# Synchronisation: Specification and Testing

Jonathan Lawrence

*The Blockhouse Technology Ltd., Oxford, UK*

Gavin Lowe*

*St Catherine's College, University of Oxford, UK*

**Abstract**

We study *synchronisation objects*: objects that allow two or more threads to synchronise, each waiting until the other threads have reached a particular point, and maybe exchanging data. We define a correctness condition for such synchronisation objects, which we call *synchronisation linearisation*: informally, the synchronisations appear to take place in a one-at-a-time order, consistent with the calls and returns of operations on the object, and giving correct results. We also define a liveness condition, which we call *synchronisation progressibility*: informally, executions of operations don't get stuck when a synchronisation is possible. We show that synchronisation linearisation can be reduced to a variant of standard linearisation, which we call *two-step linearisation*, where each operation of the synchronisation object is linearised in *two* steps.

We consider testing of implementations of synchronisation objects. The basic idea is to run several threads that use the object, record the history of operation calls and returns, and then test whether the resulting history satisfies synchronisation linearisation and progressibility. We present algorithms for this last step, and give results concerning the complexity of the problem. We describe an implementation of such a testing framework, and present experimental results.

*Keywords:* Concurrent programming, synchronisation, specification, linearisation, synchronisation linearisation, synchronisation progressibility, two-step linearisation, testing.

## 1. Introduction

In many concurrent programs, it is necessary at some point for two or more threads to *synchronise*: each of the threads waits until the other threads have

*Corresponding author.

*Email addresses:* jonathan@tbtl.com (Jonathan Lawrence), gavin.lowe@cs.ox.ac.uk (Gavin Lowe)

reached a particular point before continuing; in addition, the threads can exchange or combine data. Reasoning about programs can be easier when synchronisations are used: it helps to keep threads in consistent stages of the program, and so makes it easier to reason about the states of different threads. (The word "synchronisation" is used in a couple of different ways within concurrent programming: we use it in the sense just described, where every thread waits for the others, rather than for more general coordination between threads, such as via a semaphore which allows asynchrony between a signal and the receipt of the signal.)

We study synchronisations in this paper: we describe how synchronisations can be specified, and what it means for such a specification to be satisfied. We also describe techniques for testing implementations.

We start by giving some examples of synchronisations in order to illustrate the idea. (We use Scala notation; we explain non-standard aspects of the language in footnotes.) In each case, the synchronisation is mediated by a *synchronisation object*.

Perhaps the most common form of synchronisation object is a synchronous channel, e.g. [1, 2, 3, 4, 5, 6]. Such a channel might have signature[1]

```
class SyncChan[A]{
  def send(x: A): Unit
  def receive(): A
}
```

Each execution of one of the operations must synchronise with an execution of the other operation: the two executions must overlap in time. If an execution `send(x)` synchronises with an execution of `receive`, then the `receive` returns `x`.

Each synchronisation of a synchronous channel involves executions of two *different* operations (`send` and `receive`); we say that the synchronisation is *heterogeneous*. By contrast, sometimes two executions of the *same* operation may synchronise; we say that the synchronisation is *homogeneous*. For example, an *exchanger* [7, 8] has the following signature:

```
class Exchanger[A]{
  def exchange(x: A): A
}
```

When two threads call `exchange`, the executions can synchronise, and each receives the value passed in by the other.

For some synchronisation objects, synchronisations might involve more than two threads. For example, a *barrier synchronisation* object [3, 4, 9] can be used to synchronise `n` threads:

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

[1] The class is polymorphic in the type `A` of data. The type `Unit` is the type that contains a single value, the *unit value*, denoted `()`.

Each thread is assumed to have an integer thread identifier in the range $[0 .. \mathsf{n})$. Each thread me calls sync(me), and no execution returns until all n have called it. We say that the synchronisation has *arity* n.

A *combining barrier* [3, 9], in addition to acting as a barrier synchronisation, also allows each thread to submit a parameter, and for all to receive back some function of those parameters.[2]

```
class CombiningBarrier[A](n: Int, f: (A,A) => A){
  def sync(me: Int, x: A): A
}
```

The function f is assumed to be associative. If n threads call sync with parameters $x_1, \ldots, x_\mathsf{n}$, in some order, then each receives back $\mathsf{f}(x_1, \mathsf{f}(x_2, \ldots \mathsf{f}(x_{\mathsf{n}-1}, x_\mathsf{n}) \ldots))$. (In the common case that f is commutative, this result is independent of the order of the parameters.)

Some synchronisation objects have multiple modes of synchronisation. For example, consider a synchronous channel with timeouts: each execution might synchronise with another execution, or might timeout without synchronisation [5, 9]. Such a channel has a signature as follows.

```
class TimeoutChannel[A]{
  def send(x: A): Boolean
  def receive(): Option[A]
}
```

The send operation returns a boolean to indicate whether the send was successful, i.e. whether it synchronised. The receive operation can return a value Some(x) to indicate that it synchronised and received x, or can return the value None to indicate that it failed to synchronise[3]. Thus an execution of each operation may or may not synchronise with an execution of the other operation. Unsuccessful executions of send and receive can be considered *unary* synchronisations.

Similarly, a timeout exchanger [7, 8] can allow threads to exchange; but if a thread fails to exchange, it can return without synchronising. It has a signature as follows.

```
class TimeoutExchanger[A]{
  def exchange(x: A): Option[A]
}
```

So far, our example synchronisation objects have been *stateless*: they maintain no state from one synchronisation to another. By contrast, some synchronisation objects are *stateful*: they maintain some state between synchronisations, which might affect synchronisations. As a toy example, consider a synchronous channel that maintains a sequence counter, and such that both executions receive the current value of this counter.

[2] The Scala type (A,A) => A represents functions from pairs of A to A.
[3] The type Option[A] contains the union of such values.

```
class SyncChanCounter[A]{
  private var counter: Int
  def send(x: A): Int // Result is sequence counter.
  def receive(): (A, Int) // Result is (value received, sequence counter).
}
```

Some implementations of synchronous channels allow the channel to be closed [4, 5], say by a unary operation close.

```
class CloseableChan[A]{
  def send(x: A): Unit
  def receive(): A
  def close(): Unit
}
```

Calls to send or receive after the channel is closed throw an exception. Thus such an object is stateful, with two states, open and closed; and the operations have different modes of synchronisation, either successful or throwing an exception.

An *enrollable barrier* [10] is a barrier that allows threads to enrol and resign (via unary operations):

```
class EnrollableBarrier(n: Int){
  def sync(me: Int): Unit
  def enrol(me: Int): Unit
  def resign(me: Int): Unit
}
```

Each barrier synchronisation is between all threads that are currently enrolled, so sync has a variable arity. The barrier has a state, namely the currently enrolled threads.

