Analysing a Library of Concurrency Primitives using CSP

Journal:	*Formal Aspects of Computing*
Manuscript ID	FAC-2024-0020
Manuscript Type:	Original Research Paper
Date Submitted by the Author:	18-Jun-2024
Complete List of Authors:	Lowe, Gavin; Oxford University,
Specialty/Area of Expertise:	Concurrency, Model checking, CSP

Under consideration for publication in Formal Aspects of Computing

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. 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 invocation corresponds to a correct synchronisation) and *progressibility* (informally, that invocations don't get stuck unnecessarily): 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 and semaphores. In this paper we analyse the message-passing primitives: we build CSP models of them, and then use the model checker FDR to test for correctness. To our surprise, the analysis revealed a bug in the implementation.

We start by describing relevant aspects of SCL. Program threads can send and receive messages using *channels*. If c is a channel, then the command c.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 much wait until there is a thread willing to receive, so that the two invocations synchronise. Channels are typed: the type SyncChan[A] represents synchronous channels that send data of type A. A channel is composed of an *outport*, where values are sent, and an *inport*, where values are received.

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

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

[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

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

An 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 *inport branch* is denoted "in =?=> f" where in is a channel (or inport), and f is a function whose argument matches the type of in; we call f a *continuation* (above, "x => println(x)" denotes the function that takes argument x and executes println(x)). An *outport branch* is denoted "out =!=> e ==> cont", where out is a channel (or outport), e is an expression whose value matches the type of out, and c is a computation, which we again call a continuation (this continuation is optional).

The alt waits until there is another thread ready to communicate at the other end of the channel corresponding to one of the branches, at which point the two threads can synchronise to transmit a value. In the case of an inport branch, the continuation is applied to the value received. In the case of an outport branch, the value of the expression is sent, and the continuation (if present) is executed.

A branch may have a boolean *guard*: a branch 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 outport branch might have side effects; therefore the relevant expression 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 inport branch is feasible only if the buffer is not full and in is not closed; the outport branch is feasible only if the buffer is not empty and out is not closed. If no branch of an alt is feasible, it throws an AltAbort exception.

The construct serve(branches) is like an alt(branches) that is executed repeatedly, except any AltAbort exception is caught. Hence it runs repeatedly until no branch is feasible.

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 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 CSP model of the implementation, but 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* [LL24],

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

informally, that each completed operation invocation corresponds to a correct synchronisation. We also describe a related progress property, *synchronisation progressibility*, informally, that invocations don't get stuck unnecessarily. We present the corresponding CSP specification and refinement check for each property. We then 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; the correct implementation was found with the help of the model checking described in this paper. We then extend the analysis to the timed send and receive operations (but ignoring the interactions with alts at this point).

In Section 4 we consider alts. We describe the high-level design 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 useful in helping to develop a correct implementation: it revealed various flaws with earlier versions. However, the analysis suffers from a state-space explosion, and so it's possible to analyse only rather small systems.

In Section 5, we perform an alternative, compositional, verification. We build a more abstract CSP description of a synchronous channel, describing the way it reacts to operation calls and interacts with alts, but abstracting away from details of the implementation: we call this an *idealised channel*. We show that the CSP model of the channel implementation refines this idealised channel. Likewise, we build an idealised model of an alt, and show that it is refined by the model of the implementation. 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; we describe how we capture these assumptions. 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.

We sum up in Section 6.

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 on code;
- The demonstration of compositional verification, in particular where each component makes assumptions about the correct behaviour of other components.

1.1. Related work

CSP has been used to analyse message passing concurrency primitives on a number of previous occasions. Welch and Martin [WM00] present a model of Java multi-threading (in particular, monitors), including a model of a channel within their own concurrency API. I [Low11] use CSP to derive and verify a generalisation of the alt construct. 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.

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. 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 c.3 represents the value 3 being passed on channel c. Channels may be declared using the keyword **channel**; for example, **channel** c : 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 {|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 ✓.

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

The process P □ 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, P ⊓ Q may act like either P or Q, with the choice being made nondeterministically, not under the control of the environment. □ x:X • P(x) is an indexed external choice, with the choice being made over the processes P(x) for x in X. Likewise, ⊓ x:X • P(x) represents a nondeterministic choice over the P(x).

