threads waiting on conditions.

When a thread calls send or receive, if isChanClosed is set, it throws a Closed exception. Likewise, if a sending thread waits on slotEmptied, it performs a similar check when it receives a signal.

If a receiving thread waits on slotFull, when it receives a signal, it first checks whether status holds Filled. If so, it continues as above: thus we prioritise completing the communication over checking whether the channel has been closed (this seems necessary for correct interaction with alts). Otherwise, it checks whether the channel has been closed, and if so throws an exception; otherwise, it waits again on slotFull.

If a sending thread waits on continue, when it receives a signal, it first checks whether status holds Read, and if so continue as described earlier. Otherwise, it must be the case that isChanClosed has been set (the implementation asserts this, and the analysis below checks this). However, if there is a receiver waiting (as recorded by receiversWaiting), then that receiver will eventually return the value being sent, so the sending thread should also return successfully. If there is no waiting receiver, the sending the thread throws a Closed exception. The precise order of checks is thus rather subtle: this is where the earlier implementation went wrong.

Adapting the CSP model to model closing of channels is straightforward. The endSend and endReceive channels are extended to pass a value Closed corresponding to the Closed exception. The close operation is modelled as for earlier operations, framed by events on channels beginClose and endClosed.

To analyse this extended model, we adapt the process System from earlier to also allow threads to close channels.

The previous model of the channel was (in the terminology of [**?**]) *stateless*: no state is carried forward from one synchronisation to another. However, when we consider closing of the channel, it becomes *stateful*, with two

states, open or closed. (Other synchronisation objects have more interesting states.)

Our specification will treat close as a linearisable operation: it will appear to take place atomically, at some point, called the *linearisation point*, between the beginClose and endClose events (equivalently, the close operation can be thought of as a unary synchronisation, involving a single thread, in contrast to the earlier binary synchronisations). We require that the history is consistent with this closing: synchronisations between sends and receives should take place before the linearisation point of the close; and sends and receives that return Closed should be linearised after the close.

We use an event close.t to represent the linearisation point of a close operation by thread t. Further, we use an event closed.t to represent the linearisation point of a send or receive operation by thread t that returns Closed, because it finds the channel is closed.

Within the specification, the state of the channel is recorded by the process ChanSpec. When the channel is open, it allows threads to synchronise, or allows a thread to close the channel. When the channel is closed, it allows linearisation points of sends or receives that return Closed, or allows the linearisation point of another close operation (a close operation on a channel that is already closed has no effect).

```
ChanSpec = sync?t1?t2?x → ChanSpec □ close?t → ChanSpecClosed
ChanSpecClosed = isClosed?t → ChanSpecClosed □ close?t → ChanSpecClosed
alphaChanSpec = {|sync, close, isClosed|}
```

We adapt the lineariser processes to reflect the closing of channels. A sending thread can either synchronise with another thread, as before, or find the channel is closed and so return the Closed value. (The SendingLin process that describes this is parameterised by the corresponding endSend channel, to facilitate extension to the timed operations later.) Receiving is treated similarly. Further, the linearisation point for a close operation can take place between beginCloseand endClose events.

```
Lin(t) =
  C::beginSend.t?x → SendingLin(t, x, C::endSend.t)
  □ C::beginReceive.t → ReceivingLin(t, C::endReceive.t)
  □ C::beginClose.t → close.t → C::endClose.t → Lin(t)

SendingLin(t, x, endChan) =
  sync.t?other!x → endChan.SendSuccess → Lin(t)
  □ isClosed.t → endChan.Closed → Lin(t)

ReceivingLin(t, endChan) =
  sync?other!t?x → endChan.ReceiveSuccess.x → Lin(t)
```

$\Box$ isClosed . t $\rightarrow$ endChan . Closed $\rightarrow$ Lin(t)

We combine the linearisers as before (with suitably extended alphabets). We then synchronise them with ChanSpec on the relevant events, and hide those events.

$$\mathsf{Spec}_0 = \Vert\ t \leftarrow \mathsf{ThreadID} \bullet [\mathsf{alphaLin(t)}]\ \mathsf{Lin(t)}$$
$$\mathsf{Spec} = (\mathsf{Spec}_0\ [\vert\ \mathsf{alphaChanSpec}\ \vert]\ \mathsf{ChanSpec}) \setminus \mathsf{alphaChanSpec}$$

Thus Spec allows all traces, containing the begin and end events, that are synchronisation linearisable. Thus testing Spec $\sqsubseteq_T$ System verifies synchronisation linearisability for this system. Performing the corresponding test in the stable failures model also verifies the corresponding progress property. Finally, performing the test in the failures-divergences model also verifies that all assertions in the code pass, and that threads cannot perform an infinite amount of internal activity without a thread returning.

3.3 Timed operations

We now consider the timed send and receive operations.

The SCL conditions described earlier provide a timed wait operation: the thread waits until either it receives a signal, or the time is elapsed; the operation returns a boolean indicating whether a signal was received. The timed send and receive operations are based around this.

The sendWithin(x, duration) operation initially waits on slotEmptied until either status holds Empty or the channel is closed (which it rechecks when it receives a signal), or the deadline is reached. If the channel is closed, it throws a Closed exception. If it timedout, it returns false. Otherwise, it continues as in the untimed operation, except it waits on continue until at most duration after the initial call. If it finds that status is Read, then the send has been successful; it continues as in the untimed operation, and returns true. Otherwise status must still hold Filled and value must stll hold x (the implementation asserts this, and the analysis below checks this). If the channel is closed, it continues as in the untimed operation. Otherwise, it must have timedout, so it sets status to Empty to clear its value, signals to any thread waiting on slotEmptied, and returns false.

The receiveWithin(duration) operation acts much as the untimed version, except it waits on slotFull until at most duration after the initial call. If it finds that status holds Filled, it continues as in the untimed case, returning a suitable Some value. If the channel is closed, it throws a Closed exception. otherwise it must have timedout, so returns None.

We now describe the CSP model of these operations. Our analysis does not consider absolute time; thus we abstract away from the duration of a

timed send or receive. The difficult part of the implementation is getting the synchronisations right, rather than the length of the delay.

The CSP model of an SCL monitor also models timed waits. It records which threads are doing timed waits on which conditions. Such threads can receive a signal, as for untimed waits. In addition, they can timeout and acquire the lock on the monitor, assuming no other thread holds the lock. Thus we model that such threads can eventually timeout, but don't model the length of the delay.

The sendWithin and receiveWithin operations can then be modelled in CSP much as before, using these timed waits.

We adapt the specification of synchronisation linearisability as follows. We introduce events timeout.t to represent the linearisation point of a sendWithin or receiveWithin operation by thread t that times out. We then adapt the definition of the lineariser processes for these operations as follows, adding the possibility of such a timeout to the possibilities of the untimed operations.

```
Lin(me) =
  ... -- as before
  □ C::beginSendWithin.me?x → SendingWithinLin(me, x, C::endSendWithin.me)
  □ C::beginReceiveWithin.me → ReceivingWithinLin(me, C::endReceiveWithin.me)

SendingWithinLin(me, x, endChan) =
  SendingLin(me, x, endChan)
  □ timeout.me → endChan.Timeout → Lin(me)

ReceivingWithinLin(me, endChan) =
  ReceivingLin(me, endChan)
  □ timeout.me → endChan.Timeout → Lin(me)
```

Further, we adapt the ChanSpec process to allow timeout events only before the channel is closed. The rest of the construction and checks are then as before.

