Analysing a Library of Concurrency Primitives using CSP

Gavin Lowe[1]

Abstract. We carry out an analysis of message-passing concurrency primitives, namely a synchronous channel and an alt (alternation) construct, implemented in Scala. We model these primitives using the process algebra CSP, and analyse them using the model checker FDR. We consider the correctness properties of *synchronisation linearisation* (informally, that each completed operation execution corresponds to a correct synchronisation) and *progressibility* (informally, that executions don't get stuck if they could synchronise): we show how these properties can be captured in CSP. Our initial analysis discovered an error in a previous implementation; our subsequent analysis helped us to produce a correct implementation. It turns out that a direct analysis of the composition of an alt and corresponding channels scales quite poorly. To overcome this, we perform a compositional analysis: we show that a channel and an alt each satisfies a more abstract description; and show that the composition of these abstract descriptions satisfies synchronisation linearisation and progressibility.

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, barrier synchronisations and semaphores. SCL is based on CSO (Concurrent Scala Objects) [Suf08]; it has similar aims to JCSP [WA14].

In this paper we analyse the message-passing primitives from SCL: we build CSP models of them, and then use the model checker FDR to test for correctness. To our surprise, the analysis revealed a (fairly minor) bug in the implementation of channels.

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.send(x) (also written as c!x) sends the value x on c; the expression c.receive() (also written as c?()) receives and returns a value. We consider just *synchronous* channels in this paper: the sending thread must wait until there is a thread willing to receive, so that the two operation executions synchronise. Channels are typed: the type SyncChan[A] represents synchronous channels that send data of type A. A channel is composed of an *out-port*, where values are sent, and an *in-port*, where values are received; the ports can be thought of as the two ends of the channel.

Channels in SCL 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,

[1] Author's address: St Catherine's College, University of Oxford, UK. E-mail: gavin.lowe@cs.ox.ac.uk.
Correspondence and offprint requests to: Gavin Lowe

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. However, each synchronisation is between one sender and one receiver; the channel is responsible for pairing up a sender with a receiver. Shared ports are useful, for example, to allow communication between a server and multiple clients.

A channel can be closed (by the close operation): subsequently, an attempt to send or receive on the channel will throw a Closed exception. Closing can be useful, for example, to allow a producer thread to indicate the end of a stream of data; this allows consumer threads to move on (maybe terminating). Likewise, a consumer can close a channel to signal to a producer that it is unwilling to receive more data. (Closing is somewhat similar to the concept of poisoning in JCSP [SA05], where a special "poison" value is sent to indicate the end of a stream of data.)

An alternation, or *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 *in-port branch* is denoted "in =?=> f" where in is a channel (or in-port), and f is a function whose argument matches the type of data passed on in; we call f a *continuation* (above, "x => println(x)" denotes the function that takes argument x and executes println(x)). An *out-port branch* is denoted "out =!=> e ==> cont", where out is a channel (or out-port), e is an expression whose value matches the type of data passed on out, and cont 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, at which point the two threads can synchronise to transmit a value. In the case of an in-port branch, the continuation is applied to the value received. In the case of an out-port branch, the value of the expression is sent, and the continuation (if present) is executed.

A branch may have a boolean *guard*: a branch can be selected only if the guard evaluates to true. As an example, the following code implements a bounded buffer, with maximum capacity Bound.

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

This example also illustrates that the calculation of the value sent in an out-port branch might have side effects; therefore the expression that produces the value is evaluated only once the alt commits to communication via that branch.

We say that a branch is *feasible* if the port has not been closed and the guard is true. Above, the in-port branch is feasible only if the buffer is not full and in is not closed; the out-port branch is feasible only if the buffer is not empty and out is not closed. If no branch of an alt is feasible, the alt 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 port may be simultaneously feasible in an alt and used by a non-alt thread); 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 full Scala implementation here, because

there is so much code; but it can be obtained via the paper's web page[2]. Likewise, we do not include the full CSP model of the implementation; we instead concentrate on the specification, which we consider more interesting. All the CSP can likewise be obtained from the paper's web page.

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* [LL25]: informally, each completed operation execution corresponds to a correct synchronisation, where an execution of send(x) synchronises with an execution of receive that returns x. We also describe a related progress property, *synchronisation progressibility*, informally, that operation executions don't get stuck when they could synchronise. We present the corresponding CSP specification and refinement check for each property.

In Section 4, we 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 the analysis in this paper helped to clarify what the correctness condition should be, and so helped to focus on the critical point. Further, the correct implementation was found with the help of the model checking described in this paper. We also extend the analysis to the timed send and receive operations (but ignoring the interactions with alts at this point).

In Section 5 we consider alts. We describe the high-level design in terms of the interactions (via operation calls) between alts and channels; we 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, and construct a corresponding CSP specification for synchronisation linearisability and progressibility. This analysis was again 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 (an alt with two branches and channels, used by three threads).

In Section 6, 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. A challenge for this analysis is that each component makes assumptions about other components with which it is composed, namely that they follow the protocol that defines interactions between them; a component can fail (typically by throwing an exception) if another component does not follow the protocol. We describe how we capture these assumptions; the problem is similar to the techniques of rely-guarantee [Jon83] and assumption-commitment [MC81, PJ91], but we believe our approach is new. 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.

In Section 7, we analyse the SCL implementations of barrier synchronisations, monitors and semaphores. We sum up in Section 8.

We employed various techniques in our CSP modelling. We present some of these in Appendix A: they are rather orthogonal to the main focus of this paper, but we believe they would be useful elsewhere. In Appendix B we show a technical result: if a model satisfies the property of synchronisation progressibility when two data values are used, a corresponding model where more data values are used also satisfies the property; this implies that analysing the smaller model with two data values is enough.

We consider our main contributions to be the following:

- The modelling of a fairly large body of code, larger than previous similar analyses;
- The development of related modelling techniques;
- The illustration of how synchronisation linearisation and progressibility can be tested using model checking;
- The demonstration that this technique can discover real bugs in code;

[2] `https://www.cs.ox.ac.uk/people/gavin.lowe/SCL-CSP/`.

- The demonstration of compositional verification, in particular where each component makes assumptions about the correct behaviour of other components, within a model checking setting.

1.1. Related work

CSP has been used to analyse message passing concurrency primitives on a number of previous occasions. Welch and Martin [WM00a] present a model of Java multi-threading (in particular, monitors), including a model of a channel within their own concurrency API; in [WM00b] they also include a model of alternation. I [Low11] derive and verify a generalisation of the alt construct, by building CSP models and analysing them using FDR. In [Low19], I use CSP to discover the cause of a deadlock in an implementation of a synchronous channel. Pedersen and Chalmers [PC23] analyse communication channels in ProcessJ, in the context of cooperative scheduling.

I [Low17] use CSP and FDR to show that a lock-free queue is linearisable, a property on which synchronisation linearisation is based. Dongol and Le-Papin [DLP21], and Raad et al. [RLW+24] use similar techniques to verify variants of opacity in software transactional memories.

CSP has also been used more widely to analyse concurrent systems. Lawrence [Law05] describes the use of CSP and FDR in an industrial setting, for the analysis of a system for connection pooling. Mota and Sampaio [MS01] analyse a subset of the control system of a satellite, modelled in CSP-Z. Zhao [Zha22] analyses several concurrent objects. CSP and FDR have been widely used to analyse security protocols (e.g. [Low96]).

Hopkins and Roscoe [RH07], and Pay [Pay23] describe compilers for compiling from a simple shared-variable language into CSP.

2. Overview of CSP

In this section we give a brief overview of the syntax for the fragment of CSP that we will be using in this paper. We then review the relevant aspects of CSP semantics, and the use of the model checker FDR in verification. For more details, see [Ros10].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. Processes communicate via atomic *events*. Events often involve passing values over channels; for example, the event $\mathsf{c.3}$ represents the value $\mathsf{3}$ being passed on channel c. Channels may be declared using the keyword **channel**; for example, **channel** $\mathsf{c} : \mathsf{Int}$ declares c to be a channel that passes an Int. (In this paper, the word "channel" can mean either an SCL channel or a CSP channel; the intention should be clear from the context.) The notation $\{|\mathsf{c}|\}$ represents the set of events over channel c.

The simplest process is STOP, which represents a deadlocked process that cannot communicate with its environment. The process SKIP is a process that terminates immediately, represented by the distinguished event $\checkmark$.

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

The process $\mathsf{P} \Box \mathsf{Q}$ can act like either P or Q, the choice being made by the environment: the environment is offered the choice between the initial events of P and Q. By contrast, $\mathsf{P} \sqcap \mathsf{Q}$ may act like either P or Q, with the choice being made nondeterministically, not under the control of the environment. $\Box\, \mathsf{x:X} \bullet \mathsf{P(x)}$ is an indexed external choice, with the choice being made over the processes $\mathsf{P(x)}$ for x in X. Likewise, $\sqcap\, \mathsf{x:X} \bullet \mathsf{P(x)}$ represents a nondeterministic choice over the $\mathsf{P(x)}$.

The process **if** b **then** P **else** Q represents a conditional. The process $\mathsf{b} \,\&\, \mathsf{P}$ is a guarded process, that makes P available only if b is true; it is equivalent to **if** b **then** P **else** STOP.

The process $\mathsf{P; Q}$ represents a sequential composition of P and Q: initially, P is run, but when it terminates (as indicated by event $\checkmark$), Q is run (but the $\checkmark$ is hidden).

The process $\mathsf{CHAOS(A)}$ can perform any events from the set A, or can refuse any of those events; however, it cannot diverge. It is defined by:

$$\mathsf{CHAOS(A)} = (\Box\, \mathsf{a:A} \bullet \mathsf{a} \rightarrow \mathsf{CHAOS(A)}) \sqcap \mathsf{STOP}.$$

Thus it allows arbitrary non-divergent behaviours over A. By contrast, DIV is a divergent process that performs an unbounded number of internal τ events.

The process P [|E|⟩Q (sometimes denoted P θ_E Q) denotes a throw-catch mechanism: initially, P is executed, but if it performs an event from E, control is passed to Q; for example, the events from E could represent exceptions thrown by P, and Q could be an exception handler.

The process P [|A|] Q runs P and Q in parallel, synchronising on events from A. The process P [A‖B] Q again runs P and Q in parallel: P is given alphabet A, and Q is given alphabet B; they synchronise on events from A ∩ B. The process ‖x:X • [A(x)] P(x) represents an indexed parallel composition of processes P(x) for x in X; each P(x) is given alphabet A(x); processes synchronize on events in the intersection of their alphabets. The process P ||| Q interleaves P and Q, i.e. runs them in parallel with no synchronisation. |||x:X • P(x) represents an indexed interleaving.

The process P \ A acts like P, except the events from A are hidden, i.e. turned into internal τ events.

CSP, as implemented in the model checker FDR, is supported by a functional sublanguage, roughly equivalent to Haskell, but without type classes, and augmented with sets and mappings.

FDR also supports modules, where related definitions can be encapsulated. In particular, modules can be parameterised (like classes in an object-oriented programming language). The typical form of a module definition is as follows:

```
module M(x, y)
  ... -- private definitions
exports
  ... -- public definitions
endmodule
```

where x and y are formal parameters. Then multiple instances can be created, for example:

```
instance M1 = M(X1, Y1)
instance M2 = M(X2, Y2)
```

A value x within M1 can be referenced as M1::x. Later, we create a parameterised module representing an SCL channel, and then create multiple instances representing distinct channels.

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

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

We say that a process *diverges* if it can perform an infinite number of internal (hidden) τ events without any intervening visible events. The failures-divergences model describes a process by a set of divergences and a set of failures. The model satisfies two closure properties, in order to correctly capture intuitions: (1) the divergences of process P contains all traces after which P can diverge, and also all extensions of those traces; and (2) the failures of process P contains all its stable failures, and all failures using a trace from its divergences set. Thus the immediately divergent process DIV has all divergences and all failures. We say that P is refined by Q in the failures-divergences model, written P $\sqsubseteq_{FD}$ Q if every divergence of Q is a divergence of P, and every failure of Q is a failure of P. Thus DIV is refined by every other process in this model, so, as a specification, allows arbitrary behaviour.

3. Modelling and analysing a synchronous channel

In this section, we consider the operation of a single synchronous channel. We start by considering just the send and receive operations. For the moment, we elide the parts of the implementation related to timed

```
def send(x: A) = lock.mutex{
  while(status != Empty) slotEmptied.await()
  value = x; status = Filled; slotFull.signal()
  continue.await(); assert(status == Read)
  status = Empty; slotEmptied.signal()
}
```

```
def receive(): A = lock.mutex{
  while(status != Filled) slotFull.await()
  status = Read; continue.signal()
  value
}
```

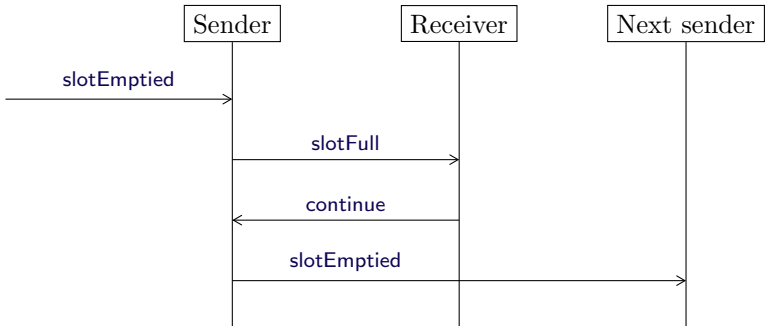


Fig. 1. The code for sending and receiving, and a sequence diagram illustrating signalling on conditions within a channel implementation.

operations, the closing of channels, and alts: we describe a stripped-down implementation with those aspects removed. Note: this is a large simplification of the full code: our main aim here is to illustrate the modelling and analysis technique. We outline the implementation of the send and receive operations, how we model them in CSP, and then how we analyse their correctness. Recall that channels are shared: multiple senders and receivers can compete to use the channel.

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

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, which is one of the following values:

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.