The process **if** b **then** P **else** Q represents a conditional. The process b & 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 P; Q represents a sequential composition of P and Q: initially, P is run, but when it terminates (as indicated by event ✓), Q is run.

The process CHAOS(A) can perform any events from the set A, or can refuse any of those events; however, it cannot diverge. Thus it allows arbtrary non-divergent behaviour 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: the events from E can be thought of as exceptions, and Q as 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 in 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. The process P⟦a \ b⟧ represents a renaming of P: each occurrence of event a is replaced by b; this extends to the renaming of multiple 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.

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

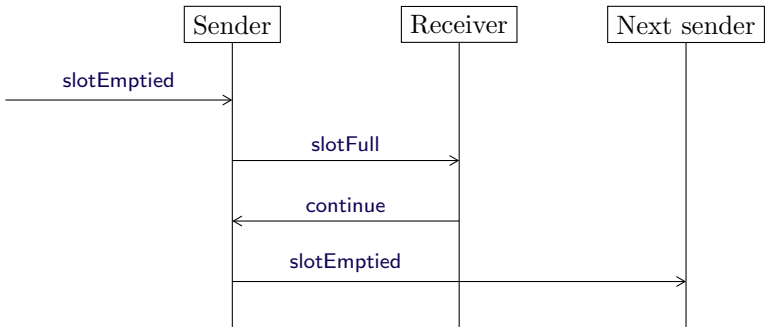


Fig. 1. Sequence diagram illustrating signalling on conditions within a channel implementation.

(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, without considering interaction with alts.

We start by considering just the send and receive operations. We outline the implementation for these operations, how we model them in CSP, and then how we analyse their correctness. In later subsections, we extend the analysis to include the timed operations and the closing of channels; for the moment we elide details related to those operations or to the use of alts. Recall that channels are shared: multiple senders and receivers can compete to use the channel.

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

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

Empty: No sender has deposited a value;

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

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

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

Figure 1 illustrates the use of conditions within the channel implementation. A sender waits (on a condition slotEmptied) until the status is Empty. It then stores its value in value, sets the status to Filled, and signals (on a condition slotFull) to a receiver. It next waits (on a condition continue) until the status is Read. Finally, it sets the status back to Empty, and signals (on slotEmptied) to the next sender.

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

We now outline the CSP model. The model uses small fixed types representing the type of data communicated by the channel, and the type of thread identities. (We discuss the choices for these types in Section 6.)

The monitor is modelled by a CSP module, encapsulating a process that maintains the state of the monitor, specifically: (1) which thread, if any, holds the lock on the monitor; and (2) which threads are waiting on which conditions. When no thread holds the lock, the monitor process allows a thread, other than any of the waiting threads, to acquire the lock. The thread that currently holds the lock can:

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

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

The model of a channel includes a monitor, and various other processes. Each variable in the implementation of the channel is modelled by a CSP process, with CSP channels to allow the variable's value to be read or written (see Appendix A).

Each of the send and receive operations can then be straightforwardly translated into a CSP process: this process performs the operations on the monitor and variables, following the Scala code. (Appendix A describes various techniques that we found useful in this modelling.) The Scala code uses several assertions: we model these by having the CSP process diverge if the property does not hold; later, we use FDR to verify that such divergences do not occur. Each operation is framed by begin and end events:

- The event beginSend.t.x represents a thread with identity t calling 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 invocation of the receive operation that successfully receives the value x.

We encapsulate the model of a channel inside a CSP module, so that only the begin and end events are visible outside the module. We achieve this encapsulation by including inside the module a process corresponding to each thread, which accepts the begin events, simulates the operation, and then performs the appropriate end event; this then allows us to hide the internal events. A process outside the module can simulate calling the operation by synchronising on the begin and end events; below, we capture correctness properties in terms of those events.

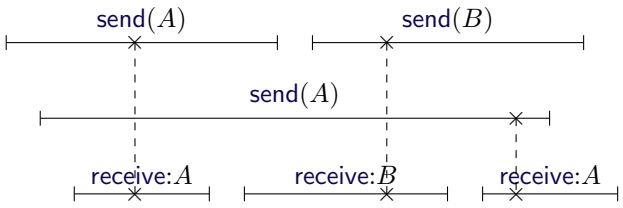
3.1. Analysing the basic channel

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

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

In [LL24], we introduced a correctness condition, *synchronisation linearisation*, corresponding to synchronisation objects like this. The idea is that each synchronisation should appear to take place between the beginning and end of the two corresponding invocations; different synchronisations should occur in a one-at-a-time order. We call the points at which the synchronisations appear to take place *synchronisation points*. (The definition is based on *linearisation* [HW90], the standard correctness property for concurrent datatypes, where invocations of operations should appear to take place in a one-at-a-time order, with each between the beginning and end of that invocation.)