4 Alts

We now describe the implementation and modelling of alts. We start by describing the high-level interactions between an alt and channels, and then go on to describe how those interactions are implemented within alts and channels, and outline how they are modelled in CSP.

4.1 High-level design

We describe the interactions between an alt and relevant channels via function calls. We call this the *alt protocol*.

When an alt runs, it starts by registering, in turn, with each of the relevant ports whose guard evaluates to true. This asks the port whether it is ready to communicate. The alt calls a function

```
def registerIn(alt: AltT, index: Int, iter: Int): RegisterInResult[A]
```

on each of its inports, where alt is a reference to the calling alt, index is the index of the branch within the alt, and iter is an iteration number (used only for assertions). It receives a result of type RegisterInResult[A], where A is the type of data passed by the port, of one of the following forms.

RegisterInSuccess(x): the port was willing to communicate, and the alt has received x from it;

RegisterInWaiting: the port is not currently willing to communicate (but the registration has been recorded);

RegisterInClosed: the port has been closed.

Similarly, the alt calls a function

```
def registerOut(alt: AltT, index: Int, iter: Int, value: () => A): RegisterOutResult
```

on each of its outports, where alt, index and iter are as for registerIn, and value is a computation that, when evaluated, produces the value to be sent. It receives a result of one of the following forms.

RegisterOutSuccess: the port was willing to communicate, and the alt has sent it a value;

RegisterOutWaiting: the port is not currently willing to communicate (but the registration has been recorded);

RegisterOutClosed: the port has been closed.

If one of the registrations is successful, the alt deregisters from the waiting branches, via functions deregisterIn and deregisterOut. It then executes the continuation of the successful branch.

Figure 1 gives an example: an alt first registers unsuccessfully with an inport *IP*; then it registers successfully from an outport *OP*; and finally deregisters from *IP*.

If no registration is successful, and the alt finds that every port is closed or has a guard that is false, then it throws an AltAbort exception. Otherwise,

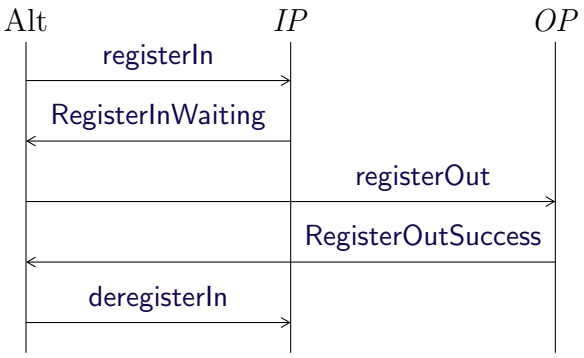


Figure 1: Sequence diagram illustrating an alt that registers successfully with a port.

it waits for a callback from one of the ports with which it is registered, of one of the following forms.

- If the alt is registered at the inport of a channel, and another thread tries to send on the channel, it calls

  ```
  def maybeReceive(value: A, index: Int, iter: Int): Boolean
  ```

  where value is the value it is trying to send to the alt, and index and iter match the values provided during registration. This asks the alt whether it is still willing to receive from the inport.

- If the alt is registered at the outport of a channel, and another thread tries to receive on the channel, it calls

  ```
  def maybeSend[A](index: Int, iter: Int): Option[A]
  ```

  This asks the alt whether it is still willing to send a value to the inport.

- If another thread closes the channel, it calls

  ```
  def portClosed(index: Int, iter: Int)
  ```

If the alt receives a call of maybeReceive or maybeSend, it responds positively to the first such call, returning true to a maybeReceive, or Some(x) to a maybeSend, where x is the value it sends. It then deregisters from the remaining branches. Finally, the alt executes the continuation of the successful branch. Figure 2 gives an example of a successful callback of maybeSend from an outport with which the alt is registered.

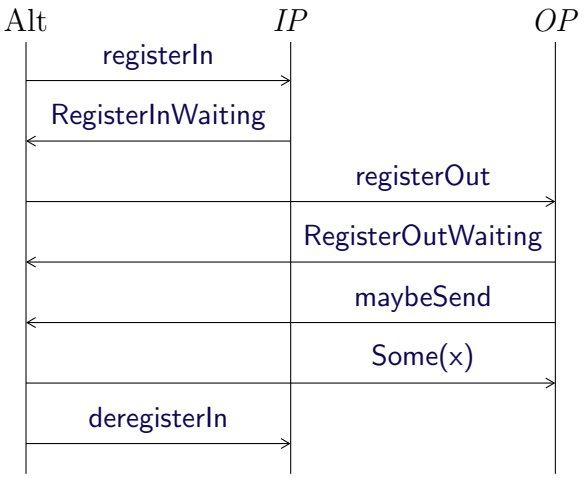


Figure 2: Sequence diagram illustrating a successful callback from a port.

If the alt receives multiple callbacks to maybeReceive or maybeSend (including during deregistration), it responds negatively, returning false or None, respectively. If all the channels with which the alt is registered call portClosed, the alt throws an AltAbort.

4.2 Implementation details

We now describe some details of the implementation. Below we will use the term "alt-thread" for the thread that is running the alt, and "channel-thread" for a thread performing an operation in a channel that makes a callback to the alt.

Each call of a function on a channel (registering or deregistering) uses that channel's lock, to avoid races.

The implementation of the alt is based on a monitor, more specifically a monitor provided by the Java Virtual Machine (JVM). (JVM monitors are more efficient than SCL monitors, because they are implemented directly in the JVM; however, they do not allow targeting of signals.)

The alt-thread holds the alt's lock throughout the registration phase. Each callback function has to obtain this lock, so those functions are blocked until registration is complete.

However, it would be a mistake for the alt-thread to continue to hold the alt's lock during deregistration, for this could lead to deadlocks. Suppose it did continue to hold the lock, and consider the case that the alt-thread is trying to deregister from channel c, at the same time that a channel-thread is

trying to perform a callback from c: the channel-thread holds the lock on c, so the deregistration would be blocked; but the alt-thread holds the lock on the alt, so the callback would be blocked.

The implementation uses a variable done that gets set to true when a branch is found that is willing to communicate, either during registration or as the result of a callback. If a callback of maybeSend or maybeReceive finds that done is true, it can return a negative result. Otherwise, it stores relevant information (like the value it is sending in the case of maybeReceive, and the index of the relevant branch), evaluates the value to be received in the case of maybeSend, sets done to true, signals to the waiting alt-thread, and returns a positive result.

If the alt-thread fails to communicate during registration, and not all ports are closed, it waits to receive a signal. If done is true, it deregisters from other branches and runs the continuation of the relevant branch. Otherwise, if all ports are closed, it throws an AltAbort.

We now describe how the implementation of channels is extended to deal with alts.

The registerIn function checks whether another alt is currently registered (which would be an error), and if so throws an exception. If the channel is closed, it returns RegisterInClosed. If there is a waiting sender (corresponding to the variable status holding Filled), it acts like a standard receive, setting status to Read and signalling to the sender, and then returns a RegisterInSuccess result. Otherwise it records the registration, and returns RegisterInWaiting.