Figure 1 gives code for sending and receiving. Each operation acts under mutual exclusion on lock (denoted "lock.mutex{...}"). The figure also gives a sequence diagram to illustrate the use of conditions. The sender waits on the condition slotEmptied until the status is Empty. It then stores its value in value, sets the status to Filled, and signals on the condition slotFull to the receiver. It next waits on the condition continue until the status is Read. Finally, it sets the status back to Empty, and signals on slotEmptied to the next sender. The receiver waits on slotFull until the status is Filled. It then sets the status to Read, signals to the sender on continue, and returns the value stored in value.

3.1. Basic CSP model

We now outline the CSP model; Figure 2 gives an overview. The model uses small fixed types Data representing the type of data communicated by the channel, and ThreadID representing the type of thread identities. (We discuss the choices for these types in Section 8.)

We model each shared variable 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.

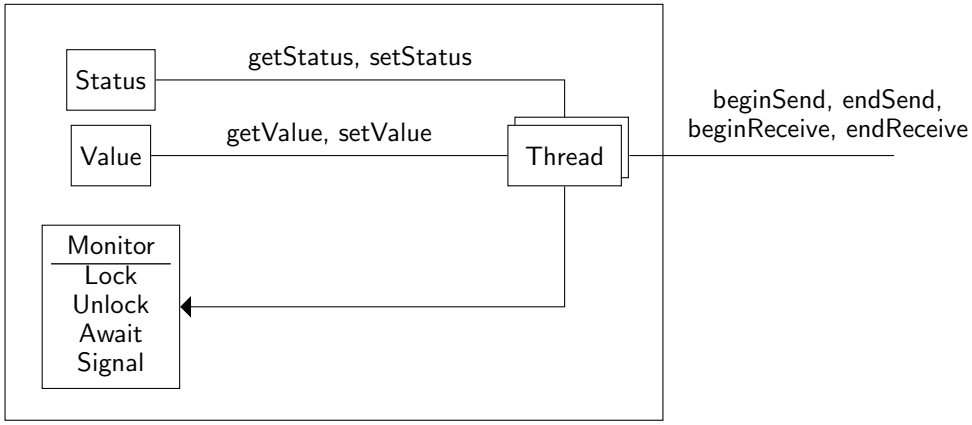


Fig. 2. Representation of the CSP model of a basic channel. Channel communications are represented by lines labelled with the channel names; calls of functions provided by the Monitor module are represented by a line with a triangular arrowhead.

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

We can then model the value and status variables is follows. (We initialise value to a particular value A.)

```
channel getValue, setValue: ThreadID.Data
Value = Var(A, getValue, setValue)
datatype StatusT = Empty | Filled | Read
channel getStatus, setStatus : ThreadID . StatusT
Status = Var(Empty, getStatus, setStatus)
```

The monitor lock is modelled by a CSP module Monitor (adapting the approach from [Zha22]). This encapsulates a process that maintains the state of the monitor, defined recursively in terms of the following two families of processes:[3]

```
Unlocked :: (⦇ConditionID ⇒ {ThreadID}⦈) → Proc
Locked :: (ThreadID, ⦇ConditionID ⇒{ThreadID}⦈) → Proc
```

(We omit here some details, which we will need to model timed operations in Section 4.2.) The process Unlocked(waiting) represents that the monitor is currently unlocked; the map waiting records, for each condition c, the set of threads currently waiting on c. The process Locked(t, waiting) represents that the monitor is currently locked by thread t; waiting again records which threads are waiting on conditions. In the Unlocked states, a thread t can acquire the lock, via event acquire.t (which moves the monitor process into a Locked state). In the Locked states, the current thread t can

- Release the lock, via event release.t (which moves the monitor process into an Unlocked state);
- Wait on a condition c, via event await.t.c (which moves the monitor process into an Unlocked state, updating the waiting map appropriately);
- Signal on a condition c, via event signal.t.c.t', to a thread t' currently waiting on c, if there is one (waiting is updated appropriately); the thread t' needs to reacquire the lock before it can continue.

However, for convenience, the CSP module exports four functions which can be called by a thread t:

- Lock(t): acquire the lock, blocking until it is available;
- Unlock(t): release the lock;

[3] The type ConditionID contains an identity for each of the conditions described earlier. The type ⦇ConditionID => {ThreadID}⦈ represents maps from condition identities to sets of thread identities.

```
Send(me, x) = Monitor::Lock(me); Send1(me, x)

Send1(me, x) =
  getStatus.me?s →
  if s = Empty then Send2(me, x) else (Monitor::Await(me, SlotEmptied); Send1(me, x))

Send2(me, x) =
  setValue.me.x → setStatus.me.Filled → Monitor::Signal(me, SlotFull);
  Monitor::Await(me, Continue); getStatus.me?s → (if s = Read then SKIP else DIV);
  setStatus.me.Empty → Monitor::Signal(me, SlotEmptied);
  Monitor::Unlock(me); endSend.me.SendSuccess → SKIP

Receive(me) = Monitor::Lock(me); Receive1(me)

Receive1(me) =
  getStatus.me?s → if s ≠ Filled then Monitor::Await(me, SlotFull); Receive1(me) else Receive2(me)

Receive2(me) =
  setStatus.me!Read → Monitor::Signal(me, Continue);
  getValue.me?v → (Monitor::Unlock(me); endReceive.me.ReceiveSuccess.v → SKIP)

ChannelThread(me) =
  beginSend.me?x → Send(me, x); ChannelThread(me)
  □ beginReceive.me → Receive(me); ChannelThread(me)
```

Fig. 3. CSP processes modelling the send and receive operations, and a ChannelThread process that performs the operations.

- Await(t,c): wait on condition c; when a signal is received, wait to reacquire the lock;
- Signal(t,c): signal to a thread on condition c (if there is a waiting thread).

Each of the send and receive operations can then be straightforwardly translated into a CSP process: see Figure 3. This process performs the operations on the monitor and variables, following the Scala code. Each expression of the form Monitor::Op(args) represents a call to Op(args) within the Monitor module, which models lock.

The Scala code uses as assertion. This is translated into a process that diverges if the property does not hold. In Section 3.2, we use FDR to verify that such divergences do not occur.

Each operation is framed by events that represent the operations being called or returning:

- The event beginSend.t.x represents a thread with identity t calling send(x) on the channel, and the event endSend.t.SendSuccess represents it successfully returning (later we add events 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 an execution of the receive operation that successfully receives the value x.

We refer to these collectively as begin and end events.

The ChannelThread process in Figure 3 accepts the begin events, simulates the operation, and then performs the appropriate end event.

We construct the model of a channel, as in Figure 2. This model is encapsulated inside a CSP module, so that only the begin and end events are visible outside the module. A process outside the module can simulate calling the operation by synchronising on the begin and end events; later, we capture correctness properties in terms of those events.

Appendix A gives techniques for improving the structure of such models, for example to model **while** loops. We do not use these techniques here, in the interests of brevity, and because they are tangential to the main ideas of this paper.

3.2. Synchronisation linearisation

In this section, we describe, abstractly, the property that we expect to hold of the basic model of the channel. In the next section, we describe how to capture this property using CSP and FDR.

Each successful execution of the send operation should synchronise with a corresponding execution of receive, and vice versa: the two executions should overlap in time, and the receive should return the value of the send's parameter.

In [LL25], we introduced a correctness condition, *synchronisation linearisation*. This property applies generally to *synchronisation objects*, i.e. objects that allow two or more threads to synchronise, and possibly to exchange data; examples include a synchronous channel, the combination of an alt and associated channels, a barrier synchronisation object (e.g. [And91]), or an exchanger (e.g. [HS12]). Below, we specialise the definition of synchronisation linearisation to the case of a channel. (Synchronisation linearisation is based upon *linearisation* [HW90], the standard correctness condition for concurrent datatypes.)

A *history* (or trace) of a channel is a sequence of begin and end events, as in the previous subsection, corresponding to threads using the channel. It is convenient to label each event with an *execution identifier*, so that a begin and end event that correspond to the same execution of an operation have the same execution identifier; for example

$$h = \langle \mathsf{beginSend}^1.t_1.3,\ \mathsf{beginReceive}^2.t_2,\ \mathsf{endReceive}^2.t_2.\mathsf{ReceiveSuccess}.3,\ \mathsf{endSend}^1.t_1.\mathsf{SendSuccess}\rangle.$$

We consider only valid histories, where each identifier appears on at most one begin event and at most one end event, and where the identifier on each end event matches that on an earlier begin event. We say that a history is *complete* if for every begin event, there is a corresponding end event, i.e. there are no pending operation executions.

The idea is that each synchronisation should appear to take place between the beginning and end of the two corresponding operation executions; 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*. Note that these synchronisation points are an abstract specification technique, rather than representing concrete steps of the code. We use events of the form $\mathsf{sync}^{i,j}.s.r.x$ to represent a synchronisation between an execution of send with execution identifier i by thread s, and an execution of receive with execution identifier j by thread r, passing data value x. A *synchronisation history* is a sequence of such sync events. Each execution identifier must appear at most once in the history. For example,

$$h_s = \langle \mathsf{sync}^{1,2}.t_1.t_2.3\rangle$$

is a synchronisation history that describes the (single) synchronisation in the earlier channel history h.

In the case of a basic synchronous channel, every such sequence of sync events is legal. However, for some synchronisation objects, only certain histories are legal. For example, when we consider closing of channels, we will require that all successful synchronisations precede the closing of the channel. In general, we identify a particular (prefix-closed) set of histories as being *legal*, based on the informal specification of the synchronisation object.

The following definition describes when a history and synchronisation history use the same execution identifiers.

Definition 1. We say that a complete history h of the channel, and a legal synchronisation history h_s *correspond* if:

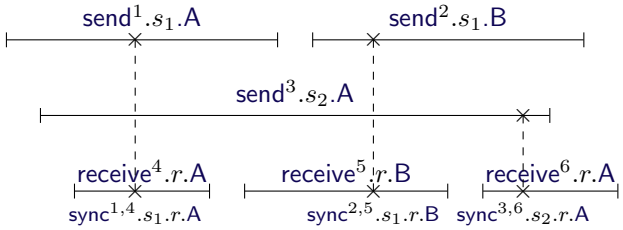
- For every sync event with identifier (i, j) in h_s, h contains an execution of send with identifier i, and an execution of receive with identifier j;
- For every execution of send with identifier i in h, h_s contains a sync event with identifier (i, j), for some j;
- For every execution of receive with identifier j in h, h_s contains a sync event with identifier (i, j), for some i.

The earlier example histories h and h_s correspond.

Definition 2. Let h be a complete history of the channel, and h_s a legal synchronisation history, such that h and h_s correspond. We say that h_s is a *synchronisation linearisation* of h if there is some way of interleaving h and h_s such that each $\mathsf{sync}^{i,j}.s.r.x$ event is between events $\mathsf{beginSend}^i.s.x$ and $\mathsf{endSend}^i.s.\mathsf{sendSuccess}$, and between events $\mathsf{beginReceive}^j.r$ and $\mathsf{endReceive}^j.r.\mathsf{ReceiveSuccess}.x$.

The earlier example synchronisation history h_s is a synchronisation linearisation of the example h: the event of h_s can be inserted after the first two events of h to produce a suitable interleaving.

The diagram below gives a larger example; it illustrates a particular history of a channel and a synchronisation linearisation.



Time goes from left to right in the diagram. Each horizontal line represents an operation execution, with the end points representing the begin and end events. A corresponding synchronisation history is written at the bottom, where each pair of "×"s, linked by a dashed vertical line, illustrates a possible synchronisation point of the corresponding executions, and so defines the interleaving of the histories.

We considered only complete histories above. We now generalise. An *extension* of a (not necessarily complete) history h is formed by adding zero or more end events corresponding to pending executions. We write $complete(h)$ for the subsequence of h formed by removing all begin events of pending operation executions.

Definition 3. Let h be a (not necessarily complete) history of the channel, and h_s a legal synchronisation history. We say that h_s is a *synchronisation linearisation* of h if there is an extension h' of h such that h_s is a synchronisation linearisation of $complete(h')$. We say that h is synchronisation linearisable in this case. We say that a synchronous channel is synchronisation linearisable if all of its histories are synchronisation linearisable.

Informally, the end events that are in h' but not in h correspond to executions that have synchronised but not yet returned; the begin events that are in h but not in $complete(h')$ correspond to executions that have not synchronised.

The definition is based on *linearisation* [HW90], the standard correctness property for concurrent datatypes, where executions of operations should appear to take place in a one-at-a-time order, each between the beginning and end of that execution. However, the two notions are distinct: informally, synchronisation linearisation requires that operation executions appear to take place in a two-at-a-time order. More precisely, synchronisation linearisation requires corresponding executions to overlap in time. A history where, say, a send returns before the corresponding receive is invoked would not represent a correct synchronisation. This overlapping property cannot be captured directly with standard linearisation [LL25].

3.3. Model checking synchronisation linearisability

We now describe how we use model checking to test synchronisation linearisability of the channel. (We drop the execution identifiers from events: they were just convenient in defining synchronisation linearisation.)

We create an instance C of the channel module from Figure 2:

```
instance C = SyncChan(ThreadID, Data)
```

where ThreadID is a type of thread identifiers, and Data is a type of data passed on the channel. We create processes that represent threads: each thread repeatedly calls the send and receive operations, and waits for them to return:

```
Thread(me) =
  C::beginSend.me?x → C::endSend.me?res → Thread(me)
  □ C::beginReceive.me → C::endReceive.me?res → Thread(me)
Threads = ||| t : ThreadID • Thread(t)
```

We build a process System that combines the threads with the channel (the function runWithAndHide from

the channel module runs its argument in parallel with the module's internal processes, and hides the internal events):

$$\mathsf{System} = \mathsf{C::runWithAndHide(Threads)}$$

We now build a CSP specification process that allows precisely the traces that are synchronisation linearisable. As above, 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 *linearisers* process for each thread as follows, which ensures that the sync events occur between the corresponding begin and end events (the approach is based on that for standard linearisation [Low17]).

$$\begin{aligned}&\mathsf{Lin(t)} = \\ &\quad \mathsf{C::beginSend.t?x} \rightarrow \mathsf{sync.t?other!x} \rightarrow \mathsf{C::endSend.t.SendSuccess} \rightarrow \mathsf{Lin(t)} \\ &\quad \Box\ \mathsf{C::beginReceive.t} \rightarrow \mathsf{sync?other!t?x} \rightarrow \mathsf{C::endReceive.t.ReceiveSuccess.x} \rightarrow \mathsf{Lin(t)}\end{aligned}$$

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 that thread's value x; this x is subsequently returned by the operation execution.