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



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

Thus a history (or trace) is synchronisation linearisable if it is possible to identify synchronisation points with the desired properties. This corresponds to identifying which invocations synchronise with one another. A synchronisation object is synchronisation linearisable if all of its histories are synchronisation linearisable [LL24].

We now describe how we use model checking to test synchronisation linearisability of the channel. We create an instance C of the channel module. Below, expressions of the form C::v represent a value v from this instance. We create a process System that combines the channel and a number of threads (with identities from a set ThreadID): each thread repeatedly calls the send and receive operations (represented by events on CSP channels C::beginSend and C::beginReceive), and waits for them to return (on CSP channels C::endSend and C::endReceive).

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

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

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

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

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

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

$$\mathsf{Spec} = \mathsf{Spec_0} \setminus \mathsf{\{|sync|\}}$$
$$\mathsf{assert\ Spec} \sqsubseteq_T \mathsf{System}$$

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

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

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

progressiblity [LL24]. This property corresponds to an assumption that the scheduler is fair in the sense that if a thread is continuously runnable, it will eventually be scheduled and so able to make progress; however, this doesn't prevent the thread being repeatedly preempted, for example when competing with other threads to acquire a lock.

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

assert Spec $\sqsubseteq_{FD}$ System

It turns out that this specification is equivalent to that of Welch and Martin [WM00] 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.

3.2. Closing channels

We now consider the closing of channels. Recall that, after the channel has been closed, a send or receive invocation 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 both A and B should think the communication has succeeded, or both should throw Closed exceptions.

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

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

If a receiving thread waits on slotFull, when it receives a signal, it first checks whether status holds Filled. If so, it continues as 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 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 implementation asserts this, and the analysis below checks this). However, if there is a receiver waiting (as recorded by receiversWaiting), then that receiver will eventually return the value being sent, so the sending thread should also return successfully. If there is no waiting receiver, the sending thread throws a Closed exception.

The precise order of checks in the previous paragraph 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, as a receiving thread might have read the sending thread's value, and returned that value, indicating that it had correctly synchronised. The correct version of the code 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 3.1 is (in the terminology of [LL24]) *stateless*: no state is carried forward from one synchronisation to another. However, when we consider closing of the channel, it becomes *stateful*, with two states, open or closed. (Other synchronisation objects have more interesting states.)

Our specification will treat close as a linearisable operation: it will appear to take place atomically, at some point, called the *linearisation point*, between the beginClose and endClose events. (Equivalently, the close operation can be thought of as a unary synchronisation, involving a single thread, in contrast to the

earlier binary synchronisations.) We require that the history is consistent with this closing: synchronisations between sends and receives should take place before the linearisation point of the close; and sends and receives that return Closed should be linearised after the close.

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

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

$$\mathsf{ChanSpec} = \mathsf{sync}?\mathsf{t1}?\mathsf{t2}?\mathsf{x} \rightarrow \mathsf{ChanSpec} \;\Box\; \mathsf{close}?\mathsf{t} \rightarrow \mathsf{ChanSpecClosed}$$
$$\mathsf{ChanSpecClosed} = \mathsf{isClosed}?\mathsf{t} \rightarrow \mathsf{ChanSpecClosed} \;\Box\; \mathsf{close}?\mathsf{t} \rightarrow \mathsf{ChanSpecClosed}$$
$$\mathsf{alphaChanSpec} = \{|\mathsf{sync}, \mathsf{close}, \mathsf{isClosed}|\}$$