The registerOut function is somewhat similar. If there are waiting receivers, it first waits for any current exchange to finish (i.e. for status to hold Empty). If there are still waiting receivers, it continues as for a standard send: it stores its value, sets status to Filled and signals to a receiver. It then waits on the continue condition for a receiver to take the value, and resets status to Empty. This latter wait is necessary to ensure correct synchronisation: without it, the value could be taken by a new receiver that calls receive only after the alt-thread has returned.

The deregister functions simply clear the registration information.

If a call of send or sendWithin finds that there is an alt registered at the inport, it calls maybeReceive on that alt, and reacts accordingly. Calls to receive or receiveWithin act similarly, calling maybeSend.

Finally, if a channel is closed, it calls portClosed on any registered alt.

4.3 CSP modelling for alts

We now describe how to model an alt, and its interactions with channels, using CSP. Much of the construction of the model follows a similar form to

the model of a channel. We highlight the main differences.

A JVM monitor is modelled by a CSP module, in a similar way to an SCL monitor. One difference, however, concerns a bug in the implementation of JVM monitors: a thread that is waiting for a signal may resume, even though it has received no signal! This is known as a *spurious wakeup*. The implementation of an alt guards against spurious wakeups by performing suitable checks when resuming after a wait, and waiting again if appropriate.

Our model reflects the possibility of spurious wakeups, allowing a waiting thread t to resume either as the result of a signal, or a spurious wakeup, modelled by the event spuriousWakeup.t. We run this in parallel with a *regulator process* that nondeterministically chooses whether or not to allow a spurious wakeup. This last point is important: if we allowed unrestricted spurious wakeups, there is a danger that our analysis would seem to show that suitable progress properties are satisfied, when in reality the system can get stuck unless there are sufficient spurious wakeups.

In the model of a channel, the registration and deregistration functions are wrapped in suitable begin and end events. In the model of an alt, the alt-thread performs these begin and end events, and then reacts to the result in the end event. Later, we combine the models of the alt and channels together, synchronising on these events, so as to achieve the desired effect.

Similarly, in the model of an alt, the callback functions are wrapped in suitable begin and end events. In the model of a channel, the channel-thread performs these events, and reacts to the result in the end event.

Each use of the alt is framed with events beginAlt.me and endAlt.me.result where result is of one of the following forms.

- AltSend.i.x, representing the sending of value x on the port corresponding to the branch with index i;
- AltReceive.i.x, similarly representing receiving of x;
- AltAbort, representing an AltAbort exception.

4.4 Direct analysis of an alt and channels

We now describe how we can perform a direct analysis of an alt together with channels.

We build a system that uses an alt with a fixed number of branches. Below, we consider an alt with two branches. The branches are defined using a definition such as

branches = <InPortBranch.c1, OutPortBranch.c2>

In different tests, we can vary whether the branches correspond to inports or outports, so test different combinations. We also create two instances C1 and C2 of channels, with identities c1 and c2. We combine these together in parallel, synchronising on the begin and end events that correspond to the alt calling operations on a channel, or vice versa.

We combine these together with some threads. One thread, which we denote AltThread, repeatedly runs the alt. The other threads, from set ChanThreads, repeatedly call the main operations on the channels.

As an initial test, we can check whether this system is divergence-free. However, recall that a thread waiting in the alt can perform a spurious wakeup, denoted by the event A1::spuriousWakeup. If we hide this event, it turns out that the system can diverge, corresponding to a waiting thread repeatedly having a spurious wakeup, rechecking the relevant condition, and waiting again. This is not a behaviour we should be concerned about: spurious wakeups do happen, but they are rather rare; in practice, such spurious wakeups will have a tiny effect on system performance. If we keep the spurious wakeups visible, then FDR verifies that the system cannot diverge check!: no assertions fail, and the only possible source of infinite internal activity is the spurious wakeups.

In the checks below, we hide the spurious wakeups. The checks will be carried out in the stable-failures model, so we should consider whether the potential divergence is masking possible errors, by making critical states unstable. But recall that we included in the model of the monitor a regulator process that could block all spurious wakeups. Thus for every state that is unstable because of the possibility of a spurious wakeup, there is another, stable state where the regulator blocks the spurious wakeup. This way of abstracting the spurious wakeups corresponds to Roscoe's *lazy abstraction* [**?**].

We now consider the appropriate correctness condition. This is an extension of the correctness condition for a single channel from Section 3.

We extend the sync events to include the identity of the channel being used: sync.t_1.t_2.c.x represent a synchronisation between a sending thread t_1 and a receiving thread t_2, both of which are channel-threads, using channel c, passing value x. We introduce similar events of the form altSync.t_1.t_2.c.x to represent a synchronisation between a sending thread t_1 and a receiving thread t_2, where one of the threads is the alt-thread.

We build lineariser processes for the channel threads. These are very similar to as in Section 3, so we elide some parts. They are extended for operations on either C1 or C2. Further, they allow synchronisations with the alt-thread.

```
-- Lineariser for a channel thread.
```

```
ChanThreadLin(me) =
  C1::beginSend.me?x → LinSending(me, x, c1, C1::endSend.me)
  □ C2::beginSend.me?x → LinSending(me, x, c2, C2::endSend.me)
  □ ... −− similar for other operations

LinSending(me, x, c, endChan) =
  sync.me?other:others(me)!c!x → endChan.SendSuccess → ChanThreadLin(me)
  □ altSync.me?altThread!c!x → endChan.SendSuccess → ChanThreadLin(me)
  □ isClosed.me.c → endChan.Closed → ChanThreadLin(me)

LinReceiving(me, c, endChan) =
  sync?other:others(me)!me!c?x → endChan.ReceiveSuccess.x → ChanThreadLin(me)
  □ altSync?altThread!me.c?x → endChan.ReceiveSuccess.x → ChanThreadLin(me)
  □ isClosed.me.c → endChan.Closed → ChanThreadLin(me)

LinSendingWithin(me, x, c, endChan) =
  LinSending(me, x, c, endChan)
  □ timeout.me.c → endChan.Timeout → ChanThreadLin(me)

LinReceivingWithin(me, c, endChan) =
  LinReceiving(me, c, endChan)
  □ timeout.me.c → endChan.Timeout → ChanThreadLin(me)
```

We similarly build a lineariser process for the alt-thread. We introduce an event allClosed which will represent the linearisation point of an alt usage that finds all the channels are closed and so throws an AltAbort. Let inports and outports be the sets of inports and outports that the alt uses, and let the function index give the index of a particular branch. The definition below captures that before a successful return, the alt-thread must perform a suitable synchronisation with a channel-thread, and that before returning an AltAbort, it must detect that all the channels are closed.

```
AltLin(me) =
  A1::beginAlt.me → (
    altSync?other:ChanThreads!me?c:inPorts?x →
      A1::endAlt.me.AltReceive.index(InPortBranch.c).x → AltLin(me)
    □ altSync.me?other:ChanThreads?c:outPorts?x →
        A1::endAlt.me.AltSend.index(OutPortBranch.c).x → AltLin(me)
    □ allClosed → A1::endAlt.me.AltAbort → AltLin(me)
  )
```

We build a ChanSpec process for each channel. Each extends the earlier definition to allow altSync events, but only before the channel is closed. Further, each can perform allClosed when the channel is closed; we synchronise the ChanSpec processes on this event, so it can happen only when all channels

are closed, as required.

```
ChanSpec(c) =
  sync?t1?t2!c?x → ChanSpec(c) □ altSync?t1?t2!c?x → ChanSpec(c)
  □ timeout?t!c → ChanSpec(c)   □ close?t!c → ChanSpecClosed(c)
ChanSpecClosed(c) =
  isClosed ?t!c → ChanSpecClosed(c) □ allClosed → ChanSpecClosed(c)
  □ close?t!c → ChanSpecClosed(c)
```

We combine the different processes together, much as before. We can then verify that the system refines this specification in both the traces and stable-failures models, showing that the system is synchronisation linearisable, and satisfies the corresponding progress property.