A *terminating queue* can also be thought of as a stateful synchronisation object with multiple modes. Such an object mostly acts like a standard partial concurrent queue: if a thread attempts to dequeue, but the queue is empty, it blocks until the queue becomes non-empty. However, if a state is reached where all the threads are blocked in this way, then they all return a special value to indicate this fact. In some concurrent algorithms, such as a concurrent graph search, this latter outcome indicates that the algorithm should terminate. Such a terminating queue might have the following signature, where a dequeue returns the value None to indicate the termination case.

```
class TerminatingQueue[A](n: Int){ // n is the number of threads.
  def enqueue(x: A): Unit
  def dequeue: Option[A]
}
```

The termination outcome can be seen as a synchronisation between all n threads. This terminating queue combines the functionality of a concurrent datatype and a synchronisation object.

In this paper, we consider what it means for such a synchronisation object

to be correct. We also present techniques for testing correctness of implementations.

In Section 2 we describe how to specify a synchronisation object: we call the property *synchronisation linearisation*. The definition has similarities with that of *linearisation* [11, 8]. Linearisation is the standard correctness property for concurrent datatypes (by which we mean implementations of abstract dataypes, such as sets, mappings, stacks and queues, that support concurrent operations). However, synchronisation linearisation talks about synchronisations between executions of operations, whereas linearisation talks about single executions. Informally, the synchronisations should appear to take place in a one-at-a-time order, consistent with the calls and returns of operations on the synchronisation object. We present a way to specify what sequences of synchronisations and what return values are consider correct, via a *synchronisation specification object*.

In fact, our property of synchronisation linearisation turns out to be the same as *set linearisation* [12], also known as *concurrency-aware linearisation* [13], where a set of operation executions appear to take place at the same time. We compare these approaches with our own in Section 9.1. We prefer the name "synchronisation linearisation" here, because it best describes our intention.

We also define a liveness condition, which we call *synchronisation progressibility*: informally, executions don't get stuck when a synchronisation is possible.

In Section 3 we consider the relationship between synchronisation linearisation and (standard) linearisation. We show that linearisation is an instance of synchronisation linearisation, but that synchronisation linearisation is more general. We also show that synchronisation linearisation corresponds to a small adaptation of linearisation, where an operation of the synchronisation object may correspond to *two* operations of the object used to specify linearisation; we call this *two-step linearisation*.

We then consider testing of synchronisation object implementations. Our experience from teaching students is that they often do not have a clear idea about how to test a synchronisation object (we suspect the same is true of other programmers). Yet, implementing synchronisation objects is tricky—subtle bugs are fairly common—and so good tests are important.

Our testing techniques are based on the techniques for testing (standard) linearisation [14, 15], which we review in Section 4: the basic idea is to record a history of threads using the object, and then to check whether that history is linearisable. In Section 5 we show how this technique can be adapted to test for synchronisation linearisation, using the result of Section 3, where an operation of the synchronisation object may correspond to two operations of the specification object.

In Section 6 we show how synchronisation linearisation can be tested more directly: we describe algorithms that check whether a history of a synchronisation object is synchronisation linearisable. We also present various complexity results. Deciding whether a given history is synchronisation linearisable is NP-complete in general, in the stateful case. However, it can be decided in polynomial time in the case of binary (heterogeneous or homogeneous) state-

less synchronisation objects. But moving to synchronisations with arity greater than 2 is again NP-complete, even in the stateless case.

We describe the implementation of a testing framework in Section 7; the framework supports both two-step linearisation and the direct algorithms. In Section 8 we describe experiments to determine the effectiveness of the testing techniques: both find errors in faulty implementations of synchronisation objects very quickly. We sum up and discuss related work in Section 9.

We consider our main contributions to be as follows.

- An exploration of the range of different synchronisation objects;
- A general technique for specifying synchronisation objects, capturing both safety and liveness properties;
- A study of the relationship between synchronisation linearisation and standard linearisation;
- Algorithms for deciding whether a history of a synchronisation object is synchronisation linearisable, together with related complexity results;
- A testing framework for synchronisation objects, and an experimental assessment of its effectiveness.

## 2. Specifying synchronisations

In this section we describe how synchronisations can be formally specified. We start by considering *heterogeneous binary* synchronisation, i.e. where every synchronisation is between executions of *two different* operations. We allow stateful synchronisation objects (which includes stateless objects as degenerate cases). We generalise in Section 2.4.

For the moment, we assume that the synchronisation object has two operations, each of which has a single parameter, as follows.

```
def op1(x1: A1): B1
def op2(x2: A2): B2
```

(We can model a concrete operation that takes $k \neq 1$ parameters by an operation that takes a $k$-tuple as its parameter. We identify a 0-tuple with the unit value, but will sometimes omit that value in examples.) In addition, the synchronisation object might have some state. Each execution of $\mathsf{op}_1$ must synchronise with an execution of $\mathsf{op}_2$, and vice versa. The result of each execution may depend on the two parameters, $\mathsf{x}_1$ and $\mathsf{x}_2$, and the current state. In addition, the state may be updated. The external behaviour is consistent with the synchronisation happening atomically at some point within the duration of both operation executions (which implies that the executions must overlap): we refer to this point as the *synchronisation point*.

Synchronisation linearisation is defined in terms of a *synchronisation specification object*: we define these specification objects in the next subsection.

In Section 2.2, we review the notion of linearisation, on which synchronisation linearisation is based. We then define synchronisation linearisation for binary heterogeneous synchronisation objects in Section 2.3. We generalise to other classes of synchronisation objects in Section 2.4. In Section 2.5, we show that our definition satisfies *locality*: a collection of objects satisfies synchronisation linearisability if and only if each individual object does. We present our liveness property, synchronisation progressibility, in Section 2.6.

### *2.1. Synchronisation specification objects*

Each synchronisation object, with a signature as above, can be specified using a *synchronisation specification object* with the following signature.

```
class Spec{
  def sync(x1: A1, x2: A2): (B1, B2)
}
```

The idea is that if two executions $\mathsf{op}_1(\mathsf{x}_1)$ and $\mathsf{op}_2(\mathsf{x}_2)$ synchronise, then the results $\mathsf{y}_1$ and $\mathsf{y}_2$ of the executions are such that $\mathsf{sync}(\mathsf{x}_1, \mathsf{x}_2)$ returns the pair $(\mathsf{y}_1, \mathsf{y}_2)$. The specification object might have private state, which can be accessed and updated within sync. Note that executions of sync occur *sequentially*.

Informally, the synchronisation specification object can be seen as an idealised description of the effects of the synchronisation, as if—instead of calling $\mathsf{op}_1$ and $\mathsf{op}_2$—the two threads had jointly called sync, each supplying one parameter, and each taking one component of the result.

In general, sync could be nondeterministic, and so allow several different results. However, in all our examples, sync will be deterministic; and we will require it to be deterministic when we consider testing, from Section 5 onwards.