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

$$\begin{aligned}&\mathsf{alphaLin(t)} = \\ &\quad \{|\mathsf{C::beginSend.t, C::endSend.t, C::beginReceive.t, C::endReceive.t}|\} \cup \\ &\quad \{|\mathsf{sync.t.other, sync.other.t} \mid \mathsf{other} \leftarrow \mathsf{ThreadID} - \{\mathsf{t}\}|\} \\ &\mathsf{Spec_0} = \|\ \mathsf{t} \leftarrow \mathsf{ThreadID} \bullet [\mathsf{alphaLin(t)}]\ \mathsf{Lin(t)}\end{aligned}$$

Thus each trace of $\mathsf{Spec_0}$ represents an interleaving of a history of the channel with a synchronisation history, as required by Definition 2. For example

$$\langle \mathsf{beginSend.T1.3},\ \mathsf{beginReceive.T2},\ \mathsf{sync.T1.T2.3},\ \mathsf{endReceive.T2.ReceiveSuccess.3},\ \mathsf{endSend.T1.SendSuccess}\rangle$$

is a trace of $\mathsf{Spec_0}$, since the restriction to the events of T1 is a trace of Lin(T1), and the restriction to the events of T2 is a trace of Lin(T2). On the other hand, the trace

$$\langle \mathsf{beginSend.T1.3},\ \mathsf{sync.T1.T2.3}, \mathsf{endSend.T1.SendSuccess}\rangle$$

is not a trace of $\mathsf{Spec_0}$: the sync event would be blocked by Lin(T2).

By hiding the synchronisation points, we obtain a process whose traces represent precisely those histories that are synchronisation linearisable:

$$\mathsf{Spec} = \mathsf{Spec_0} \setminus \{|\mathsf{sync}|\}$$

For example, Spec would allow the correct trace

$$\langle \mathsf{beginSend.T1.3},\ \mathsf{beginReceive.T2},\ \mathsf{endReceive.T2.ReceiveSuccess.3},\ \mathsf{endSend.T1.SendSuccess}\rangle,$$

but not the incorrect trace

$$\langle \mathsf{beginSend.T1.3},\ \mathsf{endSend.T1.SendSuccess}\rangle.$$

We can thus use FDR to check that the system is synchronisation linearisable as follows:

$$\mathsf{assert\ Spec} \sqsubseteq_T \mathsf{System}$$

This tests succeeds. (Fuller results, for the complete model of the channel, are given in Section 4.2.)

In particular, we can perform these tests with multiple threads, and so include cases where multiple threads access the same port concurrently. Hence the analysis shows the channel works correctly when shared.

3.4. Synchronisation progressibility

We now consider our progress property, *synchronisation progressibility* [LL25]. This property requires that if a synchronisation is possible, some such synchronisation will happen, and the threads become able to return. If several alternative synchronisations are possible, then synchronisation progressibility allows any to happen. Informally, threads do not get stuck if there are synchronisations that could occur. For example, if three threads call, respectively, send(4), send(5) and receive on the channel, the property requires that the receiver synchronises with one of the senders, and those two threads return.

The following definition captures a maximal sequence of synchronisations that might occur, given a particular history of the channel.

Definition 4. Given a history h of a channel, and a legal synchronisation history h_s, we say that h_s is a *maximal synchronisation linearisation* of h if: (a) h_s is a synchronisation linearisation of h; and (b) no proper extension $h_s{}^\frown\langle e\rangle$ is a synchronisation linearisation of h.

For example, the history

$$h = \langle \mathsf{beginSend}^1.s_1.4,\ \mathsf{beginSend}^2.s_2.5,\ \mathsf{beginReceive}^3.r\rangle$$

has two maximal synchronisation linearisations

$$h_s^1 = \langle \mathsf{sync}^{1,3}.s_1.r.4\rangle, \qquad \text{and} \qquad h_s^2 = \langle \mathsf{sync}^{2,3}.s_2.r.5\rangle,$$

corresponding to the two possible synchronisations. Each describes one possibility for *all* the synchronisations that might happen.

The following definition captures the end events that we would expect to happen given a particular sequence of synchronisations.

Definition 5. Given a history h of the synchronisation object and a maximal synchronisation linearisation h_s, we say that an end event is *anticipated* if it does not appear in h, but the corresponding sync event appears in h_s.

For the above histories h and h_s^1, the events $\mathsf{endSend}^1.s_1.\mathsf{SendSuccess}$ and $\mathsf{endReceive}^3.r.\mathsf{ReceiveSuccess}.4$ are anticipated: assuming h_s^1 describes the synchronisations that happen, we would expect those end events to eventually happen; if they do not, that is a failure of progress.

Definition 6. Let h be a history of a channel. We say that h is *synchronisation progressible* if, from the state reached after h, for every execution following h with no new begin events, there is a maximal synchronisation linearisation h_s of h such that every anticipated end event e eventually happens. We say that a channel is synchronisation progressible if all of its histories are synchronisation progressible.

For the earlier history h, synchronisation progressibility requires that either $\mathsf{endSend}^1.s_1.\mathsf{SendSuccess}$ and $\mathsf{endReceive}^3.r.\mathsf{ReceiveSuccess}.4$ eventually happen (corresponding to h_s^1), or $\mathsf{endSend}^2.s_2.\mathsf{SendSuccess}$ and $\mathsf{endReceive}^3.r.\mathsf{ReceiveSuccess}.5$ eventually happen (corresponding to h_s^2): one of the synchronisations should happen, and then the relevant operations should return.

On the other hand, for the history $\langle \mathsf{beginSend}^1.s.4\rangle$, the only maximal synchronisation linearisation is $\langle\rangle$, for which there are no anticipated returns, and so synchronisation progressibility is vacuously satisfied: it is fine for the send to get stuck in this case, since there is no receive with which it could synchronise.

Any claim of synchronisation progressibility necessarily makes assumptions about the way scheduling is performed. For the channel considered here, it is enough to assume that the scheduler schedules *some* thread whenever there is a thread that can take a step. Clearly, if no thread is scheduled, none will return. Any reasonable scheduler satisfies this property. When we consider alts in Section 5, we will need a slightly stronger assumption.

The property of *lock freedom* [HS12] requires that eventually some thread returns, assuming the scheduler repeatedly schedules threads. However, this condition treats a thread that is blocked, trying to obtain a lock, as performing infinitely many steps: thus such a system will not be lock-free, since the blocked thread could be scheduled infinitely, without any operation returning. Synchronisation progressibility is not the same as lock freedom. Indeed, SCL channels are not lock-free, because of the use of a lock. We treat a blocked thread as performing no steps (unlike lock freedom, but reflecting how most locks are implemented). Under our scheduling assumption, the thread holding the lock would eventually be scheduled and so release the lock, allowing other threads to progress.

The property of *wait freedom* requires that if a particular thread is repeatedly scheduled, it eventually returns. The property of *obstruction freedom* requires that if a thread executes in isolation (i.e. no other thread is scheduled), then it eventually returns. SCL channels are neither wait-free not obstruction-free, for the same reason as above. Hence synchronisation progressibility is also distinct from these two properties.

We can test our CSP model for synchronisation progressibility by repeating the earlier refinement test, except in the failures-divergences model (which implies the refinement in the traces model):

assert Spec $\sqsubseteq_{FD}$ System

Note that this test requires that the system cannot diverge, and so must eventually reach a stable state. The failures-divergences model records refusal information only in stable states. This reflects our assumption about scheduling: unstable states are transient, because there must be some internal step by a thread that is possible and that will be scheduled. If a synchronisation is possible, then the specification will reach a state where relevant end events are available (i.e. not refused). The test, then, requires that in stable states, the system also makes those end events available. If, in fact, there are two or more possible synchronisations, the specification will choose nondeterministically which to perform, and so the system may likewise perform any such synchronisations.

Finally, note that the fact that System does not diverge verifies that no thread in the implementation throws an exception.

It turns out that our specification is equivalent to that of Welch and Martin [WM00a] when restricted to non-shared channels (which they consider). However, we consider our specification to be an instance of a more general pattern, for synchronisation objects, rather than a single-purpose specification.

4. Extending basic channels

We now extend the basic model of channels to allow channels to be closed, and to include timed operations (but still omitting interactions with alts).

4.1. Closing channels

Recall that a channel can be closed by the close operation, and that subsequently a send or receive execution fails and throws a Closed exception.

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 A and B should synchronise, so both return successfully, or both should throw Closed exceptions; if the two threads have different views, this may lead to logical errors, for example thread A doing something with the value x that it apparently failed to send.

The SCL 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 (the signalAll operation on a condition c signals to all threads currently waiting on c).

```
def close = lock.mutex{
  if(!isChanClosed){isChanClosed = true; slotEmptied.signalAll(); slotFull.signalAll(); continue.signalAll()}
}
```

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

Figure 4 gives code to show how send and receive deal with the channel being closed. (Note: this is still a large simplification of the full code, which supports alts and the timed operations.) 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 described earlier: thus we prioritise completing the communication over checking whether the channel has been closed (this seems necessary for correct interaction with alts). If status does not hold

```
def send(x: A) = lock.mutex{
  while(status != Empty && !isChanClosed)
    slotEmptied.await()
  if(isChanClosed) throw new Closed
  value = x; status = Filled
  if(receiversWaiting > 0) slotFull.signal()
  continue.await()
  if(status == Read){
    status = Empty; slotEmptied.signal() }
  else{
    assert(isChanClosed)
    if(receiversWaiting == 0) throw new Closed }
}
```

```
def receive(): A = lock.mutex{
  if(isChanClosed) throw new Closed
  while(status != Filled){
    receiversWaiting += 1; slotFull.await()
    receiversWaiting -= 1
    if(status != Filled && isChanClosed)
      throw new Closed
  }
  status = Read; continue.signal(); value
}
```

Fig. 4. Code for deadling with closed channels when sending and receiving.

Filled, the thread 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 continues as described earlier. Otherwise, it must be the case that isChanClosed has been set (the SCL implementation asserts this, and the CSP 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 thread throws a Closed exception.

The precise order of checks in the previous paragraphs is rather subtle. This is where the earlier implementation went wrong. Previously, when the waiting thread received a signal, it *first* checked isChanClosed, and if it was set, threw a Closed exception. This could be wrong for a couple of reasons, as illustrated in Figure 5.

- In the top part of the figure, a receiving thread reads and returns the sending thread's value, indicating that it has correctly synchronised. Then the channel is closed. If the send first checks isChanClosed, it will throw a Closed exception. This is an error: either both threads should succeed, or both should see the channel had closed, and fail. The correct version (Figure 4) instead reads status first, and returns correctly.
- The bottom part of the figure is similar, but the channel is closed and the send continues *before* the receive reads the sender's value. If the send first checks isChanClosed (and maybe status), it will throw a Closed exception, even though the receive will subsequently return the sender's value. This is again an error. The correct version instead finds receiversWaiting > 0, and so returns correctly.

This correct version was found with the help of the model checking described below.

Adapting the CSP model to model closing of channels is straightforward. The endSend and endReceive channels are extended to allow a result 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 System process from earlier to also allow threads to close channels.

The channel specification in Section 3.2 is (in the terminology of [LL25]) *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.) The definition of synchronisation linearisation requires the synchronisation history to be consistent with the state.

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 a CSP event close.t to represent the linearisation point of a close operation by thread t. Further,

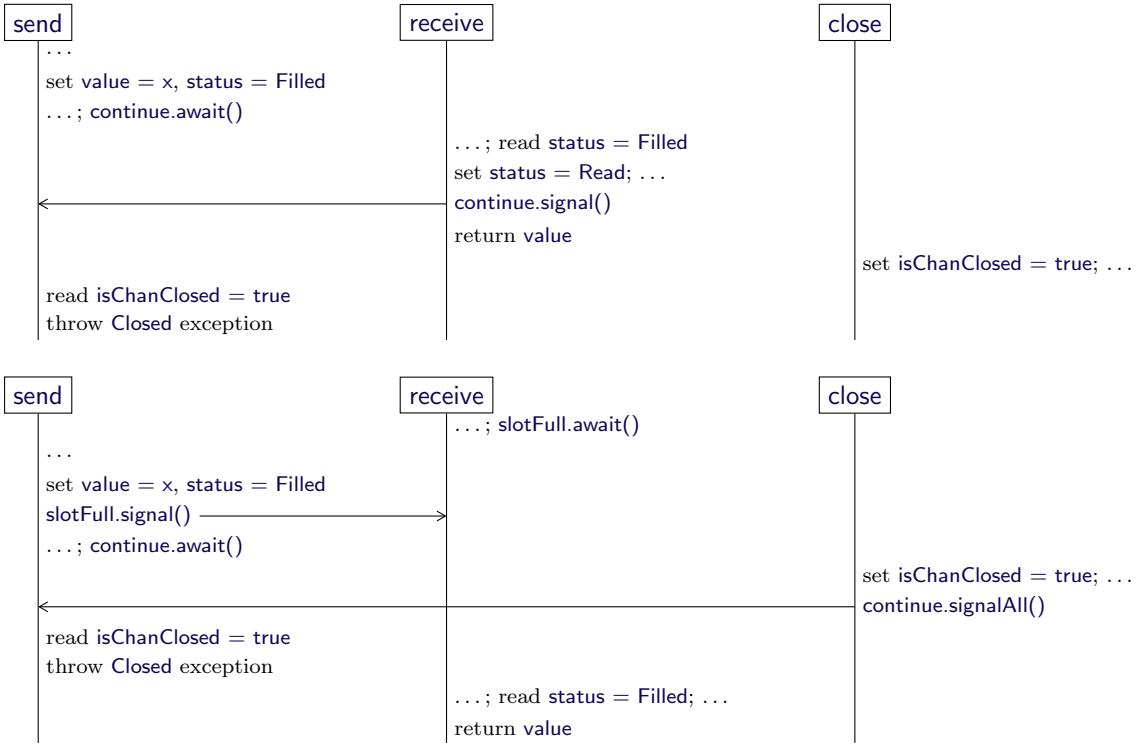


Fig. 5. Illustration of errors that arise in the earlier implementation. Several steps are elided, for brevity.