However, this approach suffers from a state-space explosion. Details.

5 Compositional verification

We now consider an alternative approach to analysing the combination of an alt and some channels.

1. We show that a single channel (in isolation) is consistent with a more abstract model, which we call IdealisedChannel;

2. We likewise show that an alt (in isolation) is consistent with a more abstract model, which we call IdealisedAlt;

3. We show that the combination of an IdealisedAlt and several IdealisedChannels satisfies a property similar to that in the previous section.

This allows us to deduce that the combination of an alt and several channels satisfies the same property.

The analysis is complicated by the following issue. A channel works correctly under the assumption that any alt that interacts with it follows the alt protocol. However, if the alt does not follow the protocol, then the channel can act incorrectly. For example, if an alt tries to register *twice* with a channel, or tries to deregister without having previously registered, then the implementation throws an exception, and the CSP model diverges. Likewise, the alt works correctly under the assumption that channels follow the alt protocol, but may act incorrectly otherwise.

When we analyse a channel in isolation, we cannot assume that its environment (an alt) follows the protocol; and when we analyse an alt in isolation, we cannot assume that its environment (channels) follows the environment: to make these assumptions would be circular reasoning.

Our approach is as follows. Our idealised model of a channel will detect if its environment breaks the alt protocol, and if so allow arbitrary behaviour. Thus testing whether the model of a channel refines this specification is equivalent to testing whether it satisfies the specification when the environment does follow the protocol. More precisely, we will produce a process IdealisedChannel that describes allowed behaviour when the environment follows the protocol, but produces an *error event* from a set Errors_C if the environment breaks the protocol. We then test whether the channel refines the process

$$(\ \mathsf{IdealisedChannel}\ \ [|\mathsf{Errors}_C|\rangle\ \mathsf{Any})\ \setminus\ \mathsf{Errors}_C$$

where Any allows arbitrary behaviour (the definition depends upon the semantic model we are using).

Likewise, we build an idealised model of an alt that allows arbitrary behaviour if its environment breaks the protocol. The model will be of the form

$$(\ \mathsf{IdealisedAlt}\ \ [|\mathsf{Errors}_A|\rangle\ \mathsf{Any})\ \setminus\ \mathsf{Errors}_A$$

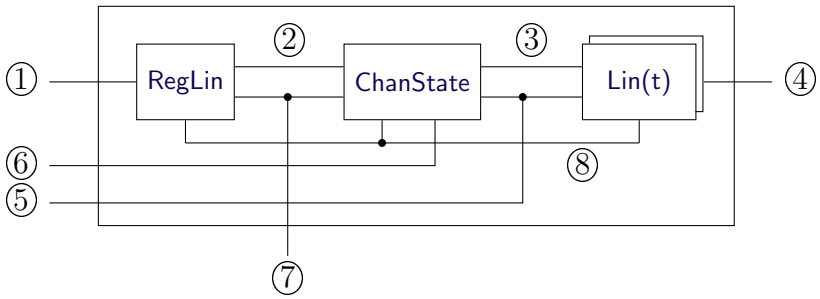
where IdealisedAlt describes allowed behaviour when the environment follows the protocol, but performs an event from Errors_A if the environment breaks the protocol.

We can then consider the combination of IdealisedAlt and several instances of IdealisedChannel. Part of the analysis will show that no events from $\mathsf{Errors}_C \cup \mathsf{Errors}_A$ occur: each component follows the alt protocol, provided the other does. But we can also show that this combination satisfies a property similar to that in the previous section, allowing us to deduce that the corresponding combination of an alt and channels satisfies that property.

5.1 Idealised model of a channel

We now give an idealised model of a channel: this describes the behaviour of a channel in terms of the calls and returns of functions, while abstracting away from the implementation. We assume (for simplicity) a single thread, AltThread, that runs any alt that interacts with the channel.

The construction is made more complicated by the fact that the model needs to deal with the begin and end events for function calls, whereas these functions will take effect at their linearisation points. To deal with this, we construct the model from several components, as depicted in Figure 3. The component ChanState keeps track of the state of the channel: whether an alt is registered at a port, and whether the channel is closed. The component RegLin is responsible for linearising registrations and deregistrations. Each



Key. We use BNF-style notation to capture channels with similar names; for example, we write "(begin|end)Send" to denote the beginSend and endSend channels. The interface with an alt appears on the left; the interface with channel-threads appears on the right; error events appear below; internal events appear inside the box.

①: (begin|end)Register(In|Out), (begin|end)Deregister(In|Out);

②: register(In|Out)Wait; deregister(In|Out), isClosed.AltThread;

③: sync, callMaybeSend, callMaybeReceive, close, isClosed, commit, timeout;

④: (begin|end)Send, (begin|end)Receive, (begin|end)SendWithin, (begin|end)ReceiveWithin, (begin|end)Close;

⑤: (begin|end)MaybeReceive, (begin|end)MaybeSend;

⑥: (begin|end)portClosed;

⑦: registerError, deregister(In|Out)Error;

⑧: register(In|Out)Sync.

Figure 3: Construction of the idealised channel.

```
RegLin =
  beginRegisterIn . AltThread ? alt ? index → RegLinRegIn(alt, index)
  □ beginRegisterOut . AltThread ? alt ? index → RegLinRegOut(alt, index)
  □ beginDeregisterIn . AltThread ? alt ? index → RegLinDeregIn(alt, index)
  □ beginDeregisterOut . AltThread ? alt ? index → RegLinDeregOut(alt, index)

RegLinRegIn(alt, index) =
  let endChan = endRegisterIn . AltThread . alt within
  registerInSync ? t ? x → endChan . RegisterSuccess . x → RegLin
  □ registerInWait . alt . index → endChan . RegisterWaiting → RegLin
  □ isClosed . AltThread → endChan . RegisterClosed → RegLin
  □ registerError → STOP

RegLinRegOut(alt, index) =
  let endChan = endRegisterOut . AltThread . alt within
  ( ⊓ x : Data • registerOutSync ? t ! x → endChan . RegisterSuccess . x → RegLin)
  □ registerOutWait . alt . index → endChan . RegisterWaiting → RegLin
  □ isClosed . AltThread → endChan . RegisterClosed → RegLin
  □ registerError → STOP

RegLinDeregIn(alt, index) =
  deregisterIn . alt . index → endDeregisterIn . AltThread . alt → RegLin
  □ deregisterInError . alt . index → STOP

RegLinDeregOut(alt, index) =
  deregisterOut . a . index → endDeregisterOut . AltThread . a → RegLin
  □ deregisterOutError . a . index → STOP
```

Figure 4: Definition of the RegLin process, controlling registration and deregistration.

component Lin(t) is responsible for linearising the function calls of channel-thread t. We describe these in more detail below. Some details of the definitions are unobvious, and were found by trial and error: their correctness is evidenced by the subsequent successful refinement checks.