We formalise below what it means for a synchronisation object to satisfy the requirements of a synchronisation specification object. But first, we give some examples to illustrate the style of specification.

A generic definition of a specification object might take the following form:

```
class Spec{
  private var state: S
  def sync(x1: A1, x2: A2): (B1, B2) = {
    require(guard(x1, x2, state))
    val res1 = f1(x1, x2, state); val res2 = f2(x1, x2, state)
    state = update(x1, x2, state)
    (res1, res2)
  }
}
```

The object has some local state, which persists between executions. The require clause of sync specifies a precondition for the synchronisation to take place: that precondition is described by the boolean function guard. The values $\mathsf{res}_1$ and $\mathsf{res}_2$ represent the results that should be returned by the corresponding executions of $\mathsf{op}_1$ and $\mathsf{op}_2$, respectively. The function update describes how the local state should be updated.

For example, consider a synchronous channel with operations

```
def send(x: A): Unit
def receive(u: Unit): A
```

(Note that we model the receive operation as taking a parameter of type Unit, in order to fit our uniform setting.) This can be specified using a synchronisation specification object with empty state:

```
class SyncChanSpec[A]{
  def sync(x: A, u: Unit): (Unit, A) = ((), x)
}
```

If send(x) synchronises with receive(()), then the former receives the unit value (), and the latter receives x.

As another example, consider a filtering channel.

```
class FilterChan[A]{
  def send(x: A): Unit
  def receive(p: A => Boolean): A
}
```

Here the receive operation is passed a predicate p describing a required property of any value received. This can be specified using a stateless specification object with operation

```
def sync(x: A, p: A => Boolean): (Unit, A) = { require(p(x)); ((), x) }
```

The require clause specifies that executions send(x) and receive(p) can synchronise only if p(x).

As an example illustrating the use of state in the synchronisation object, recall the synchronous channel with a sequence counter, SyncChanCounter, from the introduction. This can be specified using the following specification object.

```
class SyncChanCounterSpec[A]{
  private var counter = 0
  def sync(x: A, u: Unit): (Int, (A, Int)) = {
    counter += 1; (counter, (x, counter))
  }
}
```

Each synchronisation increments the counter, and the new value is returned to each thread.

### *2.2. Linearisation*

We formalise the allowable behaviours captured by a particular synchronisation specification object. Our definition has much in common with the well known notion of *linearisation* [11], used for specifying concurrent datatypes: with linearisation, operation invocations appear to happen in a one-at-a-time order; for a synchronisation object, we want to capture that synchronisations appear to happen in a one-at-a-time order. We start by reviewing linearisation. There are a number of equivalent ways of defining it: we choose a way that will be convenient subsequently.

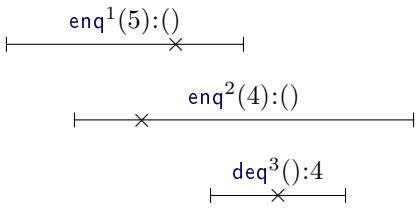



Figure 1: Timeline representing the linearisation example. Time runs from left to right; each horizontal line represents an operation execution, with the left-hand end representing the call event, and the right-hand end representing the return event.

A *concurrent history* of an object $o$ (either a concurrent datatype or a synchronisation object) records the calls and returns of operations on $o$. It is a sequence of events of the following forms:

- call.$op^i(x)$, representing a call of operation $op$ with parameter $x$;
- return.$op^i$:$y$, representing a return of an execution of $op$, giving result $y$.

Here $i$ is an *execution identity*, used to identify a particular execution, and to link the call and corresponding return. In order to be well formed, each execution identity must appear on at most one call event and at most one return event; and for each event return.$op^i$:$y$, the history must contain an earlier event call.$op^i(x)$, i.e. for the same operation and execution identity. We consider only well formed histories from now on.

We say that a call event and a return event *match* if they have the same execution identifier. A concurrent history is *complete* if for every call event, there is a matching return event, i.e. no execution is still pending at the end of the history.

For example, consider the following complete concurrent history of a concurrent datatype that is intended to implement a queue, with operations enq and deq.

$$\begin{aligned} h \;=\; \langle & \mathsf{call}.\mathsf{enq}^1(5),\ \mathsf{call}.\mathsf{enq}^2(4),\ \mathsf{call}.\mathsf{deq}^3(),\\ & \mathsf{return}.\mathsf{enq}^1{:}(),\ \mathsf{return}.\mathsf{deq}^3{:}4,\ \mathsf{return}.\mathsf{enq}^2{:}()\rangle. \end{aligned}$$

This history is illustrated by the timeline in Figure 1.

Linearisation is specified with respect to a linearisation specification object *Spec*, with the same operations (and signatures) as the concurrent datatype in question. A history of the specification object is a sequence of events of the form:

- $op^i(x)$:$y$ representing an execution of operation $op$ with parameter $x$, returning result $y$; again $i$ is an execution identity, which must appear at most once in the history.

A history is *legal* if it is consistent with the definition of *Spec*, i.e. for each operation execution, the precondition is satisfied, and the return value is as for the definition of the operation in *Spec*. In general, the specification object could be nondeterministic, and so allow several values that could be returned by an operation execution (although in all our examples it will be deterministic).

For example, consider the history

$$h_s \quad = \quad \langle \mathsf{enq}^2(4){:}(),\ \mathsf{enq}^1(5){:}(),\ \mathsf{deq}^3(){:}4 \rangle.$$

This is a legal history for a specification object that represents a queue. This history is illustrated by the "×"s in Figure 1.

Let $h$ be a complete concurrent history, and let $h_s$ be a legal history of the specification object *Spec*. We say that $h$ and $h_s$ *correspond* if they contain the same executions, i.e., for each $\mathsf{call}.op^i(x)$ and $\mathsf{return}.op^i{:}y$ in $h$, $h_s$ contains $op^i(x){:}y$, and vice versa. We say that $h_s$ is a *linearisation* of $h$ if there is some way of interleaving the two histories (i.e. creating a history containing the events of $h$ and $h_s$, preserving the order of events) such that each $op^i(x){:}y$ occurs between $\mathsf{call}.op^i(x)$ and $\mathsf{return}.op^i{:}y$. Informally, this indicates that the executions of $h$ appeared to take place in the order described by $h_s$, and that this order is legal according to the specification object. We say that $h$ is *linearisable* with respect to *Spec* in this case.

Continuing the running example, $h_s$ is a linearisation of $h$, as evidenced by the interleaving

$$\begin{array}{l}\langle \mathsf{call}.\mathsf{enq}^1(5),\ \mathsf{call}.\mathsf{enq}^2(4),\ \mathsf{enq}^2(4){:}(),\ \mathsf{enq}^1(5){:}(),\ \mathsf{call}.\mathsf{deq}^3(),\\ \ \mathsf{return}.\mathsf{enq}^1{:}(),\ \mathsf{deq}^3{:}4,\ \mathsf{return}.\mathsf{deq}^3{:}4,\ \mathsf{return}.\mathsf{enq}^2{:}() \rangle,\end{array}$$

as illustrated in Figure 1. The points at which the events of $h_s$ are inserted into $h$ can be thought of as the points where each operation has an effect; we refer to these as *linearisation points*.