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

$$\begin{aligned}&\mathsf{Lin}(\mathsf{t}) = \\ &\quad \mathsf{C::beginSend}.\mathsf{t}?\mathsf{x} \rightarrow \mathsf{SendingLin}(\mathsf{t}, \mathsf{x}, \mathsf{C::endSend}.\mathsf{t}) \\ &\quad \Box\; \mathsf{C::beginReceive}.\mathsf{t} \rightarrow \mathsf{ReceivingLin}(\mathsf{t}, \mathsf{C::endReceive}.\mathsf{t}) \\ &\quad \Box\; \mathsf{C::beginClose}.\mathsf{t} \rightarrow \mathsf{close}.\mathsf{t} \rightarrow \mathsf{C::endClose}.\mathsf{t} \rightarrow \mathsf{Lin}(\mathsf{t})\end{aligned}$$

$$\begin{aligned}&\mathsf{SendingLin}(\mathsf{t}, \mathsf{x}, \mathsf{endChan}) = \\ &\quad \mathsf{sync}.\mathsf{t}?\mathsf{other}!\mathsf{x} \rightarrow \mathsf{endChan}.\mathsf{SendSuccess} \rightarrow \mathsf{Lin}(\mathsf{t}) \\ &\quad \Box\; \mathsf{isClosed}.\mathsf{t} \rightarrow \mathsf{endChan}.\mathsf{Closed} \rightarrow \mathsf{Lin}(\mathsf{t})\end{aligned}$$

$$\begin{aligned}&\mathsf{ReceivingLin}(\mathsf{t}, \mathsf{endChan}) = \\ &\quad \mathsf{sync}?\mathsf{other}!\mathsf{t}?\mathsf{x} \rightarrow \mathsf{endChan}.\mathsf{ReceiveSuccess}.\mathsf{x} \rightarrow \mathsf{Lin}(\mathsf{t}) \\ &\quad \Box\; \mathsf{isClosed}.\mathsf{t} \rightarrow \mathsf{endChan}.\mathsf{Closed} \rightarrow \mathsf{Lin}(\mathsf{t})\end{aligned}$$

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

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

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 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.

3.3. 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 timedout, it returns false. Otherwise, it continues as in the untimed operation, except it waits on continue until at most duration after the initial call. If it 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 stll hold x (the implementation asserts this, and the analysis below checks this). If the channel is closed, it continues as in the untimed operation. Otherwise, it must have timedout, so it sets status to Empty to clear its value, signals to any thread waiting on slotEmptied, and returns false.

The receiveWithin(duration) operation acts much as the untimed version, except it waits on slotFull until at most duration after the initial call. If it 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 timedout, so returns None.

We now describe the CSP model of these operations. Our analysis does not consider absolute time; thus we abstract away from the duration of a timed send or receive. The difficult part of the implementation is getting the synchronisations right, rather than the length of the delay.

The CSP model of an SCL monitor also models timed waits. It records which threads are doing timed waits on which conditions. Such threads can receive a signal, as for untimed waits. In addition, they can time out, and subsequently acquire the lock on the monitor. Thus we model that such threads can eventually time out, but don't model the length of the delay.

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

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

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

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

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

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

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

It is worth mentioning an alternative approach, which turns out not to work in this case. For standard concurrent datatypes, linearisability is often verified by identifying *linearisation points*: specific points in the program code where an invocation appears to take place. By analogy, can we identify *synchronisation points* in the program code for each operation, where the thread is aware of the result of the invocation, and then capture the correctness condition in terms of these synchronisation points? This idea won't work in this case, for the following reason. The synchronisation point for the sending thread would have to be after it receives the signal on continue and regains the lock, because it doesn't know the result of the synchronisation before this; but this might be too late, because the channel might have been closed before it obtained the lock, and so the synchronisation would appear incorrect.

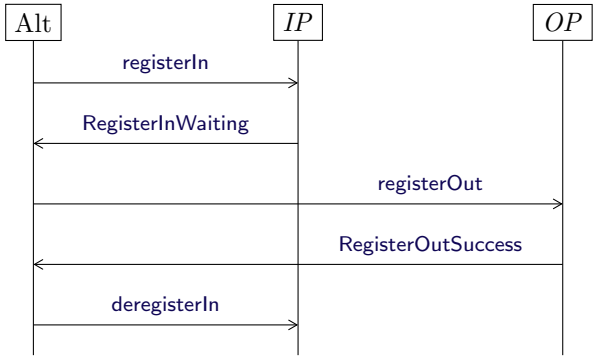


Fig. 2. Sequence diagram illustrating an alt that registers successfully with a port.