The RegLin process is defined in Figure 4; it accepts the relevant begin events, with each function being handled by a different subsidiary process.

The process RegLinRegIn(alt, index) models the linearisation of a call of registerIn(alt, index), which can happen in several ways.

- A synchronisation with a waiting send by a channel thread t, with the alt receiving x, is modelled by registerInSync.t.x; this synchronises with the corresponding Lin(t) process.

- An unsuccessful registration is modelled by registerInWait.alt.index.
- A registration that finds the channel closed is modelled by an event isClosed.AltThread.
- In incorrect registration, where an alt is already registered, is modelled by the event registerError.

The ChanState process synchronises on each of these events, to make sure the correct one is selected, based on the current channel state.

The process RegLinRegOut(alt, index) models the linearisation of a call of registerOut(alt, index), in a similar way. The processes RegLinDeregIn(alt, index) and RegLinDeregOut(alt, index) model deregistrations, including the possibility of an erroneous deregistration.

The Lin(t) processes are defined in Figure 5. Each accepts the relevant begin and end events from channel-thread t, with most of the functions modelled by subsidiary processes.

The process SendingLin models the linearisation of the send and sendWithin functions; the parameter timed indicates the latter case. The subprocess Success indicates a successful send. There are several cases.

- A synchronisation with another channel thread t' is represented by the event sync.t.t'.x. In the implementation, thread t might not be able to return immediately: it must first obtain the lock. This is modelled here by a synchronisation on event commit.t with ChanState, which might be blocked in some circumstances.
- A synchronisation with an alt performing registerIn is captured by the event registerInSync.t.x, described earlier. Again, the thread t might not be able to return immediately, as modelled here by a synchronisation on commit.t.
- A decision to call maybeReceive on an alt alt registered with index index is modelled by the event callMaybeReceive.t.alt.index.x; this event is a synchronisation with ChanState, which allows it only when alt is suitably registered. Thread t then calls maybeReceive, waits to receive back the result, and reacts accordingly.
- The thread can find the channel closed via the event isClosed.t.
- In the case of the sendWithin function, the thread can timeout, modelled by the event timeout.t.

```
Lin(t) =
  beginSend.t?x → SendingLin(t, x, endSend.t, false)
  □ beginReceive.t → ReceivingLin(t, endReceive.t, false)
  □ beginSendWithin.t?x → SendingLin(t, x, endSendWithin.t, true)
  □ beginReceiveWithin.t → ReceivingLin(t, endReceiveWithin.t, true)
  □ beginClose.t → close.t → isClosed.t → endClose.t → Lin(t)

SendingLin(t, x, endChan, timed) =
  let Success = endChan.SendSuccess → Lin(t) within
  sync.t?t':ChanThread−{t}!x → commit.t → Success
  □ registerInSync.t.x → commit.t → Success
  □ callMaybeReceive.t?alt?index!x → beginMaybeReceive.t.alt.index.x →
    endMaybeReceive.t.alt?res →
    (if res then Success else SendingLin(t, x, endChan, timed))
  □ isClosed.t → endChan.Closed → Lin(t)
  □ timed & timeout.t → endChan.Timeout → Lin(t)

ReceivingLin(t, endChan, timed) =
  let Success(x) = endChan.ReceiveSuccess.x → Lin(t) within
  sync?t':ChanThread−{t}!t?x → Success(x)
  □ registerOutSync.t?x → Success(x)
  □ callMaybeSend.t?alt?index → beginMaybeSend.t.alt.index → (
    endMaybeSend.t.alt.Some?x → Success(x)
    □ endMaybeSend.t.alt.None → ReceivingLin(t, endChan, timed) )
  □ isClosed.t → endChan.Closed → Lin(t)
  □ timed & timeout.t → endChan.Timeout → Lin(t)
```

Figure 5: Definition of the Lin processes, controlling channel-thread functions.

The process ReceivingLin models the linearisation of the receive and receiveWithin, in a similar way. There is no need for the extra synchronisation on the commit channel in this case: in the implementation, these functions can return straightaway after synchronisation.

Finally, the Lin processes directly deal with closing. The event close.t represents the linearisation point of the operation. The close function might not be able to return immediately: the channel might need to call portClosed on a registered port (modelled within ChanState), and wait for that call to return; the event isClosed.t becomes available at that point.

The ChanState process is in Figure 6. The parameter regStatus records the current registration status, and is taken from the following type.

```
datatype RegStatus = NoReg | InReg.AltID.Index | OutReg.AltID.Index
```

```
ChanState(regStatus) =
  let  regIns  = getRegIns(regStatus)
       regOuts = getRegOuts(regStatus)
  within
  regStatus  =  NoReg & (
     registerInSync ?t.x → ChanState(NoReg)
    □ registerOutSync?t.x → ChanState(NoReg)
    □ registerInWait ?alt?index → ChanState(InReg.alt.index)
    □ registerOutWait?alt?index → ChanState(OutReg.alt.index)
    □ deregisterIn ?alt?index →  ChanState(NoReg)
    □ deregisterOut?alt?index →  ChanState(NoReg)
  )
  □
  regStatus  ≠  NoReg & (
     registerError  → STOP
    □ deregisterIn ?(alt.index):regIns  →  ChanState(NoReg)
    □ deregisterOut?(alt.index):regOuts → ChanState(NoReg)
    □ deregisterInError ?(alt.index):AltIndex−regIns → STOP
    □ deregisterOutError?(alt.index):AltIndex−regOuts → STOP
    □ callMaybeReceive?t?(alt.index):regIns?x → beginMaybeReceive.t.alt.index.x →
          endMaybeReceive.t.alt?res → ChanState(NoReg)
    □ callMaybeSend?t?(alt.index):regOuts → beginMaybeSend.t.alt.index →
          endMaybeSend.t.alt?res → ChanState(NoReg)
  )
  □
  sync?t1?t2:others(t1)?x → ChanState(regStatus)
  □
  commit?t → ChanState(regStatus)
  □
  timeout?t → ChanState(regStatus)
  □
  close?t → (
    if  regStatus  =  NoReg then ChanStateClosed
    else
      beginPortClosed.t?(alt.index):regIns∪regOuts →
      endPortClosed.t.alt  → ChanStateClosed
  )
```

Figure 6: The ChanState process.

where AltID is the type of alt identities, and Index is the type of indices of branches. The subtypes of RegStatus represent that no alt is currently registered, or that an alt is registered at the inport or outport corresponding to a particular index.

In the definition of ChanState, the values regIns and regOuts store the (empty or singleton) sets of AltID.Index pairs corresponding to registrations at the inport or outport, respectively (these are calculated using straightforward helper functions).

Several possibilities are available when there is no registered alt.

- A registerIn or registerOut operation may synchronise with a waiting channel thread.
- A registerIn or registerOut operation may fail to synchronise; the registration status is updated appropriately.
- A deregisterIn or deregisterOut operation may happen; which has no effect. These events can arise if an alt is trying to deregister concurrently with an unsuccessful call back of maybeReceive or maybeSend, which clears the registration status.