A concurrent history might not be complete, i.e. it might have some pending executions that have been called but have not returned. An *extension* of a history $h$ is formed by adding zero or more return events corresponding to pending executions. We write $\mathit{complete}(h)$ for the subsequence of $h$ formed by removing all call events corresponding to pending executions.

We say that a (not necessarily complete) concurrent history $h$ is *linearisable* with respect to specification object *Spec* if there is an extension $h'$ of $h$ such that $\mathit{complete}(h')$ is linearisable with respect to *Spec*. Informally, the return events that are in $h'$ but not $h$ are for operation executions that have had an effect, but not returned in $h$; the call events removed in $\mathit{complete}(h')$ are for operation executions that have not yet had an effect.

We say that a concurrent datatype is linearisable with respect to *Spec* if each of its histories is linearisable with respect to *Spec*.

### *2.3. Synchronisation linearisation*

We now adapt the definition of linearisation to synchronisations. With standard linearisation, operations appear to take place in a one-at-a-time order,

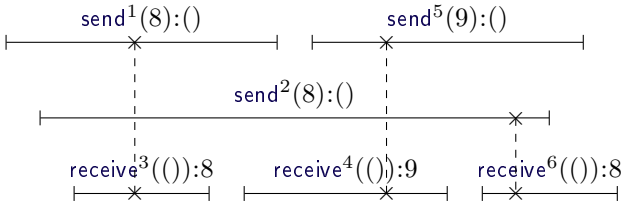



Figure 2: Timeline representing the synchronisation example.

each between the time at which the operation is invoked and when it returns. With synchronisation linearisation, synchronisations appear to take place in a one-at-a-time order, with each synchronisation between the time that each of the operations is invoked and when it returns. In each case, the order is one that satisfies the requirements captured by the specification object.

For the moment, we consider only binary heterogeneous synchronisations; we generalise in the next section. We consider a synchronisation object *Sync* with two operations, $\mathsf{op}_1$ and $\mathsf{op}_2$, as described earlier. A concurrent history of *Sync* contains call and return events, as in the previous subsection, corresponding to the operations $\mathsf{op}_1$ and $\mathsf{op}_2$.

For example, the following is a complete history of the synchronous channel from earlier, and is illustrated in Figure 2:

$$\begin{aligned} h \;=\; \langle & \mathsf{call}.\mathsf{send}^1(8),\ \mathsf{call}.\mathsf{send}^2(8),\ \mathsf{call}.\mathsf{receive}^3(()),\ \mathsf{return}.\mathsf{receive}^3{:}8, \\ & \mathsf{call}.\mathsf{receive}^4(()),\ \mathsf{return}.\mathsf{send}^1{:}(),\ \mathsf{call}.\mathsf{send}^5(9),\ \mathsf{return}.\mathsf{receive}^4{:}9, \\ & \mathsf{call}.\mathsf{receive}^6(()),\ \mathsf{return}.\mathsf{send}^2{:}(),\ \mathsf{return}.\mathsf{send}^5{:}(),\ \mathsf{return}.\mathsf{receive}^6{:}8\rangle. \end{aligned}$$

A history of a synchronisation specification object *Spec* is a sequence of events of the form $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$, representing an execution of sync with parameters $(x_1,x_2)$ and result $(y_1,y_2)$. The event's identity is $(i_1,i_2)$: each of $i_1$ and $i_2$ must appear at most once in the history. Informally, an event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ corresponds to a synchronisation between executions $\mathsf{op}_1^{i_1}(x_1){:}y_1$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ in a history of the corresponding synchronisation object.

A history is *legal* if is consistent with the definition of the specification object. For example, the following is a legal history of SyncChanSpec.

$$h_s \;=\; \langle \mathsf{sync}^{1,3}(8,()){:}((),8),\ \mathsf{sync}^{5,4}(9,()){:}((),9),\ \mathsf{sync}^{2,6}(8,()){:}((),8)\rangle.$$

The history is illustrated by the "×"s in Figure 2: each event corresponds to the synchronisation of two operations, so is depicted by two "×"s on the corresponding operations, linked by a dashed vertical line. This particular synchronisation specification object is stateless, so in fact any permutation of the history $h_s$ would also be legal (but not all such permutations will be compatible with the history of the synchronisation object); but the same will not be true in general of a specification object with state. The points at which the events of $h_s$ are

inserted into $h$ can be thought of as the points where each synchronisation takes place; we refer to these as *synchronisation points*.

**Definition 1.** Let $h$ be a complete history of the synchronisation object *Sync*. We say that a legal history $h_s$ of *Spec corresponds* to $h$ if their events agree; more precisely:

- For each $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ in $h_s$, $h$ contains events $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$, $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$ , $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$, and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$.
- For each $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$ in $h$, $h_s$ contains an event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ for some $i_2$, $x_2$, and $y_2$.
- For each $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$ in $h$, $h_s$ contains an event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ for some $i_1$, $x_1$, and $y_1$.

**Definition 2.** Given a complete history $h$ of *Sync* and a corresponding legal history $h_s$ of *Spec*, we say that $h_s$ is a *synchronisation linearisation* of $h$ if there is some way of interleaving $h$ and $h_s$ such that each event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ occurs between $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$, and between $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$.

In the running example, $h_s$ is a synchronisation linearisation of $h$, as shown by the interleaving in Figure 2.

**Definition 3.** Given a (not necessarily complete) concurrent history $h$ and a corresponding legal history $h_s$ of *Spec*, we say that $h_s$ is a *synchronisation linearisation* of $h$ if there is an extension $h'$ of $h$ such that $h_s$ is a synchronisation linearisation of $\mathit{complete}(h')$. We say that $h$ is synchronisation-linearisable with respect to *Spec* in this case. We say that a synchronisation object is synchronisation-linearisable with respect to *Spec* if each of its histories is synchronisation-linearisable with respect to *Spec*.

Informally, the $\mathsf{return}$ events that are in $h'$ but not $h$ are for operation executions that have synchronised, but not returned in $h$; the $\mathsf{call}$ events removed in $\mathit{complete}(h')$ are for operation executions that have not yet synchronised.

As noted in the Introduction, our notion of synchronisation linearisation is equivalent to *set linearisation* [12], although no formal definition was given there.