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 that the channel is closed.

Within the specification, the state of the channel is recorded by the process ChanSpec (Figure 6). 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).

We adapt the lineariser processes to reflect the closing of channels (also Figure 6). A sending thread can either synchronise with another thread, as before, or find that 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 beginClose and endClose events.

We combine the linearisers as before (with suitably extended alphabets), and synchronise them with ChanSpec on the relevant events, to produce Spec_0. We hide the internal events to produce the main specification Spec. The construction is illustrated in Figure 7 in the case of two threads.

For example, Spec_0 allows traces such as

$$\langle \mathsf{C::beginSend.T1.A},\ \mathsf{C::beginReceive.T2},\ \mathsf{C::beginClose.T3},\ \mathsf{sync.T1.T2.A},\ \mathsf{close.T3},\ \mathsf{C::endSend.T1.SendSuccess},\ \mathsf{C::endReceive.T2.ReceiveSuccess.A},\ \mathsf{C::endClose.T3}\rangle.$$

Here T1 is trying to send, T2 is trying to receive, and T3 is trying to close the channel; the synchronisation between T1 and T2 happens before the close has an effect, so the threads successfully communicate. But Spec_0 also allows traces such as

$$\langle \mathsf{C::beginSend.T1.A},\ \mathsf{C::beginReceive.T2},\ \mathsf{C::beginClose.T3},\ \mathsf{close.T3},\ \mathsf{isClosed.T1},\ \mathsf{isClosed.T2},\ \mathsf{C::endSend.T1.Closed},\ \mathsf{C::endReceive.T2.Closed},\ \mathsf{C::endClose.T3}\rangle.$$

Here the channel is closed before T1 and T2 can synchronise, so both return the Closed value.

Thus Spec allows all traces, containing the begin and end events, that are synchronisation linearisable. So testing $\mathsf{Spec} \sqsubseteq_T \mathsf{System}$ verifies synchronisation linearisability for this system. Performing the corresponding

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

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)
  □ isClosed.t → endChan.Closed → Lin(t)