Several possibilities are available when there is a registered alt.

- Another registration attempt would be erroneous, represented by the event registerError.
- The currently registered alt might be deregistered.
- An attempt to deregister a different alt would be erroneous (here AltIndex represents the set of all AltID.Index pairs).
- A channel thread t may decide to call maybeReceive or maybeSend corresponding to the current registration. The call is made and returns, and the registration is cleared (regardless of the result).

Note that other events are blocked during calls to maybeReceive or maybeSend; this corresponds to the fact that in the implementation, the relevant channel-thread keeps the lock on the channel.

Other possibilities are available regardless of the registration status.

- Two threads t1 and t2 may synchronise on a communication.
- A sending thread t may commit to returning (see the earlier explanation concerning the Lin(t) process).

```
ChanStateClosed =
  isClosed ?t → ChanStateClosed
  □ close ?t → ChanStateClosed
  □ commit ?t → ChanStateClosed
  □ deregisterIn ?alt ?index → ChanStateClosed
  □ deregisterOut ?alt ?index → ChanStateClosed
```

Figure 7: The ChanStateClosed process.

- A channel-thread in a sendWithin or receiveWithin can timeout.
- The channel can be closed. If there is a registered port, the channel calls portClosed on the relevant alt, and waits for it to return.

The process ChanStateClosed in Figure 7 corresponds to the channel having been closed.

- This process can perform inClosed.t, synchronising with either RegLin (if t is the alt-thread) or Lin(t), as described earlier.
- The channel could be closed again (having no effect).
- The process could synchronise with a Lin(t) process on event commit.t: this corresponds to sending thread t having synchronised with another thread before the channel was closed.
- A deregistration may happen, which has no effect: this corresponds to the alt-thread starting the deregistration concurrently with the callback of portClosed.

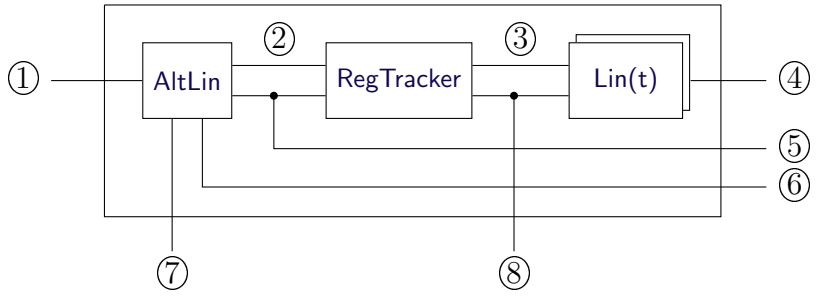
The components are combined together as illustrated in Figure 3, hiding all the internal events (those inside the box in the figure). This produces a process IdealisedChannel. Recall that we want to allow arbitrary behaviour after one of the events that represents that the environment has not followed the alt protocol, on channels registerError, deregisterInError and deregisterOutError. In the stable-failures model, the process CHAOS(Interface) allows arbitrary behaviour over the set Interface (the interface of the channel); in the failures-divergences model, the process DIV allows arbitrary behaviour. We therefore define the following.

$$\mathsf{Errors}_C = \{| \mathsf{registerError}, \mathsf{deregisterInError}, \mathsf{deregisterOutError} |\}$$
$$\mathsf{ChannelSpec}_F = (\mathsf{IdealisedChannel}\ [|\mathsf{Errors}_C|\rangle\ \mathsf{CHAOS(Interface)})\ \setminus\ \mathsf{Errors}_C$$
$$\mathsf{ChannelSpec}_D = (\mathsf{IdealisedChannel}\ [|\mathsf{Errors}_C|\rangle\ \mathsf{DIV})\ \setminus\ \mathsf{Errors}_C$$

We can then compare the CSP model of a synchronous channel implementation against the idealised model. We create a system using the model of the implementation, allowing threads to call appropriate operations. We can then verify that this system refines $\mathsf{ChannelSpec}_F$ and $\mathsf{ChannelSpec}_D$ in the relevant models.

** Number of threads, timing. FD check faster than F check.

5.2 Idealised model of an alt



Key. The interface with the alt-thread appears on the left; the interface with channels appears on the right; error events and spurious wakeups appear below; internal events appear inside the box.

①: (begin|end)Alt;

②: beginRegistration, endRegistration, getToDeregister(In|Out), deregisterDone, endWait;

③: maybe(Send|Receive), portClosed;

④: (begin|end)Maybe(Send|Receive), (begin|end)PortClosed;

⑤: endRegister(In|Out), endDeregister(In|Out);

⑥: beginRegister(In|Out), beginDeregister(In|Out);

⑦: spuriousWakeup.AltThread;

⑧: maybe(Send|Receive)Error, portClosedError.

Figure 8: Construction of the idealised alt.

We now give an idealised model of an alt. As with the idealised channel,

```
Lin(t) =
  beginMaybeReceive.t?index?x → LinMaybeReceive(t, index, x)
  □ beginMaybeSend.t?index → LinMaybeSend(t, index)
  □ beginPortClosed.t?index → LinPortClosed(t, index)

LinMaybeReceive(t, index, x) =
  maybeReceive.t.index.x?res → endMaybeReceive.t.res → Lin(t)
  □ maybeReceiveError.t.index → STOP

LinMaybeSend(t, index) =
  maybeSend.t.index?res → endMaybeSend.t.res → Lin(t)
  □ maybeSendError.t.index → STOP

LinPortClosed(t, index) =
  portClosed.t.index → endPortClosed.t → Lin(t)
  □ portClosedError.t.index → STOP
```

Figure 9: Definition of the Lin(t) processes.

the idealised alt identifies when its environment breaks the alt protocol, and signals via an appropriate error event.

We assume the definition of a value branches defining the branches of the alt, as a sequence of values InPortBranch.c and OutPortBranch.c for channels c. We assume a single alt-thread, AltThread, and a collection ChanThreads of channel-threads.

The idealised alt is constructed from several components, as depicted in Figure 8. The component ChanState keeps track of registrations of the alt at ports, or whether those ports are closed. Each component Lin(t) linearises callbacks by channel-thread t. The component RegLin linearises the main call of the alt by the alt-thread. We describe these in more detail below.

The Lin(t) processes are defined in Figure 9. Each accepts the begin event of a callback function, which is handled by a different subsidiary process. Each callback may be either linearised correctly, or with an error event if it breaks the alt protocol: The RegTracker process selects the appropriate event, based on whether the alt is currently registered at the relevant port.