The same property was rediscovered by Hemed et al. [13], and called *concurrency-aware linearisation* (CAL). They define a concurrency-aware trace (CA-trace) to be a history of *sets* of operation executions, where each set represents operation executions that seem to take effect simultaneously (such as those in a synchronisation). They then define CAL in terms of those CA traces, in a way that is equivalent to our definition. Unlike us, they do not provide a way of specifying correct CA-traces, comparable to out synchronisation specification objects: they simply assume a given set of correct CA-traces. They verify an exchanger object with respect to CAL, and then use this result in a modular verification of an elimination stack [16].

*2.4. Variations*

Above we considered heterogeneous binary synchronisations, i.e. synchronisations between *two* executions of *different* operations, with a single mode of synchronisation. In this section, we generalise. There are two aspects of this: defining the effect of a synchronisation, via a synchronisation specification object; and defining which operations of the synchronisation object can synchronise together, and relating them to the corresponding operation of the synchronisation specification object.

It is straightforward to generalise to synchronisations between an arbitrary number of operation executions, some of which might be executions of the same operation. Consider a $k$-way synchronisation between operations

**def** $\mathsf{op}_j(\mathsf{x}_j\colon \mathsf{A}_j)\colon \mathsf{B}_j$ for $j = 1, \ldots, k$,

where the $\mathsf{op}_j$ might not be distinct. The specification object will have a corresponding operation of the form

**def** $\mathsf{sync}(\mathsf{x}_1\colon \mathsf{A}_1, \ldots, \mathsf{x}_k\colon \mathsf{A}_k)\colon (\mathsf{B}_1, \ldots, \mathsf{B}_k)$

For example, for the combining barrier CombiningBarrier(n, f) of the Introduction, the corresponding specification object would be[4]

```
class CombiningBarrierSpec{
  def syncA((id1,x1): (Int,A) ..., (idn,xn): (Int,A)) = {
    val result = f(x1, f(x2,...f(xn−1, xn)...)); (result,...,result)
  }
}
```

We define the relationship between operations of the synchronisation object and of the synchronisation specification object via a *synchronisation abstraction function*: this is a partial function that maps a (non-empty) sequence of operations of the synchronisation object to the corresponding operation of the synchronisation specification object. For the combining barrier, the abstraction function is

$$\{\langle \mathsf{sync}((id_1, x_1)), \ldots, \mathsf{sync}((id_\mathsf{n}, x_\mathsf{n}))\rangle \mapsto \mathsf{syncA}((id_1, x_1), \ldots, (id_\mathsf{n}, x_\mathsf{n})) \mid id_1, \ldots id_\mathsf{n} \in \mathsf{Int}, x_1, \ldots, x_\mathsf{n} \in \mathsf{A}\}.$$

For the generic binary synchronisations we considered in Section 2.3, the abstraction function is

$$\{\langle \mathsf{op}_1(x_1), \mathsf{op}_2(x_2)\rangle \mapsto \mathsf{sync}(x_1, x_2) \mid x_1 \in \mathsf{A}_1, x_2 \in \mathsf{A}_2\}.$$

For a synchronisation object with multiple modes of synchronisation, the synchronisation specification object has a different operation for each mode. For example, recall the TimeoutChannel from the Introduction, where sends and

[4] We change the name of the operation to syncA, to avoid a name-clash with the operation sync of CombiningBarrier.

receives may timeout and return without synchronisation. The corresponding specification object is as follows (we omit arguments of type Unit, to improve readability).

```
class TimeoutChannelSpec[A]{
  def sendTO(x: A) = false // send times out.
  def receiveTO() = None // receive times out.
  def sync(x: A) = (true, Some(x)) // Synchronisation.
}
```

The operation sendTO corresponds to a send timing out (and so not synchronising), receiveTO corresponds to a receive timing out, and sync corresponds to a send and receive synchronising. This is formalised by the synchronisation abstraction function:

$$\{\langle \mathsf{send}(x)\rangle \mapsto \mathsf{sendTO}(x) \mid x \in \mathsf{A}\} \cup \{\langle \mathsf{receive}()\rangle \mapsto \mathsf{receiveTO}()\} \cup$$
$$\{(\langle \mathsf{send}(x), \mathsf{receive}()\rangle \mapsto \mathsf{sync}(x) \mid x \in \mathsf{A}\}.$$

As another example, the following is a specification object for a channel with a close operation.

```
class ClosableChannelSpec[A]{
  private var isClosed = false // Is the channel closed?
  def close() = { isClosed = true; () }
  def sync(x: A) = { require(!isClosed); ((), x) }
  def sendFail(x: A) = { require(isClosed); throw new Closed }
  def receiveFail() = { require(isClosed); throw new Closed }
}
```

The synchronisation abstraction function is

$$\{\langle \mathsf{close}()\rangle \mapsto \mathsf{close}()\} \cup \{\langle \mathsf{send}(x), \mathsf{receive}()\rangle \mapsto \mathsf{sync}(x) \mid x \in \mathsf{A}\} \cup$$
$$\{\langle \mathsf{send}(x)\rangle \mapsto \mathsf{sendFail}(x) \mid x \in \mathsf{A}\} \cup \{\langle \mathsf{receive}()\rangle \mapsto \mathsf{receiveFail}()\}.$$

A send and receive can synchronise corresponding to sync, but only before the channel is closed; or each may fail once the channel is closed, corresponding to sendFail and receiveFail.

We now describe how to generalise the definition of synchronisation linearisation. Fix a synchronisation specification object *Spec* and corresponding synchronisation abstraction function $\mathcal{A}$. For each operation

```
  def sync(x1: A1,...,xn: Ak): (B1,...,Bk)
```

of *Spec* (corresponding to $k$ threads synchronising), we introduce events $sync^{i_1,\ldots,i_k}(x_1,\ldots,x_k){:}(y_1,\ldots,y_k)$ (where, for each $j$, $x_j \in A_j$, $y_j \in B_j$, and $i_j$ is an execution identity). A legal history of *Spec* is then a sequence of such events, consistent with the definition of *Spec*.

**Definition 4.** Let $h$ be a complete history of the synchronisation object. Then a legal history $h_s$ of *Spec* corresponds to $h$ if:

- For each event $sync^{i_1,\ldots,i_k}(x_1,\ldots,x_k){:}(y_1,\ldots,y_k)$ in $h_s$, there is some maplet $\langle op_1(x_1),\ldots,op_k(x_k)\rangle \mapsto sync(x_1,\ldots,x_k)$ in $\mathcal{A}$ such that, for each $j$, $h$ contains an execution of $op_j(x_j)^{i_j}{:}y_j$.
- For each execution $op_j(x_j)^{i_j}{:}y_j$ in $h$, $h_s$ contains some event of the form $sync^{i_1,\ldots,i_k}(x_1,\ldots,x_k){:}(y_1,\ldots,y_k)$ (necessarily the abstraction function $\mathcal{A}$ relates operations as in the previous bullet point).