4. Alts

We now consider alts. 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.

4.1. High-level design

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

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

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

on each of its inports, where alt is a reference to the calling alt, index is the index of the branch within the alt, and iter is an iteration number within a serve construct (used only for assertions). 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 is willing to communicate, and the alt has received x from it;
RegisterInWaiting: the port is not currently willing to communicate (but the registration has been recorded);
RegisterInClosed: the port has been closed.

Similarly, the alt calls an operation

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

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

RegisterOutSuccess: the port is willing to communicate, and the alt has sent it a value;
RegisterOutWaiting: the port is not currently willing to communicate (but the registration has been recorded);
RegisterOutClosed: the port has been closed.

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

Figure 2 gives an example: an alt first registers unsuccessfully with an inport *IP*; then it registers successfully with an outport *OP*; and finally it deregisters from *IP*.

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

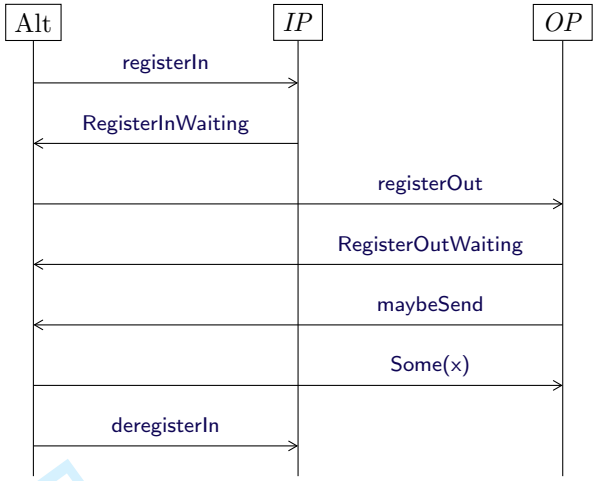


Fig. 3. Sequence diagram illustrating a successful callback from a port.

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

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

  on the alt, where value is the value it is trying to send to the alt, and index and iter match the values provided during registration. This asks the alt whether it is still willing to receive from the inport.
- If the alt is registered at the outport of a channel, and another thread tries to receive on the channel, it calls

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

  This asks the alt whether it is still willing to send a value to the inport.
- If another thread closes the channel, it calls

  def portClosed(index: Int, iter: Int)

If the alt receives a call of maybeReceive or maybeSend, it responds positively to the first such call, returning true to a maybeReceive, or Some(x) to a maybeSend, where x is the value it sends. It then deregisters from the remaining branches. Finally, the alt executes the continuation of the successful branch. Figure 3 gives an example of a successful callback of maybeSend from an outport 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 throws an AltAbort.

4.2. Implementation details

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

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

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

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

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

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.

The deregister operations simply clear the registration information.

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

4.3. CSP modelling for alts

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

A JVM monitor is modelled by a CSP module, in a similar way to an SCL monitor. 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 in the implementation of JVM monitors: a thread that is waiting for a signal may resume, even though it has received no signal! This is known as a *spurious wakeup*. The 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, allowing a waiting thread t to resume either as the result of a signal, or a spurious wakeup, modelled by the event spuriousWakeup.t. We run this monitor process in parallel with a *regulator process* $\mathsf{Reg} = \mathsf{CHAOS}(\{|\mathsf{spuriousWakeup}|\})$ that nondeterministically chooses whether or not to allow a spurious wakeup. This last point is important: if we allowed unrestricted spurious wakeups, there is a danger that our analysis would seem to show that suitable progress properties are satisfied, when in 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 don't model the expression that generates the value to be sent in an outport branch, but just pick

the value nondeterministically. Likewise, we don't model the continuations of branches. Each of these 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.

Finally, we don't model the iter parameter in the registration and deregistration operations, since it is used only in assertions as sanity checks, and would greatly increase the state space of the models.

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

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

Each use of the alt by an alt-thread t is framed with events beginAlt.t and endAlt.t.result, where result is of one of the following forms.

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

4.4. Direct analysis of an alt and channels

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

We build a system that uses an alt with a fixed collection of branches. Below, we consider an alt A1 with two branches (but the approach generalises to more branches). The branches are defined using a definition such as

branches = <InPortBranch.c1, OutPortBranch.c2>

In different tests, we can vary whether the branches correspond to inports or outports, so test different combinations. We also create two instances C1 and C2 of 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.