AllLins = ||| t ← ThreadID • [alphaLin(t)] Lin(t)
Spec0 = AllLins [| alphaChanSpec |] ChanSpec
Spec = Spec0 \ alphaChanSpec
```

Fig. 6. The specification process for closeable channels.

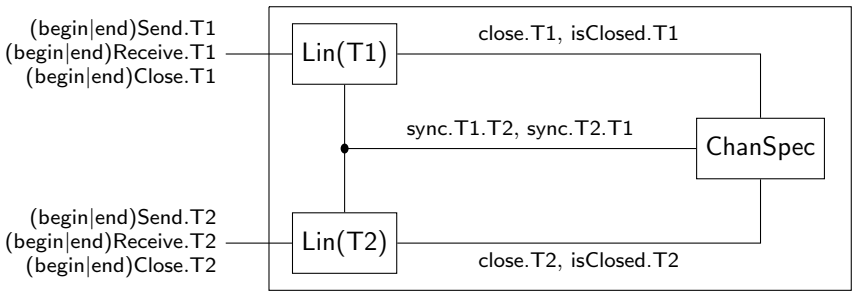


Fig. 7. Construction of the specification process, with two threads. We use BNF-style notation to capture channels with similar names; for example, we write "(begin|end)Send" to denote the beginSend and endSend channels.

test in the stable failures model also verifies synchronisation progressibility. 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.

4.2. Timed operations

We now consider the timed send and receive operations on channels.

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(duration)(x) 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 timed out, 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 then 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 still 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

timed out, 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 then 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 timed out, so returns None.

We now describe the CSP model of these operations. Our model and subsequent analysis do 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, described in Section 3.1, also models timed waits, via a function TimedAwait. A thread that calls this function might receive a signal, as for untimed waits. In addition, it can time out. Thus we model that such threads can eventually time out, but, as noted above, 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 (again abstracting away from the length of the delay). We then adapt the definition of the lineariser processes for these operations as follows, adding the possibility of such a time out to the possibilities of the untimed operations.

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

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

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

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.

The table to the right gives statistics about the number of states explored and the times taken to perform these checks, in different models, and for different numbers of threads. Each test used two data values (and this is the case for all later checks reported in this paper). All experiments in this paper were performed on a 32-core server (two 2.1GHz Intel(R) Xeon(R) Gold 6130 CPUs with hyperthreading enabled, with 512GB of RAM). The check with 5 threads in the failures-divergences model was beyond the limits of this machine. As is normally the case, the state space, and hence the checking time, grows rapidly with the number of threads.

Model	Threads	States	Time
FD	3	1.23M	3.6s
FD	4	92.5M	43s
F	5	7.22B	34min

5. Alts

We now consider alts.

Recall that an alt has a number of branches. Each in-port branch is associated with a boolean guard, an in-port from which the alt is willing to receive a value, and a continuation function that is applied to a value received. Each out-port branch is associated with a boolean guard, an out-port to which the alt is trying to send a value, an expression defining the value to be sent, and a continuation to be executed after the value is sent.

We call the thread that is executing the alt the *alt-thread*, and a thread performing a send, receive or close operation in a channel a *channel-thread*.

We start by describing the high-level interactions between an alt and channels, and how those interactions are implemented within alts and channels. We then outline how these interactions are modelled in CSP, and describe a direct analysis of an alt and associated channels.

5.1. High-level design

An alt and its channels interact by calling operations on each other. We call the way they interact the *alt protocol*. The alt-thread calls (package-private) operations upon relevant channels; and likewise, a channel-thread may call (package-private) operations upon relevant alts. When the alt-thread calls an operation on a channel, it uses the channel's lock, so as to provide mutual exclusion; similarly, when a channel-thread calls an operation on the alt, it uses the alt's lock.

A successful execution of an alt goes through up-to four stages:

Registration: the alt-thread informs each port that it is willing to communicate; if the port is also willing to communicate, the communication takes place at this point.

Waiting: (omitted if a communication took place during the registration phase) the alt-thread waits for a callback from a port to indicate that it is now willing to communucate.

Deregistration: the alt-thread informs the other ports that it is no longer willing to communicate.

Execution: the continuation of the relevant branch is executed.

We describe these in more detail below.

When an alt-thread executes an alt, it starts by locking the alt. It then registers, in turn, with each of the relevant ports whose guard evaluates to true. More precisely, the alt-thread calls an operation

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

on each of its in-ports whose guard is true, where alt is a reference to the calling alt, and index is the index of the branch within the alt.[4] The operation returns 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; the communication took place, and the alt has received x from the port;

RegisterInWaiting: the port is not currently willing to communicate (the port stores the details of the registration in this case);

RegisterInClosed: the port has been closed.

Similarly, the alt-thread calls an operation

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

on each of its out-ports whose guard is true, where alt and index are as for registerIn, and value is a computation that, when evaluated, produces the value to be sent. The operation returns a result of one of the following forms:

RegisterOutSuccess: the port was willing to communicate; the communication took place, and the alt sent a value to the port;

RegisterOutWaiting: the port is not currently willing to communicate (the port stores the details of the registration);

RegisterOutClosed: the port has been closed.

If one of the registrations is successful, the alt-thread releases the lock on the alt (we explain the reason for this below). It then deregisters from the waiting branches, via operations deregisterIn and deregisterOut; this tells the port that the alt is no longer willing to communicate (and the channel removes the details of the registration). The alt-thread then executes the continuation of the successful branch.

Figure 8 gives an example: an alt first registers unsuccessfully with an in-port *IP*; then it registers successfully with an out-port *OP*; and finally it deregisters from *IP*.

If no registration is successful, and the alt-thread finds that every port is closed or has a guard that is false, then it throws an AltAbort exception. Otherwise, it waits (releasing the lock on the alt) for a callback from one of the ports with which it is registered. The callback is of one of the following forms.

[4] In the implementation, there is an additional parameter iter that represents an iteration number when an alt is executed repeatedly; other operations described in this section have a similar parameter. It is used only in assertions, checking that both components have the same value, and so we omit it from our analysis.

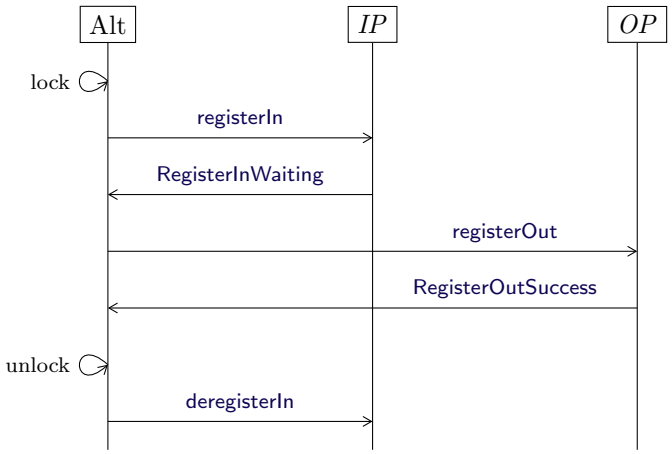


Fig. 8. Sequence diagram illustrating an alt that registers successfully with a port.

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

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

  on the alt, where value is the value it is trying to send to the alt, and index matches the value provided during registration. This asks the alt whether it is still willing to receive from the in-port, and if so completes the communication, storing value in the alt.
- If the alt is registered at the out-port of a channel, and another thread tries to receive on the channel, it calls

  def maybeSend[A](index: Int): Option[A]

  This asks the alt whether it is still willing to send a value to the in-port, and if so completes the communication.
- If another thread closes the channel, it calls

  def portClosed(index: Int)

Each of the callbacks uses the alt's lock, to ensure mutual exclusion. Callbacks are blocked during the registration phase, because the alt-thread holds the alt's lock.

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. The callback operation wakes up the alt-thread, which then deregisters from the remaining branches. Finally, the alt-thread executes the continuation of the successful branch. Figure 9 gives an example of a successful callback of maybeSend from an out-port with which the alt is registered.

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

5.2. Implementation details

We now describe some details of the implementation.

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. As noted above, each callback operation has to obtain this lock, so those operations are blocked until registration is complete.

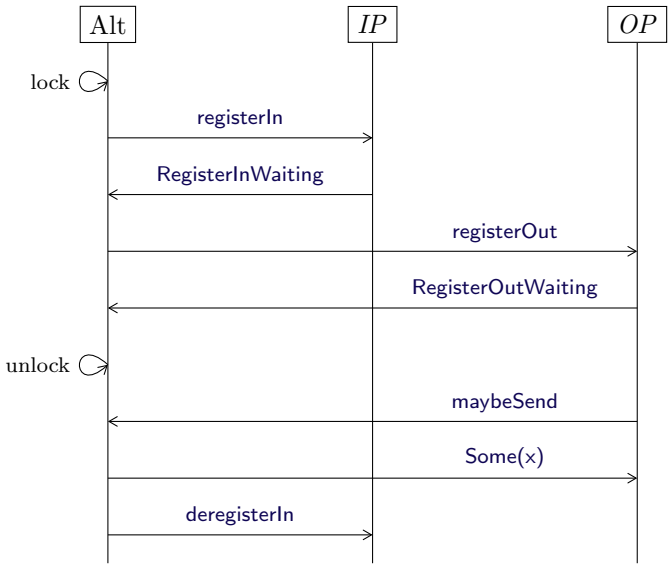


Fig. 9. Sequence diagram illustrating a successful callback from a port.

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 would hold the lock on the alt, so the callback would be blocked. The alt-thread therefore releases the lock during deregistration.

The implementation uses a variable done that is 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. Then, 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.

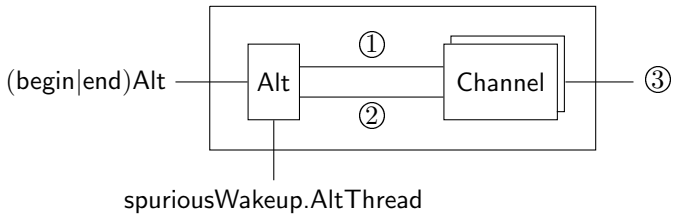
Each channel has variables to store information about alts currently registered there. These are set by the registerIn and registerOut operations, and cleared by the deregisterIn and deregisterOut operations.

Recall from the Introduction that the ports of a channel may not be simultaneously feasible in two alts. The registerIn operation checks whether another alt is currently registered with the channel (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 operation 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 then 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.

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

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



Key.

①: (begin|end)Register(In|Out), (begin|end)Deregister(In|Out);
②: (begin|end)maybe(Send|Receive), (begin|end)PortClosed;
③: (begin|end)Send, (begin|end)Receive, (begin|end)SendWithin, (begin|end)ReceiveWithin, (begin|end)Close.

Fig. 10. The interface of the CSP model of an alt and associated channels. (Notation is as in Figure 7.)

5.3. 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. A process records which thread, if any, currently holds the lock, and which threads are currently waiting for a signal. One difference, however, concerns a bug concerning JVM monitors[5]: 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 alt implementation guards against spurious wakeups by performing a suitable check when resuming after a wait, and waiting again if appropriate. Our CSP model of a monitor reflects the possibility of spurious wakeups: it allows 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 monitor process in parallel with a *regulator process*

$$\mathsf{Reg} = \mathsf{spuriousWakeup}\,?\,\mathsf{t} \rightarrow \mathsf{Reg} \sqcap \mathsf{STOP}$$

(equivalently, $\mathsf{Reg} = \mathsf{CHAOS}(\{|\mathsf{spuriousWakeup}|\})$). The effect of this combination is that spurious wakeups (nondeterminstically) might or might not happen. The latter option is important: spurious wakeups can happen, but they can't be relied on to happen. If we allowed unrestricted spurious wakeups in the model, there is a danger that our analysis would seem to show that suitable progress properties are satisfied, when in fact it is only spurious wakeups that allow progress, and without them the system would get stuck.

We choose not to model the guards of alt branches, for we believe that doing so would add a lot of complexity to the model, and cause FDR checks to take longer, without adding much assurance. The part of the implementation concerning guards is rather straightforward: a branch whose guard is false is simply ignored. We think it is best to concentrate on the more difficult parts of the implementation.

We also do not model the expression that generates the value to be sent in an out-port branch, but just pick the value nondeterministically. Likewise, we do not model the continuations of branches, which are executed after the communication itself. Each continuation could contain arbitrary code (of the correct type); but they are outside the operation of the alt itself. Instead, the model just records the index of the branch selected.

Figure 10 depicts the interface of the model of an alt, and how the interface of the model of a channel is adapted to deal with alts. In the model of a channel, the registration and deregistration operations are framed by 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 operations are framed by 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 by an alt-thread t is framed by events beginAlt.t and endAlt.t.result, where result is of one of the following forms.

[5] https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html#wait--.

- 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.

5.4. 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 collection of branches. Below, we consider an instance A1 of the alt module:

instance A1 = Alt(Alt1, branches, ThreadID, ChanID)

where Alt1 is the name of this alt, ThreadID is the type of thread identifiers, ChanID is the type of channel identifiers, and branches defines the branches of the alt, using a definition such as

branches = <InPortBranch.c1, OutPortBranch.c2>

In different tests, we can vary the number of branches, and whether they correspond to in-ports or out-ports, so test different combinations.

We also create two instances C1 and C2 of the channel module, corresponding to channels with identities c1 and c2. We combine these together in parallel with A1, synchronising on the begin and end events that correspond to the alt calling operations on a channel, or vice versa, as depicted in Figure 10.

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: 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, but recall that this model records refusal information only in *stable* states, where no internal event is possible. We should therefore consider whether the potential divergence is masking possible errors, by making critical states unstable. 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, and so the model does include the stable refusal information we want. This way of abstracting the spurious wakeups corresponds to Roscoe's *lazy abstraction* [Ros10].

This modelling assumes that spurious wakeups do not happen so frequently that they prevent other threads from making progress: thus we can assume that each thread not involved in spurious wakeups can take steps so that it reaches a stable state (either blocked or ready to perform an end event).

We now consider the appropriate correctness condition. This is an adaptation of the synchronisation linearisation and progressibility conditions for a single channel from Section 3. The combination of an alt and associated channels is a synchronisation object that allows several modes of synchronisation: either between the alt-thread and a channel-thread at the other end of a channel; or between two channel-threads (recall that a channel-thread and alt-thread may use a port concurrently). The specifications that follow capture that these synchronisations happen between appropriate begin and end events, and are consistent with the states of channels.

Figure 11 gives an overview of the construction of the specification. We extend the sync events to include the identity of the channel being used: $\mathsf{sync}.t_1.t_2.c.x$ represents 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 $\mathsf{altSync}.t_1.t_2.c.x$ 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.

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

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

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

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

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

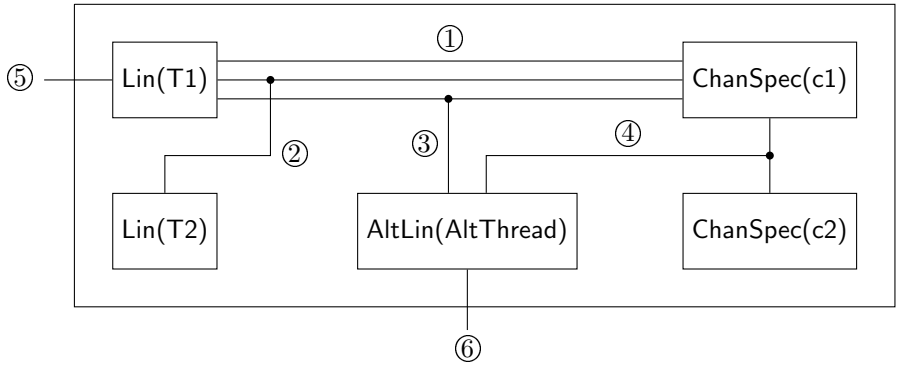
We similarly build a lineariser process for the alt-thread. We introduce an event allClosed which will represent the linearisation point of an alt usage that finds all the channels are closed and so throws an AltAbort. Let inports and outports be the sets of in-ports and out-ports that the alt uses ({c1} and {c2} for the earlier definition of branches); and let the function index give the index of a particular branch (so index(c1) = 0 and index(c2) = 1 for the earlier definition of branches). In the definition below, the first branch of the external choice represents a synchronisation between the alt-thread t (as receiver) and a channel-thread other (as sender) on in-port c passing value x, followed by the alt signalling that it has received x on the relevant branch of the alt. The second branch of the external choice is similar, but where the alt-thread sends x on out-port c. The third branch represents that the alt detects that all the channels are closed, and returns an AltAbort.

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

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 all 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 ChanThreadLin, AltLin and ChanSpec processes in parallel, synchronising on shared events,



Key.

①: close.T1.c1, timeout.T1.c1, isClosed.T1.c1;
②: sync.T1.T2.c1, sync.T2.T1.c1;
③: altSync.AltThread.T1.c1, altSync.T1.AltThread.c1;
④: allClosed;
⑤: C1::(begin|end)Send.T1, C1::(begin|end)Receive.T1, C1::(begin|end)SendWithin.T1, C1::(begin|end)ReceiveWithin.T1, C1::(begin|end)Close.T1;
⑥: A1::(begin|end)Alt.AltThread.

Fig. 11. Construction of the specification process for an alt with two channels and two channel-threads. Notation is as in Figure 7. We omit events for T2 that are symmetric to events for T1, and events for C2 that are symmetric to events for C1, for clarity.

as illustrated in Figure 11, to produce a process Spec_0. For example, Spec_0 allows traces such as the following (based on the earlier definiton of branches):

⟨A1::beginAlt.AltThread, C1::beginSend.T1.A, altSync.T1.AltThread.c1.A,
C1::endSend.T1.SendSuccess, A1::endAlt.AltThread.AltReceive.0.A,
C1::beginClose.T2, close.T2.c1, C1::endClose.T2, C2::beginClose.T2, close.T2.c2, C2::endClose.T2,
A1::beginAlt.AltThread, allClosed, A1::endAlt.AltThread.AltAbort⟩.

Here, alt-thread AltThread receives value A from channel-thread T1 on c1 (first two lines); then thread T2 closes both channels (third line); then AltThread finds all channels are closed and so returns an AltAbort (final line).

Then the specification

$\mathsf{Spec} = \mathsf{Spec}_0 0 \setminus \{|\mathsf{sync}, \mathsf{altSync}, \mathsf{timeout}, \mathsf{close}, \mathsf{isClosed}, \mathsf{allClosed}|\}$

allows just traces (of begin and end events) where every complete operation execution represents a correct synchronisation. We can verify that the system refines this specification in both the traces and stable-failures models, showing that the system is synchronisation linearisable and progressible.

However, this approach suffers from a state-space explosion. The table on the right gives statistics about checks, for divergence freedom (D) and progressibility (F); each test considers an alt with two branches (one in-port branch and one out-port branch) and the corresponding two channels, used by three threads (one alt-thread and two channel-threads). The corresponding tests with four threads were beyond the limits of the machine used.

Model	States	Time
D	854M	872s
F	1.11B	319s

6. 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 is consistent with a more abstract model, which we call IdealisedChannel;

2. We likewise show that an alt is consistent with a more abstract model, which we call IdealisedAlt;
3. We show that the combination of an IdealisedAlt and several IdealisedChannels satisfies the correctness property from the previous section.

This allows us to deduce that the combination of an alt and several channels satisfies the same property. The idea is that the "idealised" models, IdealisedChannel and IdealisedAlt, capture the desired behaviour of the components, abstracting away from their implementations.

The fundamental reason why such an approach works is that CSP refinement is compositional: if $P \sqsubseteq Q$ then $C(P) \sqsubseteq C(Q)$ for any context C defined using CSP operators [Ros10]. This means that if we refine one component of a system, we also refine the whole system. Alternatively, if we show that a component refines some specification, we can replace that component by its specification in the analysis of the whole system.

This approach scales better than that in the previous section, so allows us to deduce correctness for a larger number of threads.

The analysis in step 1 considers the channel in isolation, in the sense that we do not model the alt explicitly. Instead, we treat the alt as part of the environment of the channel. This means we model interactions between the channel and the alt, at the interface, without modelling how the alt deals with those interactions. Likewise, the analysis in step 2 considers the alt in isolation, without modelling the channel explicitly.

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, 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 it may act incorrectly otherwise, for example if it receives a call-back that does not correspond to a current registration.

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.

The issue of components of a system making assumptions about other components has been widely considered before, for example using the techniques of rely-guarantee [Jon83] and assumption-commitment [MC81, PJ91]. Our approach shows how to use CSP model checking in such a situation.

We tackle this issue as follows. Our idealised model of a channel detects if its environment breaks the alt protocol, and if so allows 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 produce a process IdealisedChannel that describes allowed behaviour when the environment does follow the protocol, but produces an *error event* from a set Errors_C if the environment breaks the protocol. We then test (Section 6.1) whether the channel model (from Section 3) refines the process

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

where Any allows arbitrary behaviour. Thus the above process acts like Any after an error event. The definition of Any defends upon the semantic model we are using: in the stable-failures model, the appropriate process is CHAOS(Interface), where Interface is the interface of the channel; in the failures-divergences model, the appropriate process is DIV. Showing that the channel refines ChannelSpec will show that it acts like IdealisedChannel, provided its environment follows the alt protocol.

Likewise, we build a process IdealisedAlt, which describes allowed behaviour of an alt when the environment does follow the protocol, but performs an event from Errors_A if the environment breaks the protocol. We then build a specification of an alt that allows arbitrary behaviour if its environment breaks the protocol (Section 6.2), as follows:

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

Showing that the model of an alt (from Section 5) refines AltSpec will show that it acts like IdealisedAlt, provided its environment follows the alt protocol.

We then consider the combination of IdealisedAlt and several instances of IdealisedChannel, and test (Section 6.3) whether it refines the specification Spec from Section 5.4.

The following proposition generalises this style of compositional verification.

Proposition 7. Let M be a CSP model, either T (traces), F (stable failures), or FD (failures-divergences). Let P_i, for $i \in I$, be a collection of processes, with alphabets α_i. For each i, let Err_i be a set of fresh "error" events (disjoint from each other and from each α_j), let $Idealised_i$ be a process with alphabet $\alpha_i \cup Err_i$, and let

$$Spec_i = (Idealised_i \,[|\, Err_i \,|\rangle\, Any_M) \setminus Err_i,$$

where $Any_T = Any_F = \mathsf{CHAOS}(\alpha_i)$ and $Any_{FD} = \mathsf{DIV}$. Let $Spec$ be a process with alphabet $\bigcup_{i \in I} \alpha_i$ (so without any error events), and let A be a set of events disjoint from $\bigcup_{i\in} Err_i$. Suppose

1. $Spec_i \sqsubseteq_M P_i$, for each $i \in I$;
2. $Spec \sqsubseteq_M \left(\left|\left|\right| i : I \bullet [\alpha_i \cup Err_i]\, Idealised_i\right) \setminus A$.

Then

$$Spec \sqsubseteq_M \left(\left|\left|\right| i : I \bullet [\alpha_i]\, P_i\right) \setminus A.$$

The idea is that, for each i, $Idealised_i$ is an idealised model of P_i, which describes P_i's behaviour under the assumption that its environment satisfies certain assumptions, but performs an error event otherwise; so $Spec_i$ allows arbitrary behaviour when the environment does not follow the assumptions. The final refinement in the proposition is the result we are interested in: the proposition allows us to deduce this from proofs of conditions 1 and 2.

Proof: From condition 2, and the assumption that $Spec$ performs no error event from $\bigcup_{i \in I} Err_i$, we can deduce that $\left|\left|\right| i : I \bullet [\alpha_i \cup Err_i]\, Idealised_i$ performs no error event. But this implies that $\left|\left|\right| i : I \bullet [\alpha_i]\, Spec_i$ likewise performs no (hidden) error event, since the two processes behave identically up to the first error event. So, in fact,

$$\left|\left|\right| i : I \bullet [\alpha_i \cup Err_i]\, Idealised_i = \left|\left|\right| i : I \bullet [\alpha_i]\, Spec_i,$$

since the "throws" operator in each $Spec_i$ is never triggered. Thus

$$Spec \sqsubseteq_M \left(\left|\left|\right| i : I \bullet [\alpha_i \cup Err_i]\, Idealised_i\right) \setminus A = \left(\left|\left|\right| i : I \bullet [\alpha_i]\, Spec_i\right) \setminus A \sqsubseteq \left(\left|\left|\right| i : I \bullet [\alpha_i]\, P_i\right) \setminus A,$$

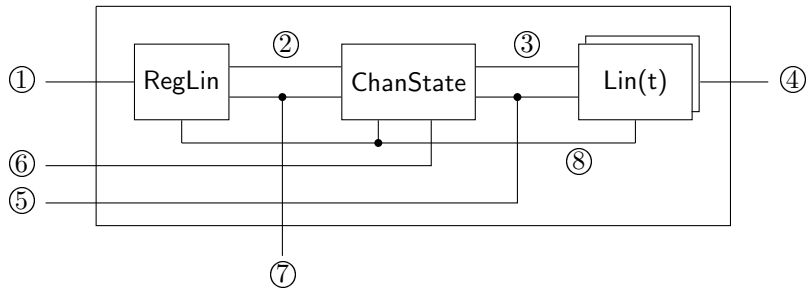
using conditions 1 and 2, and the fact that refinement is compositional with respect to CSP operators. □

We can apply the above proposition, instantiating the P_i with a model of a channel or alt implementation, instantiating the $Idealised_i$ with IdealisedChannel or IdealisedAlt, instantiating the $Spec_i$ with ChannelSpec or AltSpec, and instantiating $Spec$ with the specification from Section 5.4. Condition 1 of the proposition is discharged via the refinement checks, concerning ChannelSpec and AltSpec, described above. Condition 2 can be discharged by combining IdealisedChannel and IdealisedAlt processes, in the same way that we combined channel and alt processes in Section 5.4.

6.1. 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 operations, 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 operation calls, whereas these operations will take effect at their linearisation points. To deal with this, we construct the model from several components, as depicted in Figure 12. 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 component Lin(t) is responsible for linearising the operation calls of channel-thread t. We describe these components in more detail below. Some details of the definitions are not obvious, and were found by trial and error: their correctness is evidenced by the subsequent successful refinement checks.



Key. 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.

Fig. 12. Construction of the idealised channel.

The RegLin process. The RegLin process deals with the linearisation of registration and deregistrations. It is defined in Figure 13; it accepts the relevant begin events, with each operation 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.
- An 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 processes. Each Lin(t) process is responsible for linearising operations of channel-thread t. They are defined in Figure 14. Each accepts the relevant begin events, with most of the operations modelled by subsidiary processes.

The process SendingLin models the linearisation of the send and sendWithin operations; the parameter timed indicates the latter case. The subprocess Success indicates a successful send. There are several cases. The ChanState process synchronises on the relevant events to ensure an appropriate one is selected.

- 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

```
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
```

Fig. 13. Definition of the RegLin process, controlling registration and deregistration.

earlier. Again, the thread t might not be able to return immediately; a synchronisation on commit.t captures the point at which it becomes able to return.

- 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 operation, the thread can time out, modelled by the event timeout.t.

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 operations 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 operation 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. The ChanState process keeps track of the state of the channel. It is defined in Figure 15. The parameter regStatus records the current registration status, and is taken from the following type.

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

where AltID is the type of alt identities, and Index is the type of indices of branches. The subtypes of

```
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)
```

Fig. 14. Definition of the Lin processes, controlling channel-thread operations.

RegStatus represent that no alt is currently registered, or that an alt is registered at the in-port or out-port 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 in-port or out-port, 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, and so have to wait; 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 that 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.