Then $h_s$ is a synchronisation linearisation of $h$ if there is some way of interleaving $h$ and $h_s$ such that each event $sync^{i_1,\ldots,i_k}(x_1,\ldots,x_k){:}(y_1,\ldots,y_k)$ is between $\mathsf{call}.op_j^{i_j}(x_j)$ and $\mathsf{return}.op_j^{i_j}{:}y_j$, for each $j$.

The definition of synchronisation linearisability then follows precisely as in Definition 3.

### *2.5. Locality*

So far, we have implicitly assumed a single synchronisation object. However, a program may employ multiple synchronisation objects. Consider a set of independent synchronisation objects $\mathcal{O}$. Given a history $h$ and an object $o$, we write $h|o$ for the restriction of $h$ to events of $o$. We assume that the objects are independent, so $h$ is a history of $\mathcal{O}$ if and only if $h|o$ is a history of $o$ for each $o$.

We assume that each synchronisation object $o$ has its own synchronisation specification object $Spec_o$ and abstraction function $\mathcal{A}_o$. We can combine the specification objects into a single specification object $Spec$, with the operations of all $Spec_o$ (assumed distinct): a history $h_s$ is legal if and only if $h_s|Spec_o$ is legal for each $Spec_o$. Likewise we can combine the abstraction functions into a single abstraction function $\mathcal{A}$, by taking their union.

We show that synchronisation linearisation satisfies a locality property: the combined objects $\mathcal{O}$ are synchronisation-linearisable if and only if each individual object is synchronisation-linearisable. This is an important result, as it allows for compositional development and verification. (Neiger [12] states a similar result.)

**Lemma 5.** Let $h$ be a complete history of the synchronisation objects $\mathcal{O}$. Then $h$ is synchronisation-linearisable (with respect to $Spec$ and $\mathcal{A}$) if and only if $h|o$ is synchronisation-linearisable (with respect to $Spec_o$ and $\mathcal{A}_o$), for each $o \in \mathcal{O}$.

Proof. The left-to-right implication is trivial.

For the right-to-left implication, suppose that $h|o$ is synchronisation-linearisable, for each $o$. That means that there is a legal history $h_{s_o}$ of $Spec_o$ that is a synchronisation linearisation of $h|o$. Write $h_o^I$ for the interleaving of these histories that demonstrates this (following Definition 4).

We build an interleaving $h^I$ of $h$ and all $h_{s_o}$ for $o \in \mathcal{O}$, consistent with the $h_o^I$ interleavings: if, in $h_o^I$, a $\mathsf{call}$ or $\mathsf{return}$ event $e$ of $o$ is followed by events $s_1,\ldots,s_k$ of $h_{s_o}$, then we insert $s_1,\ldots,s_k$ into $h$ directly after $e$.

In the resulting interleaving $h^I$, each synchronisation event is still between the relevant $\mathsf{call}$ and $\mathsf{return}$ events, by construction. Let $h_s$ be the restriction of $h^I$

to the events of the specification objects. Then for each $o$, $h_s|Spec_o = h_{s_o}$, by construction, so is legal. Hence $h_s$ is legal.

**Proposition 6.** The collection of objects $\mathcal{O}$ is synchronisation-linearisable (with respect to $Spec$ and $\mathcal{A}$) if and only if $o$ is synchronisation-linearisable (with respect to $Spec_o$ and $\mathcal{A}_o$), for each $o \in \mathcal{O}$.

Proof. The left-to-right implication is again trivial.

For the right-to-left implication, consider a (not necessarily complete) history $h$ of $\mathcal{O}$. For each $o$, since $o$ is synchronisation-linearisable, there is an extension $h'_o$ of $h|o$ such that $complete(h'_o)$ is synchronisation-linearisable with respect to $Spec_o$ and $\mathcal{A}_o$. Let $h'$ be an extension of $h$, adding the events in each $h'_o$ but not $h_o$ in the same order (but interleaving events from different objects arbitrarily). Then $complete(h')|o = complete(h'_o)$, for each $o$. Also, $complete(h')$ is a complete history of $\mathcal{O}$, and by Lemma 5, $complete(h')$ is synchronisation-linearisable with respect to $Spec$ and $\mathcal{A}$, as required.

*2.6. Specifying liveness*

We now consider a liveness condition for synchronisation objects. Fix a synchronisation object SyncObj and a synchronisation specification object SyncSpec. We assume that SyncObj is synchronisation-linearisable with respect to SyncSpec.

The liveness condition can be stated informally as:

- If some pending operation executions can synchronise (according to the synchronisation specification object), then some such synchronisation should happen;
- Once a particular execution has synchronised, it should eventually return.

Several different synchronisations might be possible. For example, consider a synchronous channel, and suppose there are pending calls to send(4), send(5) and receive. Then the receive could synchronise with *either* send, nondeterministically; subsequently, the receive should return the appropriate value, and the corresponding send should also return. In such cases, our liveness condition allows *either* synchronisation to occur. However, our liveness condition allows all pending executions to block if no synchronisation is possible. For example, if every pending execution on a synchronous channel is a send, then clearly none can make progress.

The following definition identifies maximal sequences of synchronisations that could occur given a particular history of a synchronisation object.

**Definition 7.** Given a history $h$ of the synchronisation object and a legal history $h_s$ of the specification object, we say that $h_s$ is a *maximal synchronisation linearisation* of $h$ if:

- $h_s$ is a synchronisation linearisation of $h$;
- no proper legal extension $h_s{}^\frown\langle e\rangle$ of $h_s$ is a synchronisation linearisation of $h$.

For example, the following history of a synchronous channel

$$h \quad = \quad \langle \mathsf{call.send}^1(4), \mathsf{call.send}^2(5), \mathsf{call.receive}^3(()) \rangle$$

has two maximal synchronisation linearisations

$$\begin{aligned} h_s^1 &= \langle \mathsf{sync}^{1,3}(4,()):((),4) \rangle, \\ h_s^2 &= \langle \mathsf{sync}^{2,3}(5,()):((),5) \rangle, \end{aligned}$$

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

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

**Definition 8.** Consider a history $h$ of the synchronisation object.

- Given a maximal synchronisation linearisation $h_s$, we say that a return event $e$ is *anticipated following* $h_s$ if $e$ does not appear in $h$, but the corresponding sync event appears in $h_s$.
- We say that *some return event is anticipated* if for every maximal synchronisation linearisation $h_s$, a return event is anticipated following $h_s$.

For example, considering the above histories $h$ and $h_s^1$, the events $\mathsf{return.send}^1{:}()$ and $\mathsf{return.receive}^3{:}4$ are anticipated following $h_s^1$: assuming $h_s^1$ describes the synchronisations that happen, we would expect those return events to eventually happen; if they do not, that is a failure of liveness. Likewise, events $\mathsf{return.send}^2{:}()$ and $\mathsf{return.receive}^3{:}5$ are anticipated following $h_s^2$. Hence some return event is anticipated after $h$ (regardless of the synchronisation linearisation).