The AltLin process, which linearises the main calls of the alt, is defined in Figure 10. It initially signals to the RegTracker that it is beginning registration, via event beginRegistration. It then iterates through branches, trying to register at each branch in turn (the expression nth(branches, i) gives the branch at index i). The RegTracker keeps track of the results of the registration attempts. If a registration is successful, AltLin moves to the deregistra-

```
AltLin = beginAlt.AltThread → beginRegistration → Register(0)

Register(i) =
  if i = size then endRegistration?ac →
    if ac then endAlt.AltThread.AltAbort → AltLin else Waiting
  else Register1(i, nth(branches, i))

Register1(i, InPortBranch.c) =
  beginRegisterIn.AltThread.c.i → (
    endRegisterIn.AltThread.c.RegisterSuccess?x → Deregister(AltReceive.i.x)
    □ endRegisterIn.AltThread.c.RegisterWaiting → Register(i+1)
    □ endRegisterIn.AltThread.c.RegisterClosed → Register(i+1)
  )

Register1(i, OutPortBranch.c) =
  beginRegisterOut.AltThread.c.i → (
    endRegisterOut.AltThread.c.RegisterSuccess?x → Deregister(AltSend.i.x)
    □ endRegisterOut.AltThread.c.RegisterWaiting → Register(i+1)
    □ endRegisterOut.AltThread.c.RegisterClosed → Register(i+1)
  )

Deregister(result) =
  getToDeregisterIn?c.i → beginDeregisterIn.AltThread.c.i →
      endDeregisterIn.AltThread.c → Deregister(result)
  □ getToDeregisterOut?c.i → beginDeregisterOut.AltThread.c.i →
      endDeregisterOut.AltThread.c → Deregister(result)
  □ deregisterDone → endAlt.AltThread.result → AltLin

Waiting =
  endWait?result → (
    if result = AltAbort then endAlt.AltThread.result → AltLin
    else Deregister(result))
  □ (spuriousWakeup.AltThread → Waiting ⊓ STOP)
```

Figure 10: Definition of the AltLin processes.

tion phase. If it gets to the end of branches, it synchronises with RegTracker on channel endRegistration to indicates to RegTracker that registration is over. AltLin receives from RegTracker a boolean, denoted ac, that indicates whether all ports have been closed; if so, the call on the alt returns with an AltAbort; otherwise, the AltLin moves to the waiting phase.

The deregistration phase is defined by the process Deregister(result) (where result will be the result of the call of the alt). It repeatedly gets from RegTracker information about a port to be deregistered, calls the relevant deregistration function, and waits for it to return. When there are no more ports to deregister, RegTracker signals this on deregisterDone, at which point AltLin ends the call on the alt.

The waiting phase is defined by the process Waiting. It waits for a synchronisation on channel endWait with RegTracker, as a result of a successful callback. The endWait event contains the result of the call to alt: if this is an AltAbort, AltLin simply returns; otherwise it moves to the deregistration phase. In addition, the process might have a spurious wakeup while waiting, corresponding to event spuriousWakeup.AltThread.

The RegTracker process is defined starting in Figure 11. It waits to receive notification, via event beginRegistration that registration has started. However, any callback before this point would be outside the alt protocol, in which case it signals an error.

The process RegTracker1 has a parameter reg which is a mapping from indices of branches to the type RegInfo, defined as follows.

datatype RegInfo = NoReg | InPortReg.ChanID | OutPortReg.ChanID | Closed

The clauses in RegInfo represent, respectively, that the corresponding branch is not registered, registered at an inport, registered at an outport, or the port is closed. Initially all indices are marked as not registered.

The process RegTracker1 synchronises on the end events of call to registerIn and registerOut. In the case of a RegisterWaiting or RegisterClosed result, reg is updated to map the relevant index to an appropriate value. If a registration attempt is successful, the process moves to state RegTracker2, corresponding to the deregistration phase, described below. If the AltLin synchronises on endRegistration, indicating the end of the registration phase, the RegTracker sends a value indicating whether all the ports have been closed; if so, it returns to its initial state; otherwise it moves to state RegTracker3, corresponding to the waiting phase. Note that during the registration phase, RegTracker blocks all callbacks, reflecting the behaviour of the implementation.

The process RegTracker2(reg) deals with deregistration. The names inRegs, outRegs and allRegs are set to the indices of registered inports, registered outports, and all registered ports, respectively. It can send to AltLin in-

```
RegTracker =
  beginRegistration  → RegTracker1({i ↦ NoReg | 0 ≤ i < length(branches)})
  □ maybeReceiveError?t?index → STOP
  □ maybeSendError?t?index → STOP
  □ portClosedError?t?index → STOP

RegTracker1(reg) =
  endRegisterIn.AltThread?c!RegisterWaiting →
    RegTracker1(reg ⊕ {indexFor(InPortBranch.c) ↦ InPortReg.c})
  □ endRegisterOut.AltThread?c!RegisterWaiting →
    RegTracker1(reg ⊕ {indexFor(OutPortBranch.c) ↦ OutPortReg.c})
  □ endRegisterIn.AltThread?c!RegisterClosed →
    RegTracker1(reg ⊕ {indexFor(InPortBranch.c) ↦ Closed})
  □ endRegisterOut.AltThread?c!RegisterClosed →
    RegTracker1(reg ⊕ {indexFor(OutPortBranch.c) ↦ Closed})
  □ endRegisterIn.AltThread?c!RegisterSuccess?x → RegTracker2(reg)
  □ endRegisterOut.AltThread?c!RegisterSuccess?x → RegTracker2(reg)
  □ let ac = allClosed(reg) within
    endRegistration ! ac → (if ac then RegTracker else RegTracker3(reg))

RegTracker2(reg) =
  let inRegs = getInRegs(reg)
      outRegs = getOutRegs(reg)
      allRegs = inRegs ∪ outRegs within
  ( □ (i ↦ InPort.c): reg • getToDeregisterIn.c.i → RegTracker2(reg) )
  □ ( □ (i↦ OutPortReg.c): reg • getToDeregisterOut.c.1 → RegTracker2(reg) )
  □ allRegs={} & deregisterDone → RegTracker
  □ endDeregisterIn.AltThread?c →
    RegTracker2(reg ⊕ {indexFor(InPortBranch.c) ↦ NoReg})
  □ endDeregisterOut.AltThread?c →
    RegTracker2(reg ⊕ {indexFor(OutPortBranch.c) ↦ NoReg})
  □ maybeReceive?t?index:inRegs?x!false → RegTracker2(reg ⊕ {index ↦ NoReg})
  □ maybeSend?t?index:outRegs!None → RegTracker2(reg ⊕ {index ↦ NoReg})
  □ portClosed?t?index:allRegs → RegTracker2(reg ⊕ {index ↦ Closed})
  □ maybeReceiveError?t?index:Index−inRegs → STOP
  □ maybeSendError?t?index:Index−outRegs → STOP
  □ portClosedError?t?index:Index−allRegs → STOP
```

Figure 11: The RegTracker, RegTracker1 and RegTracker2 processes.

```
RegTracker3(reg) =
  let inRegs = getInRegs(reg)
      outRegs = getOutRegs(reg)
      allRegs = union(inRegs,outRegs) within
  maybeReceive?t?index:inRegs?x!true →
    endWait.AltReceive.index.x → RegTracker2(reg ⊕ {index ↦ NoReg})
  □ ( ⊓ x : Data • maybeSend?t?index:outRegs!Some.x →
      endWait.AltSend.index.x → RegTracker2(reg ⊕ {index ↦ NoReg}) )
  □ portClosed?t?index:allRegs → (
    let reg' = reg ⊕ {index ↦ Closed} within
    if allClosed(reg') then endWait.AltAbort → RegTracker else RegTracker3(reg')
  )
  □ maybeReceiveError?t?index:Index−inRegs → STOP
  □ maybeSendError?t?index:Index−outRegs → STOP
  □ portClosedError?t?index:Index−allRegs → STOP
```

Figure 12: The RegTracker3 process.

formation about a port that can be deregistered (channels getToDeregisterIn and getToDeregisterOut), or an indication that there are no such (channel deregisterDone). It synchronises on the end events of deregistrations, and updates its state to map the relevant index to NoReg (via the subsidiary process RecordDereg, omitted for brevity).