```
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
  )

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

Fig. 15. The ChanState process.

- 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).
- A channel-thread in a sendWithin or receiveWithin can time out.
- 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 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.

Testing the idealised channel. The components of the idealised channel are combined together as illustrated in Figure 12, 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 if the environment has not followed the alt protocol, represented by events 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.

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

We can then compare the CSP model of a synchronous channel implementation against the specifications. 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 $\mathsf{ChannelSpec}_{FD}$ in the relevant models.

The table to the right gives statistics about the number of states explored and the times taken to perform these checks, in different models; in each case, one thread was an alt-thread and the remainder where channel-threads. With five threads, the checks fail to complete on the available architecture.

Model	Threads	States	Time
F	4	236M	569s
FD	4	176M	107s

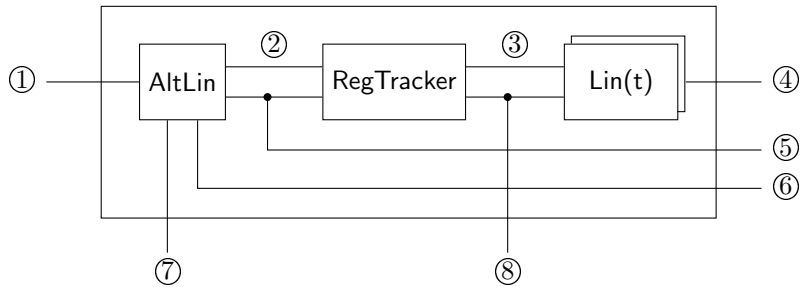
The bottleneck in these checks is the time taken to normalise the specification: this accounts for about 90% of the total in the stable-failures model; checks with more than four threads get stuck at this point. This step builds an automaton equivalent to the specification, but with the property that after each trace (of visible events), a unique state is reached; if, after trace tr, the specification can reach states $st_1, \ldots, st_k$, then the normalised automaton contains a corresponding state equivalent to $st_1 \sqcap \ldots \sqcap st_k$. The idealised channel is a complex process, with much internal nondeterminism, so normalising it is slow.

Interestingly, the failures-divergences check is faster than the stable-failures model: normally, it is the other way round. The difference is due to the time taken to normalise the specification. Consider the case where a trace tr might have included a (hidden) error event. In the failures-divergences model, the resulting normalised state after tr includes DIV, and so is equal to DIV: the normalisation algorithm identifies this, and so does not need to expand its successor states. However, in the stable-failures model, the algorithm does not identify that the corresponding state is equivalent to CHAOS(Interface): it does construct the successor states, making normalisation much slower than in the failures-divergences model. It is straightforward to show that the failures-divergences check implies the corresponding stable-failures check, because IdealisedChan is divergence-free.

6.2. Idealised model of an alt

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



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.

Fig. 16. Construction of the idealised alt.

```
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
```

Fig. 17. Definition of the Lin(t) processes.

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 16. The component RegTracker 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 AltLin linearises the main call of the alt by the alt-thread. We describe these components in more detail below.

The Lin processes. The Lin(t) processes, which linearise callbacks by channel-threads, are defined in Figure 17. 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:

```
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)
```

Fig. 18. Definition of the AltLin processes.

the RegTracker process selects the appropriate event, based on whether the alt is currently registered at the relevant port.

The AltLin process. The AltLin process, which linearises the main calls of the alt, is defined in Figure 18. 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, as described by process Register1 (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 deregistration phase.

If the registration phase gets to the end of branches without a successful registration, 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.

```
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

RegTracker3(reg) =
  let inRegs = getInRegs(reg)
      outRegs = getOutRegs(reg)
      allRegs = 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
```

Fig. 19. The RegTracker process.

The RegTracker process. The RegTracker process is defined starting in Figure 19. 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.

Each subsidiary process 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 in-port, registered at an out-port, 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 (calculated using the helper function allClosed); 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 in-ports, registered out-ports, and all registered ports, respectively. The process can send to AltLin information about a port that can be deregistered (channels getToDeregisterIn and getToDeregisterOut), or an indication that there is no such port (channel deregisterDone). It synchronises on the end events of deregistrations, and updates its state to map the relevant index to NoReg.

RegTracker2 can synchronise with Lin(t) on the linearisation point of a maybeReceive operation corresponding to a registered in-port; 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 out-port, 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 deals with the waiting phase, waiting for callbacks from registered ports. It can synchronise with Lin(t) on the linearisation point of a maybeReceive operation corresponding to a registered in-port. 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 out-port; 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.

Testing the idealised alt. The components are combined together, as illustrated in Figure 16, to produce a process IdealisedAlt. We then allow arbitrary behaviours after the error events, much as for the idealised model of a channel.

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

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

The table to the right gives statistics about these checks. In each case, we considered an alt with two branches, one in-port branch and one out-port branch. The checks are faster than the corresponding checks for a channel, mainly because normalisation of the specification was faster than for a channel, because the idealised alt has fewer states than an idealised channel. We think the main reason for this is that the Lin(t) processes in the idealised alt have fewer states than the Lin(t) processes in the idealised channel (there is one such process for each channel-thread, so the total number of states increases exponentially). In

Model	Threads	States	Time
F	4	33.1M	21s
FD	4	5.6M	5.4s
F	5	601M	472s
FD	5	62.7M	46s
F	6	10.6B	153min
FD	6	662M	701s

addition, the implementation model for an alt has fewer states than for a channel.

We used a couple of techniques to improve the efficiency of these checks. Firstly, we applied the prioritisation operator of FDR to IdealisedAlt to give priority to error events over all other events (except τs); so in any state where an error event (or τ) is possible, all other events are blocked. This is sound here because the subsequent behaviours allowed by the specification (in CHAOS(Interface) or DIV) include the behaviours corresponding to the blocked events.

Further, for the check in the stable-failures model, we normalised IdealisedAlt before the application of the throws operator. This made the subsequent normalisation of $\mathsf{AltSpec}_F$ faster. (However, the same technique in the failures-divergences model made the check slower.)

6.3. Combining the idealised models

We now perform the final step in our compositional verification. Fix 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 (keeping the error events visible).

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 5.4 (with the spuriousWakeup events hidden). The table to the right gives statistics about these checks, where branches contains one in-port branch and one out-port branch.

Model	Threads	States	Time
F	4	12.2M	19s
D	4	5.53M	24s
F	5	909M	1820s
D	5	227M	1710s

We can then apply Proposition 7, as described earlier. Condition 1 of the proposition is shown to hold by the checks in Sections 6.1 and 6.2; condition 2 is shown to hold by the tests in the previous paragraph. Thus we can deduce that the corresponding combinations of the models of channels and alt implementations also refine the specifications described above.

7. Other components

In this section, we describe the modelling and analysis of some other components from the SCL library.

7.1. Barrier synchronisation

A barrier synchronisation object [And91] has signature as follows.

```
class Barrier(n: Int){
  def sync(me: Int): Unit
}
```

Each such object allows n threads to synchronise together. Each thread calls sync, passing in its identity, which is in the range $\{0, \ldots, \mathsf{n}-1\}$; each call blocks until all n threads have called sync.

The SCL implementation of a barrier synchronisation object arranges the threads into a binary heap, based on their identities. A Signal object, with signature as in Figure 20 (left), is used between each child node and its parent, to allow signals in both directions: each execution of signalUpAndWait first synchronises with an execution of waitForSignalUp, then synchronises with an execution of signalDown.

The implementation of sync, then, is as in Figure 20 (right). Each thread waits for a signal from its children (if it has children); signals to its parent (if it is not the root of the heap), and waits for a signal back; and then signals to each of its children. Thus each signal up the heap indicates that all the threads in that sub-heap have called sync, and signals down the heap indicates that all the threads have synchronised and so can return.

Our analysis starts by considering a single Signal object, interacting with its child and parent threads. The implementation is based on a JVM monitor, together with a boolean variable that indicates the direction of the last signal. This is easily modelled in CSP, using similar techniques to earlier. Each operation is framed by appropriate begin and end events (for later convenience, we include both child and parent identities, in that order, in the events for the parent thread). We then combine this with processes that represent the

```
private class Signal{
  /** Signal to the parent, and wait for
    * a signal back. */
  def signalUpAndWait = ...
  /** Wait for a signal from the child. */
  def waitForSignalUp = ...
  /** Signal to the child. */
  def signalDown = ...
}
```

```
def sync(me: Int) = {
  val leftC = 2*me+1; val rightC = 2*me+2
  if(leftC < n) signals(leftC).waitForSignalUp
  if(rightC < n) signals(rightC).waitForSignalUp
  if(me != 0) signals(me).signalUpAndWait
  if(leftC < n) signals(leftC).signalDown
  if(rightC < n) signals(rightC).signalDown
}
```

Fig. 20. Signature of a Signal object (left), and the sync operation.

```
If(b)(P) = if b then P else SKIP  -- macro for if statements.
Thread(me) =
  let leftC = 2*me+1
      rightC = 2*me+2 within
  beginSync.me →
  If(leftC < N)(beginWaitForSignalUp.leftC.me → endWaitForSignalUp.leftC.me → SKIP);
  If(rightC < N)(beginWaitForSignalUp.rightC.me → endWaitForSignalUp.rightC.me → SKIP);
  If(me ≠ 0)(beginSignalUpAndWait.me → endSignalUpAndWait.me → SKIP);
  If(leftC < N)(beginSignalDown.leftC.me → endSignalDown.leftC.me → SKIP);
  If(rightC < N)(beginSignalDown.rightC.me → endSignalDown.rightC.me → SKIP);
  endSync.me → Thread(me)
```

Fig. 21. CSP model of a thread that repeatedly calls sync.

child and parent threads, which repeatedly call the relevant operations (with the parent alternating between waitForSignalUp and signalDown, reflecting the intended usage).

We can then test for correctness against a specification that captures the expected synchronisations: we use event sync1.c.p for the synchronisation between signalUpAndWait (by child thread c) and waitForSignalUp (by parent thread p); and use sync2.c.p for the synchronisation between signalUpAndWait (by child thread c) and signalDown (by parent thread p). The following lineariser processes describe the expected orders.

```
LinChild(t, p) =
  beginSignalUpAndWait.t → sync1.t.p → sync2.t.p → endSignalUpAndWait.t → LinChild(t, p)
LinParent(t, c) =
  beginWaitForSignalUp.c.t → sync1.c.t → endWaitForSignalUp.c.t →
  beginSignalDown.c.t → sync2.c.t → endSignalDown.c.t → LinParent(t, c)
```

We can then build a specification process, SignalSpec, by combining the two linearisers in parallel, and hiding the sync1 and sync2 events. Testing the system against this specification in the traces and stable failures models verifies synchronisation linearisation and progressibility. We can also verify divergence freedom (when events representing spurious wakeups are kept visible). Each test completes in less than a second.

We now consider the barrier synchronisation object itself. We perform a compositional verification, and model each Signal object by the specification SignalSpec we used above. It is then straightforward to translate the sync operation into CSP, simulating the calls of operations on the signal objects: see Figure 21 (the macro If simulates an **if** statement with no **else** clause). Each call of sync is framed between beginSync and endSync events.

We then construct a system, combining n Thread processes and n−1 Signal processes; Figure 22 illustrates the system for n = 5. We can specify the barrier as a synchronisation object. A single lineariser is as follows, where event sync models the synchronisation.

```
Lin(id) = beginSync.id → sync → endSync.id → Lin(id)
```

We then combine together n such processes, synchronising on the sync events, and then hiding the sync events.

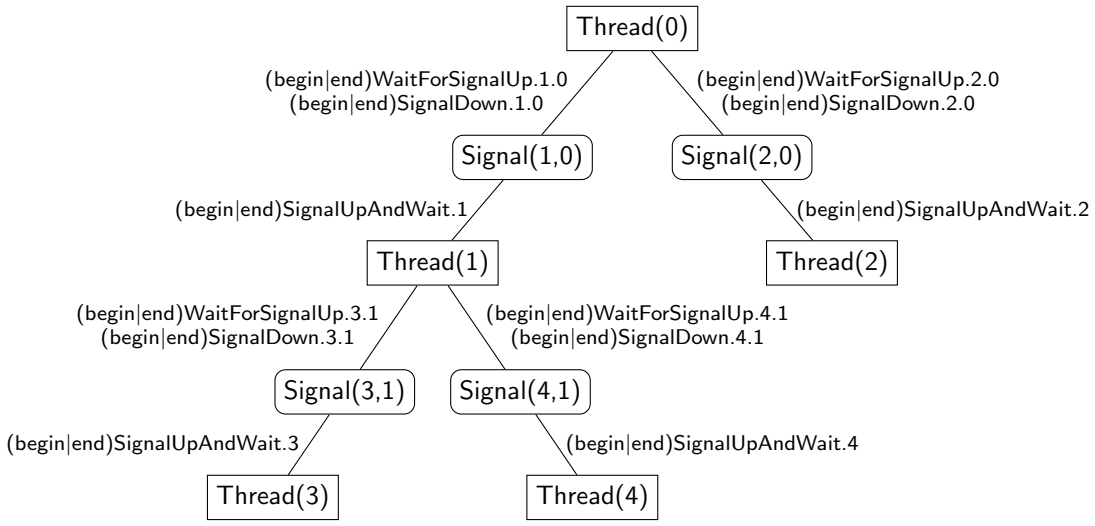


Fig. 22. Depiction of the CSP model of the barrier synchronisation object, with five threads.