As another example, after the history

$$h' \quad = \quad \langle \mathsf{call.send}^1(4), \mathsf{call.receive}^2(()), \mathsf{call.receive}^3(()), \mathsf{return.send}^1{:}() \rangle,$$

some return event is anticipated, either $\mathsf{return.receive}^2{:}4$ or $\mathsf{return.receive}^3{:}4$, depending on which synchronisation happened.

We consider the semantics of the system to be described by a state machine. Some transitions in the state machine correspond to call and return events; other transitions are internal, corresponding to steps taken by threads inside an operation. A history corresponds to a path through the state machine, in the normal way.

We adopt some terminology from CSP [17]. We say that a state $s$ is *divergent* if it is possible to perform an infinite sequence of consecutive internal transitions from $s$. We say that a state is *stable* if there is no internal transition from it.

We say that the system *refuses* a particular return event $r$ in state $s$ if either $s$ is divergent, or $s$ is stable and has no transition corresponding to $r$. If the system does not refuse $r$ in state $s$ or any subsequent state, then, under reasonable assumptions about the scheduler, $r$ will eventually happen.

**Definition 9.** We say that a system *can fail to progress* after history $h$ if some return event is anticipated after $h$, but $h$ can lead to a state where every return event is refused.

We say that a system is *synchronisation-progressible* (with respect to the synchronisation specification object) if, for every history $h$, it cannot fail to progress after $h$.

For the earlier history $h$, synchronisation progressibility requires that not all return events are refused, so necessarily one of $\mathsf{return.send}^1{:}()$, $\mathsf{return.receive}^3{:}4$, $\mathsf{return.send}^2{:}()$, or $\mathsf{return.receive}^3{:}5$ is not refused. One of the synchronisations should happen, and then one of the relevant operations should be able to return. Subsequently, synchronisation progressibility requires that the other operation involved in the synchronisation is also able to return.

For the earlier history $h'$, synchronisation progressibility requires that one of the anticipated return events is not refused (although the other such event will be refused).

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

Synchronisation progressibility is defined in terms of a state machine. However, for a given synchronisation object, the state machine may depend upon the way scheduling is performed; hence, whether or not the synchronisation object is synchronisation-progressible might depend on assumptions about the scheduling. Possible assumptions are as follows.

1. The minimal possible assumption is that the scheduler schedules *some* thread whenever there is a thread that can take a step. Clearly, if no thread is scheduled, none will return. Any reasonable scheduler satisfies this property.
2. Several concurrency primitives allow a thread to suspend and wait for a signal. However, some (e.g. monitors in the JVM) allow waiting threads to wake up and resume without a corresponding signal, so called *spurious wakeups*. Code that uses such primitives is expected to guard against spurious wakeups, typically by checking an appropriate condition, and waiting again if it doesn't hold. When analysing such code for progressibility, we need to assume that spurious wakeups do not happen so often that no other thread is scheduled. In other words, we assume there is no infinite execution where one or more threads repeatedly perform a spurious wakeup, check the condition, and wait again, without any other threads being scheduled. Spurious wakeups happen rarely in practice, so this is a very reasonable assumption.
3. Some synchronisation objects may require an assumption that scheduling is *fair* between threads. For example, consider an object that uses a spin-lock [8, Chapter 7], where a thread that is trying to acquire the lock spins, repeatedly checking some condition, until no other thread holds the lock.

Consider a situation where thread $A$ is holding the lock, and thread $B$ is spinning, waiting to acquire the lock. In this situation, we will need to assume that thread $A$ is scheduled sufficiently often, so that it eventually releases the lock. In other words, we assume there is no infinite execution where only thread $B$ is scheduled. Most modern schedulers are fair in this sense.

4. Some synchronisation objects may require synchronisation primitives, such as locks and semaphores, to be fair. Such an assumption might require that there is no execution where one thread repeatedly fails to obtain a lock, while other threads obtain the lock infinitely often.

In the examples we consider in Section 8, the implementations require assumptions 1 and 2 to satisfy synchronisation progressibility.

Our liveness condition is somewhat different from the condition of *lock freedom* for concurrent datatypes [8]. That condition requires that whenever there are pending operation executions, eventually some such execution returns. However, this condition treats a thread that is blocked, trying to obtain a lock (whether a spin lock or not), as performing infinitely many steps. Hence any object that uses a lock in a non-trivial way is not lock-free.

By contrast, if a thread suspends, waiting to acquire a non-spin lock, we do not consider it to be taking steps. In most cases, assumption 1 will mean that the thread holding the lock will be scheduled sufficiently many times that it eventually releases the lock. Thus such an object can satisfy synchronisation progressibility. A similar argument holds for a spin lock, but requires assumption 3.

In fact, any synchronisation object without unary synchronisations is necessarily not lock-free: if a particular operation is unable to synchronise, then it will never return.

We now show that synchronisation progressibility satisfies locality. Consider a collection $\mathcal{O}$ of objects, and associated synchronisation specification object *Spec*, as in Section 2.5.

**Lemma 10.** Suppose that in $\mathcal{O}$, some return event is anticipated after history $h$. Then in some $o \in \mathcal{O}$, some return event is anticipated after history $h|o$.

PROOF. We argue by contradiction. Suppose that for each $o \in \mathcal{O}$, it is not the case that some return event is anticipated after $h|o$. Then, for each $o$, there is some maximum synchronisation $h^o_s$ of $h|o$ such that no return event of $o$ is anticipated following $h^o_s$. We interleave the $h^o_s$, for $o \in \mathcal{O}$, to build a history $h_s$ of *Spec*, as in Lemma 5. Then $h_s$ is a maximum synchronisation of $h$. But no return event is anticipated following $h_s$ (since none is anticipated for any $o \in \mathcal{O}$). This gives a contradiction.

**Proposition 11.** A system $\mathcal{O}$ of synchronisation objects is synchronisation-progressible if and only if each object $o \in \mathcal{O}$ is synchronisation-progressible.

PROOF. The left-to-right direction is trivial.

For the right-to-left direction, suppose each $o \in \mathcal{O}$ is synchronisation-progressible. Suppose, for a contradiction, $\mathcal{O}$ can fail to progress after some history $h$. Then some return event is anticipated after $h$, but $h$ can lead to a state $s$ where every return event is refused. But then Lemma 10 implies that for some $o \in \mathcal{O}$, some return event is anticipated after history $h|o$. However, every return event of $o$ is refused in its local state corresponding to $s$. This gives a contradiction.

## 3. Comparing synchronisation linearisation and standard linearisation

In this section we describe the relationship between synchronisation linearisation and standard linearisation.

It is clear that synchronisation linearisation is equivalent to standard linearisation in the (rather trivial) case that no operations actually synchronise, so each operation of the synchronisation specification object corresponds to a single operation of the concurrent datatype. Put another way: standard linearisation is an instance of synchronisation linearisation with just unary synchronisations.