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, so we should consider whether the potential divergence is masking possible errors, by making critical states unstable. But recall that we included in the model of the monitor a regulator process that could block all spurious wakeups. Thus for every state that is unstable because of the possibility of a spurious wakeup, there is another, stable state where the regulator blocks the spurious wakeup. This way of abstracting the spurious wakeups corresponds to Roscoe's *lazy abstraction* [Ros10].

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

We extend the sync events to include the identity of the channel being used: sync.t_1.t_2.c.x 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 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 (recall that a channel-thread and alt-thread may use a port concurrently). Further, they allow synchronisations with the alt-thread.

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

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

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

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

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

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

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

We build a ChanSpec process for each channel. Each extends the earlier definition to allow altSync events, but only before the channel is closed. Further, each can perform allClosed when the channel is closed; we synchronise 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 different processes together, much as before. We can then verify that the system refines this specification in both the traces and stable-failures models, showing that the system is synchronisation linearisable and 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 inport branch and one outport 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	Time	States
D	872s	854M
F	319s	1.11B

5. Compositional verification

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

1. We show that a single channel (in isolation) is consistent with a more abstract model, which we call IdealisedChannel;
2. We likewise show that an alt (in isolation) is consistent with a more abstract model, which we call IdealisedAlt;
3. We show that the combination of an IdealisedAlt and several IdealisedChannels satisfies 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. This approach scales better than that in the previous section, so will allow us to deduce correctness for a larger number of threads.

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

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

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

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

where Any allows arbitrary behaviour (the definition depends upon the semantic model we are using). Thus the above process acts like Any after an error event.

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

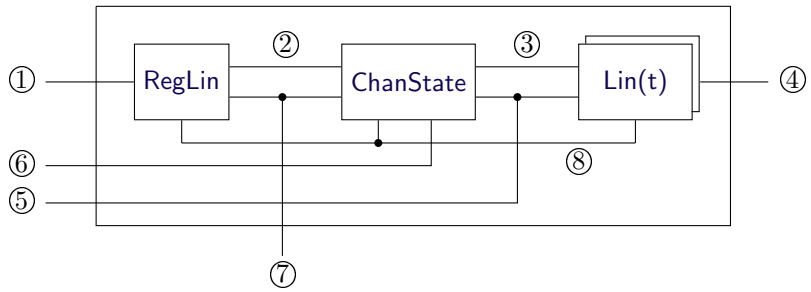
$$(\mathsf{IdealisedAlt}\ [|\mathsf{Errors}_A|\rangle\ \mathsf{Any})\ \setminus\ \mathsf{Errors}_A$$

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

We can then consider the combination of IdealisedAlt and several instances of IdealisedChannel. Part of the analysis will show that no events from $\mathsf{Errors}_C \cup \mathsf{Errors}_A$ occur: each component follows the alt protocol, provided the other has not previously broken the protocol. But we can also show that this combination satisfies the correctness property from the previous section, allowing us to deduce that the corresponding combination of an alt and channels satisfies that property.

5.1. Idealised model of a channel

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



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

①: (begin|end)Register(In|Out), (begin|end)Deregister(In|Out);
②: register(In|Out)Wait; deregister(In|Out), isClosed.AltThread;
③: sync, callMaybeSend, callMaybeReceive, close, isClosed, commit, timeout;
④: (begin|end)Send, (begin|end)Receive, (begin|end)SendWithin, (begin|end)ReceiveWithin, (begin|end)Close;
⑤: (begin|end)MaybeReceive, (begin|end)MaybeSend;
⑥: (begin|end)portClosed;
⑦: registerError, deregister(In|Out)Error;
⑧: register(In|Out)Sync.

Fig. 4. Construction of the idealised 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 4. 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 unobvious, and were found by trial and error: their correctness is evidenced by the subsequent successful refinement checks.

The RegLin process. The RegLin process deals with the linearisation of registration and deregistrations. It is defined in Figure 5; 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.

```
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. 5. Definition of the RegLin process, controlling registration and deregistration.

The Lin processes. Each Lin(t) process is responsible for linearising operations of channel-thread t. They are defined in Figure 6. 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 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.