Testing the system against this specification in the traces and stable failures models verifies synchronisation linearisation and progressibility (for a particular value of n).

7.2. SCL conditions

Earlier, we described Conditions, included in the SCL implementation of monitors. We now describe their implementation and analysis.

Each Condition is associated with a lock, which can be acquired and released; multiple conditions can be associated with a single lock. A thread that performs an await or a signal on the Condition must hold the lock.

The implementation of a Condition is based around the Java LockSupport class[6]. An object of this class allow a thread t to *park*, i.e. suspend. It also allows another thread to *unpark* t, i.e. wake t up. It uses a permit mechanism: if t is not parked, a permit is stored for it, so when it next parks, it resumes immediately.

However, LockSupport objects suffer from spurious wakeups: a parked thread may resume even though no other thread has performed an unpark. To counteract these spurious wakeups, before parking, a thread creates a ThreadInfo object as follows.

```
class ThreadInfo{
  val thread = Thread.currentThread //the waiting thread.
  var ready = false // has this thread received a signal?
}
```

The ready variable indicates whether the thread has received a signal. This ThreadInfo object is stored in a queue. A signalling thread sets the ready flag. A thread that resumes after parking checks the flag: if it is false, the thread must have had a spurious wakeup, so parks again.

Modelling the Condition in CSP is fairly straightforward. The lock is modelled by a simple two-state process. Each ThreadInfo object is modelled by a process with events allowing threads to read or write the ready flag, or to read the thread field. The queue is also modelled by a process, storing a sequence of identities of ThreadInfo processes, and with events to allow threads to test if the queue is empty, and to enqueue or dequeue ThreadInfos. The LockSupport mechanism is modelled (as in [Low19]) by a process parameterised by the set of identities of threads that are waiting, and the set of identities of threads for which there is a stored permit; events correspond to threads parking, unparking another thread, and spurious wakeups. Each

[6] https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/locks/LockSupport.html.

```
private var isUp = false

def up = synchronized{
  require(!isUp); isUp = true; notify()
}

def down = synchronized{
  while(!isUp) wait()
  isUp = false
}
```

Fig. 23. The implementation of a binary semaphore.

operation can then be easily modelled to interact with the other components; these operations are framed by begin and end events in the normal way.

We model a system with threads that may acquire the lock, and when holding the lock may (nondeterministically) perform a signal, an await, or release the lock. We specify this as a synchronisation object. In particular, we model a signal by thread t1 that awakens thread t2, as a synchronisation between the two threads, captured by event sync.t1.t2. However, a signal will be unsuccessful if there is no thread waiting, which we model by event nullSignal.t1. Thus we can define lineariser processes as follows.

```
Lin(t) = acquire.t → Lin'(t)
Lin'(t) =
  let others = ThreadID−{t} within
  beginSignal.t → (sync.t?other:others → endSignal.t → Lin'(t) □ nullSignal.t → endSignal.t → Lin'(t))
  ⊓ beginAwait.t → release.t → sync?other:others!t → acquire.t → endAwait.t → Lin'(t)
  ⊓ release.t → Lin(t)
```

The state Lin'(t) corresponds to thread t is holding the lock. The branch for signals reflects the two possibilities just described. The branch for awaits reflects the fact that the thread releases the lock before suspending, and reacquires it when awoken.

We need to give successful signals priority over unsuccessful ones: if a thread is waiting, a signal should wake up some such thread. We therefore run the linearisers in parallel with a process that keeps track of which threads are currently waiting, and allows a nullSignal only when none is waiting. For simplicity, we treat a thread as waiting from the point of its beginWait event.

```
MonitorSpec = MonitorSpec'({})
MonitorSpec'(ts) =        −− ts is the set of waiting threads.
  let others = ThreadID−ts within
  beginAwait?t:others  → MonitorSpec'(ts∪{t})
  □
  if empty(ts) then nullSignal?t1:others → MonitorSpec'(ts)
  else sync?t1:others?t2:ts → MonitorSpec'(ts−{t2})
```

We then combine the linearisers and MonitorSpec in parallel, and hide the sync and nullSignal events. We can then test that the system refines this specification in the traces and stable failures models to verify synchronisation and progressibility. We can also verify divergence freedom (when the events representing spurious wakeups are visible).

7.3. Binary semaphores

A binary semaphore is a very common concurrency primitive [And91]. The semaphore has two states, *up* and *down*. It has two operations up and down. The down operation waits until the semaphore is in the *up* state, and moves it to the *down* state. The up operation moves the semaphore from the *down* state to the *up* state. Some implementations treat an up from the *up* state as being a no-op; but this often leads to bugs, where signals are lost. Our implementation therefore requires that up is executed only from the *down* state, and throws an exception otherwise.

Our implementation is in Figure 23. It uses a boolean variable isUp to record if the semaphore is in the *up* state, and uses a JVM monitor.

Modelling a semaphore is mostly a straightforward adaptation of the techniques described earlier. However, the require statement is treated differently from earlier assert statements: if the condition is not satisfied, this represents a misuse of the semaphore, rather than an error in the implementation of the semaphore; we should expect our refinement checks to allow such misuses, as long as they are treated appropriately. Hence we arrange that if a thread t process detects that the condition does not hold, it releases the lock on the monitor and performs an event error.t (instead of the standard end event) to signal the misuse.

The correctness condition for a binary semaphore is standard linearisation: no synchronisation is involved. To specify this linearisation, we define a lineariser for each thread as follows, where events up.t and down.t represent correct linearisations of the two operations, and errorUp.t represents the linearisation of an execution of up outside its precondition.

```
Lin(t) =
  beginUp.t → (up.t → endUp.t → Lin(t) □ errorUp.t → error.t → Lin(t))
  □ beginDown.t → down.t → endDown.t → Lin(t)
```

Note that this allows the up operation to be linearised either correctly or incorrectly. We put these linearisers in parallel with the process Down, below, which allows only the correct linearisation events; we then hide the up, down and errorUp events.

```
Down = up?t → Up
Up = down?t → Down □ errorUp?t → Up
```

We can then use FDR to verify that the implementation model refines this specification in both the traces and stable failures model, and also that the model is divergence-free (when spurious wakeups are visible). With five threads, each check completes within five seconds.

8. Conclusions

In this paper we have analysed a library of communication primitives, using CSP and its model checker FDR. We have shown how the properties of synchronisation linearisability and synchronisation progressibility can be captured as CSP refinement checks, and tested using FDR. The specification is built from a lineariser process for each thread: these processes synchronise on events that represent the synchronisations of operations.

We believe that this style is widely applicable to other synchronisation objects. We sketch the general construction of the specification process.

Consider a synchronisation object that allows binary synchronisations between operations op1 and op2. We can use events of the form sync.t1.t2.x1.x2.y1.y2 to represent a synchronisation between an execution by t1 of op1(x1) returning y1, and an execution by t2 of op2(x2) returning y2. (In many cases, the form of these events can be simplified; the specifications in the body of this paper replaced the last four fields by just the data value being passed.) Then we can define linearisers processes as follows.

```
Lin(t) =
  beginOp1.t?x1 → sync.t?other!x1?x2?y1?y2 → endOp1.t.y1 → Lin(t)
  □ beginOp2.t?x2 → sync?other!t?x1!x2?y1?y2 → endOp2.t.y2 → Lin(t)
```

In each case, this thread contributes its identity and parameter to the sync event, and receives back its return value. The lineariser ensures that each synchronisation event happens between the corresponding begin and end events. These linearisers can then be placed in parallel with a suitable process that specifies what values should be returned, possibly as a function of some state. A generic definition for this process would be

```
SyncSpec(state) = sync?t1?t2?x1?x2!f1(state,x1,x2)!f2(state,x1,x2) → SyncSpec(update(state,x1,x2))
```

where state captures the current state, f1 and f2 define the return values as functions of the current state and the parameters, and update defines how the state is updated. These functions can be chosen so as to capture the informal specification of the synchronisation object.

This construction can be generalised to allow an operation to synchronise in multiple different ways, as in Section 5.4. It can also be extended to allow synchronisations between more than two threads.

The analysis revealed an error in a previous version of the code. But it also helped to reach a correct

implementation:I went through multiple iterations of finding bugs using FDR, and coming up with possible fixes. I don't think I would have reached correct code without the benefits of model checking.

We have shown how compositional verification can be used to make an analysis more efficient, and so allow larger systems to be analysed. The compositional verification proved more difficult that we had expected. One reason for this is that the concrete components have complex behaviours which the idealised versions have to reflect. There is a decoupling between the external interface —in terms of begin and end events— and where those operation executions have an effect: the use of lineariser components within the idealised models proved useful in capturing this decoupling. The other main source of difficulty was the necessity of capturing environmental assumptions upon the components: we used error events to signal a violation of an environmental assumption, which led to subsequent arbitrary behaviour; we believe this technique works well.

We should be clear about what we have shown. We have analysed particular models, with particular numbers of channel-threads, particular choices for the branches of an alt, and a particular choice of the type Data of data values. In the absence of further arguments, these results to not imply that corresponding results hold for larger values of those parameters would also hold — but they do help to give us confidence that they would, since experience shows that most bugs are exhibited by rather small instances.

The *parameterised model checking problem* seeks to verify a family of systems, for all values of certain parameters, such as those identified in the previous paragraph. The problem is undecidable in general [AK86, ML14]. Nevertheless, several approaches have been proposed that work in a number of situations.

The easier parameter to deal with is the type Data of data values. The models in this paper are data independent in this type: values are input, nondeterministically chosen, stored, and output, but no operations are applied to values that constrain the type. Data independence has been much studied previously, e.g. [Wol86, Laz99]: typically these results show that if a correctness property holds for a particular size of a type, it also holds for all larger types. However, these results are not useful for the correctness properties in this paper: the results of [Ros98, Section 15.2], when applied to the systems of Section 4.2 with five threads, would require us to use *eleven* data values, which is almost certainly infeasible. In Appendix B we take a more direct approach, and prove that, in fact, taking Data to include just *two* data values is enough to deduce that the results also hold for larger types.

There has been much work considering the parameterised model checking problem where the parameter is the number of processes in the system, e.g. [Lub84, CG87, WL89, GS92, EN95, PXZ02, ML14, AHH16, Low22]. A particular difficulty in applying these techniques to the work in this paper is that the model of a monitor is parameterised by the set of identities of threads that are currently waiting, which is potentially unbounded: to our knowledge, none of the existing techniques can be applied in such a setting. We leave consideration of this question as future work.

Acknowledgements

This work has benefited from discussions with Jonathan Lawrence, Zeyang Zhao, and Ilker Cicek. I would like to thank the anonymous reviewers for their useful comments.

References

[AHH16] Parosh Abdulla, Frédéric Haziza, and Lukáš Holík. Parameterized verification through view abstraction. *International Journal on Software Tools for Technology Transfer*, 18:495–516, 2016.

[AK86] K. R. Apt and D. C. Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 22(6):307–309, 1986.

[And91] Gregory Andrews. *Concurrent Programming: Principles and Practice*. Benjamin/Cummings, 1991.

[CG87] E. M. Clarke and O. Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the 6th Annual Association for Computing Machinery Symposium on Principles of Distributed Computing*, pages 294–303, 1987.

[DLP21] Brijesh Dongol and Jay Le-Papin. Checking opacity and durable opacity with FDR. In *Proceedings of SEFM 2021*, volume 13085 of *LNCS*, pages 222–242. Springer, 2021.

[EN95] E. Allen Emerson and Kedar S. Namjoshi. Reasoning about rings. In *Proceedings of the Symposium on Principles of Programming Languages (POPL '95)*, 1995.