RegTracker2 can synchronise with Lin(t) on the linearisation point of a maybeReceive operation corresponding to a registered inport; at this point, the callback will be unsuccessful, as captured by the final false field in the event. Likewise, it can synchronise on the linearisation point of a maybeSend operation corresponding to a registered outport, which again will be unsuccessful. Further, it can synchronise on the linearisation point of a closed operation on a registered port. However, any callback not corresponding to a suitably registered port would be outside of the alt protocol, so an error is signalled.

The RegTracker3 process (Figure 12) deals with the waiting phase, waiting for callbacks from registered ports. If can synchronise with Lin(t) on the linearisation point of a maybeReceive operation corresponding to a registered inport. The operation will be successful, as captured by the final true field in the event. It informs AltLin of the success, on channel endWait, and moves to the deregistration phase. Likewise, it can synchronise on the linearisation point of a maybeSend operation corresponding to a registered outport; this succeeds with a value x sent (modelled as being chosen nondeterministically). Further, it can synchronise on the linearisation point of a close operation; if

all ports are now closed, it indicates this to AltSpec on channel endWait. However, any callback not corresponding to a suitably registered port would be outside of the alt protocol, so an error is signalled.

The components are combined together, as illustrated in Figure 8, to produce a process IdealisedAlt. We then allow arbitrary behaviours after the error events, much as for the idealised model of a channel.

$$\mathsf{Errors}_A = \{|\mathsf{maybeReceiveError}, \mathsf{maybeSendError}, \mathsf{portClosedError}|\}$$
$$\mathsf{AltSpec}_F = (\ \mathsf{IdealisedAlt}\ \ [|\mathsf{Errors}_A|\rangle\ \mathsf{CHAOS(Interface)}\ )\ \setminus\ \mathsf{Errors}_A$$
$$\mathsf{AltSpec}_D = (\ \mathsf{IdealisedAlt}\ \ [|\mathsf{Errors}_A|\rangle\ \mathsf{DIV}\ )\ \setminus\ \mathsf{Errors}_A$$

We can then compare the CSP model of an alt implementation to the idealised model. We can verify that the implementation model refines $\mathsf{AltSpec}_F$ and $\mathsf{AltSpec}_D$ in the relevant models.

** Number of threads, timing.

5.3 Combining the idealised models

We now perform the final step in our compositional verification by combining an IdealisedAlt and corresponding IdealisedChannels.

Fix, for the moment, a definition of branches, defining the branches of an alt. We can then build a system comprising an IdealisedAlt (based on branches) and a corresponding set of IdealisedChannels, synchronising appropriately.

We can then use FDR to verify that this system is divergence-free (when the spuriousWakeup events are kept visible), and that it refines (in the stable-failures model) the specification from Section 4.4 (with the spuriousWakeup events hidden). In particular, the latter test verifies that the system does not perform any of the error events from $\mathsf{Errors}_A \cup \mathsf{Errors}_C$: each component follows the alt protocol assuming the other does.

Recall that we earlier considered processes ChannelSpec and AltSpec, that allowed arbitrary behaviours after an error event. The fact that the combination of IdealisedChannels and IdealisedAlt does not perform any error events means that the same combination of ChannelSpecs and AltSpec would behave identically, so that combination would also pass the tests in the previous paragraph (although the checks would take slightly longer).

Further, we earlier showed that the models of the implementations refine ChannelSpec and AltSpec. This then implies that the same combination of the implementations would also pass the tests described above (although the checks might be infeasible in practice): formally, this is because all CSP operators are monotonic with respect to refinement.

Type sizes, timing

A Modelling techniques

We describe here a few modelling techniques that we found useful during our analysis. These techniques are rather orthogonal to the main ideas of the paper, but we think they might be useful elsewhere.

We model variables using a CSP process as follows. Here value is the current value of the variable, and get and set are channels on which a thread t can get or set the value.

```
Var(value, get, set) =
  get?t!value → Var(value, get, set)
  □ set?t?value' → Var(value', get, set)
```

For example, the implementation of a channel has a variable receiversWaiting that stores the current number of threads that are waiting to receive on the channel. In the implementation, this can be an arbitrary Int; however, we would expect the value to be non-negative and at most the number of threads in the CSP model (and the subsequent analysis confirms this). The variable can therefore be modelled as follows.

```
N = card(ThreadID)
channel getReceiversWaiting, setReceiversWaiting : ThreadID.{0..N}
ReceiversWaiting = Var(0, getReceiversWaiting, setReceiversWaiting)
```

The Scala implementation contains a number of assertions. We model such assertions by having the model diverge if the property does not hold; our subsequent analysis verifies that such a divergence does not happen. (An alternative is to perform an explicit error event; however, our experience is that it is easy to get that approach wrong, for example by accidentally omitting error from a process's alphabet.) The following two macros are useful.

```
Assert(b, P) = if b then P else DIV
Assert1(b) = Assert(b, SKIP)
```

In the former, the continuation P is provided explicitly, whereas the latter can be used with sequential composition. The two processes Assert(b, P) and Assert1(b); P are CSP-equivalent; but the FDR compiler treats them differently. In the former, P is compiled only if b is true, but in the latter, P is compiled regardless. This can make a difference if compilation would fail when b is false. The following macros, to increment or decrement the receiversWaiting variable, illustrate this point; the use of Assert ensures that the compiler does not try to produce an event on setReceiversWaiting outside the correct range {0..N}.

```
IncReceiversWaiting(me) =
```

```
  getReceiversWaiting.me?r → Assert(r < N, setReceiversWaiting.me.r+1 → SKIP)
DecReceiversWaiting(me) =
  getReceiversWaiting.me?r → Assert(r > 0, setReceiversWaiting.me.r−1 → SKIP)
```

We now discuss the modelling of Scala functions in the implementation, in particular how to model the value returned. CSP has no notion of returning a value. Instead, we adopt a continuation-passing approach. If the Scala function returns a result of type A, the corresponding CSP process is given a parameter cont :: A → Proc, representing a continuation, i.e. what the rest of the program does with the result. Returns from the Scala function are then modelled by applying cont to the returned value. A call to the function is modelled by providing a suitable function as the continuation.

A continuation-passing style can also be used to model computations that are passed to other constructs. For example, the following macro captures an **if** statement. The test of the **if** is modelled by a process test(k) that applies its continuation parameter k to an appropriate boolean.

```
If :: (((Bool) → Proc) → Proc, Proc, Proc) → Proc
If(test, P, Q) = test(λres • if res then P else Q)
```

For example, Scala code **if**(result == None && status != Filled){...} **else** {...} can be modelled as follows.

```
let Test(k) = if result = None then getStatus.me?s → k(s ≠ Filled) else k(false)
within If(Test, ..., ...)
```

Likewise, a **while** loop can be captured using the following macro.

```
While :: (((Bool) → Proc) → Proc, Proc) → Proc
While(test, body) = test(λb • if b then body; While(test, body) else SKIP)
```