```
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. 6. Definition of the Lin processes, controlling channel-thread operations.

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 7. 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 RegStatus represent that no alt is currently registered, or that an alt is registered at the inport or outport corresponding to a particular index.

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

Several possibilities are available when there is no registered alt.

- A registerIn or registerOut operation may synchronise with a waiting channel thread.
- A registerIn or registerOut operation may fail to synchronise, 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.

```
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. 7. The ChanState process.

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

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

Other possibilities are available regardless of the registration status.

- Two threads t1 and t2 may synchronise on a communication.
- A sending thread t may commit to returning (see the earlier explanation concerning the Lin(t) process).
- 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 4, 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.

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

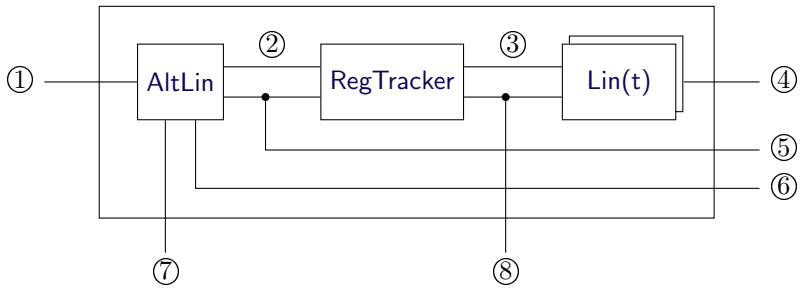
We can then compare the CSP model of a synchronous channel implementation against the idealised model. We create a system using the model of the implementation, allowing threads to call appropriate operations. We can then verify that this system refines $\mathsf{ChannelSpec}_F$ and $\mathsf{ChannelSpec}_D$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

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,



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. 8. Construction of the idealised alt.

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.

5.2. 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 InPortBranch.c and OutPortBranch.c for channels c. We assume a single alt-thread, AltThread, and a collection ChanThreads of channel-threads.

The idealised alt is constructed from several components, as depicted in Figure 8. The component 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 9. Each accepts the begin event of a callback function, which is handled by a different subsidiary process. Each callback may be either linearised correctly, or with an error event if it breaks the alt protocol: the RegTracker process selects the appropriate event, based on whether the alt is currently registered at the relevant port.

The AltLin process. The AltLin process, which linearises the main calls of the alt, is defined in Figure 10. It initially signals to the RegTracker that it is beginning registration, via event beginRegistration. It then iterates through branches, trying to register at each branch in turn, 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.

```
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. 9. Definition of the Lin(t) processes.

```
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. 10. Definition of the AltLin processes.

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.

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

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 inport, registered at an outport, or the port is closed. Initially all indices are marked as not registered.

The process RegTracker1 synchronises on the end events of call to registerIn and registerOut. In the case of a RegisterWaiting or RegisterClosed result, reg is updated to map the relevant index to an appropriate value. If a registration attempt is successful, the process moves to state RegTracker2, corresponding to the deregistration phase, described below. If the AltLin synchronises on endRegistration, indicating the end of the registration phase, the RegTracker sends a value indicating whether all the ports have been closed (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 inports, registered outports, 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 inport; at this point, the callback will be unsuccessful, as captured by the final false field in the event. Likewise, it can synchronise on the linearisation point of a maybeSend operation corresponding to a registered outport, which again will be unsuccessful. Further, it can synchronise on the linearisation point of a closed operation on a registered port. However, any callback not corresponding to a suitably registered port would be outside of the alt protocol, so an error is signalled.

The RegTracker3 process deals with the waiting phase, waiting for callbacks from registered ports. If can synchronise with Lin(t) on the linearisation point of a maybeReceive operation corresponding to a registered inport. The operation will be successful, as captured by the final true field in the event. It informs AltLin of the success, on channel endWait, and moves to the deregistration phase. Likewise, it can synchronise on the linearisation point of a maybeSend operation corresponding to a registered outport; this succeeds with a value x sent (modelled as being chosen nondeterministically). Further, it can synchronise on the linearisation point of a close operation; if all ports are now closed, it indicates this to AltSpec on channel endWait. However, any callback not corresponding to a suitably registered port would be outside of the alt protocol, so an error is signalled.

```
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. 11. The RegTracker process.