[GS92] Steven M. German and A. Prasad Sistla. Reasoning about systems with many processes. *Journal of the ACM*, 39(3):675–735, 1992.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2012.
[HW90] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
[Jon83] Cliff Jones. Tentative steps towards a development method for interfering programs. *ACM Transations on Programming Languages and Systems*, 5(4):596–619, 1983.
[Law05] Jonathan Lawrence. Practical applications of CSP and FDR to software design. In *Communicating Sequential Processes: The First 25 Years*, volume 3525 of *Lecture Notes in Computer Science*, pages 151–174. Springer, 2005.
[Laz99] Ranko Lazić. *A Semanti Study of Data Independence with Applications to Model Checking*. DPhil thesis, University of Oxford, 1999.
[LL25] Jonathan Lawrence and Gavin Lowe. Synchronisation: Specification and testing. Submitted for publication, 2025.
[Low96] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proceedings of TACAS*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996. Also in *Software—Concepts and Tools*, 17:93–102, 1996.
[Low11] Gavin Lowe. Implementing generalised alt — a case study in validated design using CSP. In *Communicating Process Architectures*, pages 1–34, 2011.
[Low17] Gavin Lowe. Analysing lock-free linearizable datatypes using CSP. In Thomas Gibson-Robinson, Philippa Hopcroft, and Ranko Lazić, editors, *Concurrency, Security, and Puzzles*, volume 10160 of *LNCS*, pages 162–184. Springer, 2017.
[Low19] Gavin Lowe. Discovering and correcting a deadlock in a channel implementation. *Formal Aspects of Computing*, 31:411–419, 2019.
[Low22] Gavin Lowe. Parameterized verification of systems with component identities, using view abstraction. *Software Tools for Technology Transfer*, 24(2), 2022.
[Low24] Gavin Lowe. On data independence. Technical report, University of Oxford, 2024. `https://www.cs.ox.ac.uk/people/gavin.lowe/Papers/dataindependence.pdf`.
[Lub84] B. Lubachevsky. An approach to automating the verification of compact parallel coordination programs. *Acta Informatica*, 21(2):125–169, 1984.
[MC81] Jayadev Misra and K. Mani Chandy. Proofs of networks of processes. *Transations on Software Engineering*, 7(4):417–426, 1981.
[ML14] Tomasz Mazur and Gavin Lowe. CSP-based counter abstraction for systems with node identifiers. *Science of Computer Programming*, 81:3–52, 2014.
[MS01] A. Mota and A. Sampaio. Model-checking CSP-Z: Strategy, tool support and industrial application. *Science of Computer Programming*, 40(1):59–96, 2001.
[Pay23] Alex Pay. Automated modelling of an imperative language using CSP. Master of Computer Science, final-year project, University of Oxford, 2023.
[PC23] Jan B. Pedersen and Kevin Chalmers. Towards verifying cooperatively-scheduled runtimes using CSP. *Formal Aspects of Computing*, 35(4):1–45, 2023.
[PJ91] Paritosh Pandya and Mathai Joseph. P-A logic — a compositional proof system for distributed programs. *Distributed Computing*, 5:37–54, 1991.
[PXZ02] A. Pnueli, J. Xu, and L. D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *CAV'02: Proceedings of the 14th International Conference on Computer Aided Verification*, pages 107–122, 2002.
[RH07] A. W. Roscoe and David Hopkins. SVA, a tool for analysing shared-variable programs. In *Proceedings of AVoCS 2007*, pages 177–183, 2007.
[RLW+24] Azalea Raad, Ori Lahav, John Wickerson, Piotr Balcer, and Brijesh Dongol. Intel PMDK transations: Specification, validation and concurrency. In *Programming Languages and Systems (ESOP 2024)*, volume 14577 of *LNCS*, pages 150–179, 2024.
[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.
[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
[SA05] Bernhard H.C. Sputh and Alastair R. Allen. JCSP-poison: Safe termination of CSP process networks. In *Communicating Process Architectures*, 2005.
[Suf08] Bernard Sufrin. Communicating Scala Objects. In *Proceedings of Concurrent Process Architectures*, 2008.
[WA14] P.H. Welch and P.D. Austin. JCSP home page. `http://www.cs.ukc.ac.uk/projects/ofa/jcsp/`, 2014.
[WL89] P. Wolper and V. Lovinfosse. Verifying properties of large sets of processes with network invariants. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 68–80, 1989.
[WM00a] Peter Welch and Jeremy Martin. A CSP model for Java multithreading. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 114–122. IEEE, 2000.
[WM00b] Peter Welch and Jeremy Martin. Formal analysis of concurrent Java systems. In *Proceedings of Communicating Process Architecture*, 2000.
[Wol86] Pierre Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.
[Zha22] Zeyang Zhao. Model checking concurrent objects. Master's thesis, Oxford University, 2022.

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.

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.

For example, the implementation of an SCL 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 following macros, to increment or decrement the receiversWaiting variable, illustrate the above 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[7] 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.

For example, the ChannelThread process from Figure 3 can be written more conveniently as follows, to allow for different results.

```
ChannelThread(me) =
  beginSend.me?x → Send(me, x, λres • endSend.me.res → ChannelThread(me))
  □ ...
```

Then the Send process takes a continuation parameter, which it applies to the final result.

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 :: Bool → Proc 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** {...} (where result is a parameter of the current process, and status is a shared variable) 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.

[7] Proc is the type of CSP processes.

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

For example, the loop **while**(status != Empty) slotEmptied.await() from Figure 1 could be written more more concisely as

```
Send1(me, x) =
  let Test(k) = getStatus.me?s → k(s ≠ Empty)
  within While(Test, Monitor::Await(me, SlotEmptied))
```

B. On the number of data values

In this section we show that for each of the checks we have considered in this paper (with a particular choice of threads), if the check succeeds when we use a type Data of data values of size 2, then the check would also succeed for a larger choice of Data. We consider Data a type parameter that can be instantiated in different ways.

B.1. Data independence

Our approach makes use of techniques from data independence [Wol86][Laz99][Ros98, Section 15.2]. Consider a family $P[t]$ of processes, parameterised by a type variable t that can be instantiated with different non-empty types. We say that the processes are *data independent* in t if they can input, nondeterministically choose, store, and output values from t, but can't perform any other operations on such values, including equality tests, and don't use any constants from t.

The processes we have used in this paper are nearly data independent in Data, but with one exception. The implementation of sendWithin(d)(x), in the case of a time out, checks, via an assertion, that the value variable still holds the value x that the operation wrote there previously. This assertion plays no role in the functionality of the channel: it is merely a sanity check. But this assertion contains an equality test, so means the process isn't data independent. We could adapt the models, to make them data independent, by simply removing this assertion from the model (and, indeed, from the implementation). For the moment, we consider these adapted models. In Section B.3, we argue that, in fact, this change isn't necessary.

Previous versions of the models were not data independent for another reason. The implementations of channels include a variable value that stores the current (or previous) value being sent; this variable is initialised arbitrarily (to null). This variable is modelled by a CSP process; previously that process was initialised to a constant value A from Data. Similarly, the model of an alt contains a process that models a variable that holds the current (or previous) value received; this was also initialised to a constant value. This usage of constants from Data took the models outside the realm of the data independence assumptions. We therefore adapted the models so that the initial values are chosen nondeterministically, so as to make them data independent. (This change made negligible difference to the checks in Section 3, but increased the state space of the checks in Section 5 by 2–3%.)

We will make use of the following results. Consider a data independent family of processes $P[t]$. Let T and T' be two concrete types, and let $f : T \to T'$ be a surjective function. Lift f to events by pointwise application; and then lift to traces by pointwise application. The following results link the semantics of $P[T]$ and $P[T']$ [Low24].

- If $tr \in traces(P[T])$ then $f(tr) \in traces(P[T'])$;
- If $(tr, X) \in failures(P[T])$ then[8] $(f(tr), \{e \mid f^{-1}(\!|e|\!) \subseteq X\}) \in failures(P[T'])$;
- If $tr \in divs(P[T])$ then $f(tr) \in divs(P[T'])$.

Informally, whenever $P[T]$ has a transition (in the operational semantics) labelled with event e, $P[T']$ has a corresponding transition labelled with $f(e)$, and vice versa; the above results lift this to the denotational models.

[8] The notation $f^{-1}(\!|e|\!)$ represents the relational inverse image of e under f, i.e. $\{x \mid f(x) = e\}$.

B.2. Synchronisation progressibility for basic channels

Let $System[t]$ be one of the families of systems for channels that we considered in Section 3, where the type parameter t represents the type of data. Let T be an arbitrary type, with $\#T \geq 2$, and let $T_2 = \{A, B\}$. We show that if $System[T]$ fails synchronisation progressibility, then so does $System[T_2]$; thus it is enough to verify $System[T_2]$.

We start by considering basic channels, with just send and receive operations.

Consider a maximal failure (tr, X) (i.e. where X contains *all* events that would be refused in the relevant state) of a correct implementation. Suppose there is a pending send execution (i.e., that has been called but not returned); if that execution has synchronised, then the corresponding endSend event is not refused (i.e. is not in X); and if the execution has not synchronised, then the corresponding endSend event is refused. Likewise, suppose there is a pending receive execution; if that execution has synchronised with an execution of send(v), then the corresponding endReceive event with data value v is not refused (but all the other corresponding endReceive events of that thread are refused); and if the execution has not synchronised, then all corresponding endReceive events of that thread are refused. Thus by examining this maximal failure, we can calculate which executions have synchronised. This prompts the following definition.

Definition 8. Let (tr, X) be a maximal failure of the system. We say that an operation execution has *synchronised* if either there is a corresponding end event with a Success value (i.e. the execution returned successfully) or such an end event is not refused.

We define the *data value* of an operation execution as follows. The data value of an execution send(v) is v. The data value of a synchronised execution of receive is either the data value in the endReceive event, or (for a pending execution) the data value in the available endReceive event.

Definition 9. We define the restriction of trace tr to data value v, written $tr|v$, to be the subtrace of tr containing: (1) events of operation executions whose data value is v, and (2) the beginReceive events of unsynchronised executions of receive. We define the restriction of (tr, X) to v, written $(tr, X)|v$, to be $(tr|v, X)$.

The following lemma shows that, in deciding synchronisation progressibility, we can consider each value in turn. The contrapositive shows that if a behaviour is not synchronisation progressible, there is a particular value that's responsible.

Lemma 10. Let (tr, X) be a maximal failure. Suppose that for every data value v, the restriction $(tr, X)|v$ is synchronisation progressible; then the whole failure is synchronisation progressible.

Proof: Consider a particular value of v. The fact that $(tr, X)|v$ is synchronisation progressible implies that there is some way of pairing up the synchronised operation executions for v, corresponding to the synchronisations, and such that no further such synchronisations are possible:

- If there is an unsynchronised execution of send(v), then there is no unsynchronised execution of receive;
- If there is an unsynchronised execution of receive, there is no unsynchronised execution of send(v).

Considering all values of v together, we deduce that there is some way of pairing up all the synchronised executions, corresponding to the synchronisations, and such that no further such synchronisations are possible:

- If there is an unsynchronised execution of send, then there is no unsynchronised execution of receive;
- If there is an unsynchronised execution of receive, there is no unsynchronised execution of send.

This implies that the whole failure is synchronisation progressible. □

Note 11. The above lemma does not hold when we replace synchronisation progressibility (a property of stable failures) by synchronisation linearisability (a property of traces). Consider the trace:

$tr = \langle$beginSend.t1.A, beginSend.t2.B, beginReceive.t3, endSend.t1, endSend.t2$\rangle$.

This represents a complete execution of send(A), a complete execution of send(B), and a pending execution of receive, all overlapping. For each value v, the restriction $tr|v$ is synchronisation linearisable: the send could synchronise with the pending receive. However, tr itself is clearly not synchronisation linearisable, because only one send can synchronise with the receive.

Proposition 12. Consider some type T of data values, with $\#T \geq 2$. Suppose $System[T]$ is not synchronisation progressible. Then $System[T_2]$ is not synchronisation progressible.

Proof: By the assumption, there is some maximal failure (tr, X) of $System[T]$ that is not synchronisation progressible. Then, by Lemma 10, there is a value v such that $(tr, X)|v$ is not synchronisation progressible. Consider the function $f : T \to T_2$ such that $f(v) = A$, and $f(x) = B$ for all $x \neq v$. Then

$$(f(tr), \{e \in \Sigma \mid f^{-1}(e) \subseteq X\}) \in failures(System[T_2]),$$

by the earlier data-independence result. But that failure is not synchronisation progressible for the same reason that $(tr, X)|v$ is not synchronisation progressible. □

B.3. Extensions

We now extend our analysis to consider the closing of channels, the timed operations and alts. We then consider divergence freedom, and remove the restriction mentioned earlier.

Closing and timed operations. We extend Definition 8 to sendWithin and receiveWithin in the obvious way. Given a maximal failure, we say that an execution of send, receive, sendWithin or receiveWithin *detects closure* if either it returns the Closed value, or is pending and the return of the Closed value is not refused (so all successful returns are refused). We say that a sendWithin or receiveWithin execution *times out* if either it returns Timeout, or such an event is not refused.

We extend the definition of the restriction of trace tr to data value v, written $tr|v$, to include: (1) events of operation executions whose data value is v; (2) the begin events of unsynchronised executions of receive and receiveWithin; (3) begin and end events of all executions that detect closure; (4) begin and end events of all executions that time out; and (5) all beginClose and endClose events.

The proof then proceeds much as before. In the proof of the adaptation of Lemma 10, the fact that $(tr, X)|v$ is synchronisation progressible implies there is some way of choosing the first linearisation point of an execution of close (if there is one) such that:

- all operation executions that synchronise with data value v can be linearised before the linearisation of close;
- all executions (for any data value) that time out can be linearised before the linearisation of close;
- all executions (for any data value) that detect closure can be linearised after the linearisation of close.

Consider all values of v together. Let $\hat{v}$ be the value of v that has the latest linearisation point for close; we linearise the close at this point.

- This is necessarily after all linearisations of synchronisations and executions that time out, by the first two bullet points above;
- The begin and end events of all executions that detect closure are included in $(tr, X)|\hat{v}$; hence they are after the linearisation point of close, by the third bullet point above.

Alts. We now extend the analysis to alts. Consider a maximal refusal. We say that an execution of the alt *synchronises to receive* x if either an endAlt.t.AltReceive.i.x event occurs, or such an event is not refused. We say that an execution *synchronises to send* x similarly. We say that an execution *aborts* if either an endAlt.t.AltAbort event occurs, or such an event is not refused.

We then extend the restriction of trace tr to data value v in the obvious way, to also include: (1) beginAlt and endAlt events of executions that send or receive v; and (2) beginAlt events of executions that fail to synchronise (so either abort or are blocked).

The proof then proceeds much as before.

Divergence freedom. Suppose for some concrete type T, with $\#T \geq 2$, that $System[T]$ can diverge after trace tr. Let $f : T \to T_2$ be an arbitrary surjection. Then by the earlier data-independence result, $f(tr)$ would be a divergence for $System[T_2]$. Thus it is enough to verify that $System[T_2]$ is divergence-free. (In fact, we could consider a single data value here; but we need two data values for the argument in the following paragraphs.)

Recall that the implementation of sendWithin uses an equality test, corresponding to an assertion, and so the model is not data independent: if the equality test returns false, the process diverges. We now argue that when we include this assertion in the model, verifying there is no such divergence when using $T_2 = \{A, B\}$ implies that there is no such divergence for larger types of data.

Suppose otherwise, and there is a divergence in the system using T, say when the process compares values x and y, following trace tr. Consider the function f that maps x to A, and maps all other values to B. Then the system using T_2 would compare values A and B following trace $f(tr)$, and so would also have a divergence. This is a contradiction.

Thus, the behaviours of the models with and without the assertion have identical behaviours. In particular, the former satisfies synchronisation progressibility, because the latter does.