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

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

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

The table to the right gives statistics about these checks. In each case, we considered an alt with two branches, one inport branch and one outport 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 addition, the implementation model for an alt has fewer states than for a channel.

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

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.)

5.3. 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 4.4 (with the spuriousWakeup events hidden). The table to the right gives statistics about these checks, where branches contains one inport branch and one output branch.

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

In particular, the above stable-failures test verifies that the system does not perform any of the error events from $\mathsf{Errors}_A \cup \mathsf{Errors}_C$: this confirms that each component follows the alt protocol assuming the other has not previously broken the protocol.

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

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

6. 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 synchronisation points of operations. We believe that this style is widely applicable to other synchronisation objects. Our analysis revealed an error in a previous version of the code, but also helped us reach a correct implementation. 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. Further, there is a decoupling between the eternal interface —in terms of begin and end events— and where those invocations 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 must 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 3.3 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.

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.

[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.

[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.

[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.

[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.

[LL24] Jonathan Lawrence and Gavin Lowe. Understanding synchronisation. Draft paper in preparation, 2024.

[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.

[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.

[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.

[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.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1998.

[Ros10] A. W. Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[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.

[WM00] 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.

[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.

A. Modelling techniques

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

We model 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.

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

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)
```

By contrast, we normally model each thread-local variable as a parameter of the relevant process. This

reduces the number of events performed by processes, and so reduces the state-space of the system to be checked; however, it does increase the number of states in each individual process, and so increases the compilation time.

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

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

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

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

We now discuss the modelling of Scala functions in the implementation, in particular how to model the value returned. CSP has no notion of returning a value. Instead, we adopt a continuation-passing approach. If the Scala function returns a result of type A, the corresponding CSP process is given a parameter[3] cont :: A → Proc, representing a continuation, i.e. what the rest of the program does with the result. Returns from the Scala function are then modelled by applying cont to the returned value. A call to the function is modelled by providing a suitable function as the continuation.

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

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

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.

[3] Proc is the type of CSP processes.

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 4 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[4] $(f(tr), \{e \mid f^{-1}(\!|e|\!) \subseteq X\}) \in failures(P[T'])$;
- 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.

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 operation (i.e., that has been called but not returned); if that operation has synchronised, then the corresponding endSend event is not refused (i.e. is not in X); and if the operation has not synchronised, then the corresponding endSend event is refused. Likewise, suppose there is a pending receive operation; if that operation has synchronised with an invocation 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 operation has not synchronised, then all corresponding endReceive events of that thread are refused. Thus by examining this maximal failure, we can calculate which operations have synchronised. This prompts the following definition.

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

Definition 1. Let (tr, X) be a maximal failure of the system. We say that an invocation has *synchronised* if either there is a corresponding end event with a Success value (i.e. the invocation returned successfully) or such an end event is not refused.

The data value of an invocation send(v) is v. The data value of a synchronised invocation of receive is either the data value in the endReceive event, or (for a pending invocation) the data value in the available endReceive event.

Definition 2. We define the restriction of trace tr to data value v, written $tr|v$, to be the subtrace of tr containing: (1) events of invocations whose data value is v, and (2) the beginReceive events of unsynchronised invocations 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 3. 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 off the synchronised invocations for v, corresponding to the synchronisations, and such that no further such synchronisations are possible:

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

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

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

This implies that the whole failure is synchronisation progressible. □

Note 4. 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 invocation of send(A), a complete invocation of send(B), and a pending invocation 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 5. 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 3, 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 1 to sendWithin and receiveWithin in the obvious way. Given a maximal failure, we say that an invocation 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 invocation *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 invocations whose data value is v; (2) the begin events of unsynchronised invocations of receive and receiveWithin; (3) begin and end events of all invocations that detect closure; (4) begin and end events of all invocations 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 3, the fact that $(tr, X)|v$ is synchronisation progressible implies there is some way of choosing the first linearisation point of an invocation of close (if there is one) such that:

- all invocations that synchronise with data value v can be linearised before the linearisation of close;
- all invocations (for any data value) that time out can be linearised before the linearisation of close;
- all invocations (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 invocations that time out, by the first two bullet points above;
- The begin and end events of all invocations 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 invocation 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 invocation *synchronises to send* x similarly. We say that an invocation *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 invocations that send or receive v; and (2) beginAlt events of invocations 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.