However, linearisation and synchronisation linearisation are not equivalent in general. For example, consider the synchronous channel from Section 2, where synchronisation linearisation is captured by SyncChanSpec. We show that this property does not correspond to standard linearisable.

**Proposition 12.** The property of an implementation of a synchronous channel being synchronisation-linearisable with respect to SyncChanSpec cannot be captured as an instance of standard linearisability.

Proof. Suppose, for a contradiction, otherwise, so there is a linearisation specification object Spec such that any implementation of a synchronous channel is synchronisation-linearisable with respect to SyncChanSpec if and only if it is linearisable with respect to Spec. That would imply that for every concurrent history $h$, $h$ is synchronisation-linearisable with respect to SyncChanSpec if and only if $h$ is linearisable with respect to Spec. Consider the history

$$h \quad = \quad \langle \mathsf{call}.\mathsf{send}^1(3),\ \mathsf{call}.\mathsf{receive}^2(),\ \mathsf{return}.\mathsf{send}^1{:}(),\ \mathsf{return}.\mathsf{receive}^2(){:}3 \rangle.$$

This is synchronisation-linearisable with respect to SyncChanSpec. By the assumption, it must also be linearisable with respect to Spec; so there must be a legal history $h_s$ of Spec such that $h_s$ is a linearisation of $h$. Without loss of generality, suppose the send in $h_s$ occurs before the receive, i.e.

$$h_s \quad = \quad \langle \mathsf{send}^1(3){:}(),\ \mathsf{receive}^2(){:}3 \rangle.$$

But the history

$$h' \quad = \quad \langle \mathsf{call}.\mathsf{send}^1(3),\ \mathsf{return}.\mathsf{send}^1{:}(),\ \mathsf{call}.\mathsf{receive}^2(),\ \mathsf{return}.\mathsf{receive}^2(){:}3 \rangle$$

is also linearised by $h_s$, so $h'$ is linearisable with respect to Spec. But then the assumption would imply that $h'$ is synchronisation linearisable with respect to SyncChanSpec. This is clearly false, because the operations do not overlap. Hence no such linearisation specification Spec exists.

*3.1. Two-step linearisation*

We now show that binary heterogeneous synchronisation linearisation corresponds to a small adaptation of linearisation, where one of the operations on the synchronisation object corresponds to *two* operations of the linearisation specification object. We define below what we mean by this. We prove the correspondence in the next subsection. We consider some variations, including synchronisations of more than two threads, and the homogeneous case in Section 3.3. We prove the general case in Section 3.4.

Given a synchronisation object with signature

```
class SyncObj{
  def op1(x1: A1): B1
  def op2(x2: A2): B2
}
```

we will consider a linearisation specification object with signature

```
class TwoStepLinSpec{
  def op1(x1: A1): Unit
  def op̄1(): B1
  def op2(x2: A2): B2
}
```

The idea is that the operation $\mathsf{op}_1$ on SyncObj will be linearised by the composition of the two operations $\mathsf{op}_1$ and $\overline{\mathsf{op}}_1$ of TwoStepLinSpec; but operation $\mathsf{op}_2$ on SyncObj will be linearised by just the operation $\mathsf{op}_2$ of TwoStepLinSpec, as before. We call such an object a *two-step linearisation specification object*.

We describe how to define a two-step linearisation specification object corresponding to a given synchronisation specification object in the following subsections. First, we define our notion of two-step linearisation with respect to such a two-step linearisation specification object. In the definitions below, we describe just the differences from standard linearisation, to avoid repetition.

We define a history $h_s$ of such a two-step specification object much as in Section 2.2, with the addition that for each event $\overline{\mathsf{op}}_1^{i_1}():y$ in $h_s$, we require that there is an earlier event $\mathsf{op}_1^{i_1}(x):()$ in $h_s$ with the same execution identity; other than in this regard, execution identities are not repeated in $h_s$.

Let $h$ be a complete concurrent history of a synchronisation object, and let $h_s$ be a legal history of a two-step specification object. We say that $h_s$ *corresponds* to $h$ if:

- For every $\mathsf{call}.\mathsf{op}_1^{i_1}(x)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}:y$ in $h$, $h_s$ contains $\mathsf{op}_1^{i_1}(x):()$ and $\overline{\mathsf{op}}_1^{i_1}():y$; and vice versa;
- For every $\mathsf{call}.\mathsf{op}_2^{i_2}(x)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}:y$ in $h$, $h_s$ contains $\mathsf{op}_2^{i_2}(x):y$; and vice versa.

We say that $h_s$ is a *two-step linearisation* of $h$ if there is some way of interleaving the two histories such that

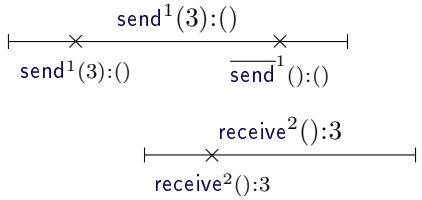
- Each $\mathsf{op}_1^{i_1}(x):()$ and $\overline{\mathsf{op}}_1^{i_1}():y$ occur between $\mathsf{call}.\mathsf{op}_1^{i_1}(x)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}:y$, in that order;

- Each $\mathsf{op}_2^{i_2}(x)\!:\!y$ occurs between $\mathsf{call}.\mathsf{op}_2^{i_2}(x)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}\!:\!y$.

For example, consider a synchronous channel, with send corresponding to $\mathsf{op}_1$, and receive corresponding to $\mathsf{op}_2$. Then the following would be an interleaving of histories of the channel and a two-step linearisation specification object.

$$\langle \mathsf{call}.\mathsf{send}^1(3),\ \mathsf{send}^1(3)\!:\!(),\ \mathsf{call}.\mathsf{receive}^2(),\ \mathsf{receive}^2()\!:\!3,\\ \overline{\mathsf{send}}^1()\!:\!(),\ \mathsf{return}.\mathsf{send}^1\!:\!(),\ \mathsf{return}.\mathsf{receive}^2\!:\!3\rangle.$$

This is represented by the following timeline, where the horizontal lines and the labels above represent the execution of operations on the channel, and the "×"s and the labels below represent the corresponding operations of the specification object.



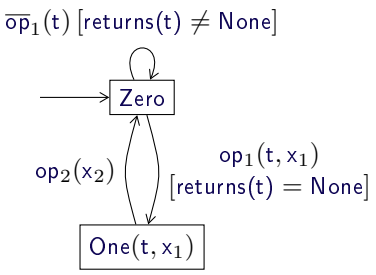

The definition of two-step linearisation of a synchronistion object then follows from this definition of two-step linearisation of complete histories, analogously to as in Definition 3.

*3.2. Proving the relationship*

We now prove the relationship between synchronisation linearisation and two-step linearisation. Consider a synchonisation specification object SyncSpec. We build a corresponding two-step linearisation specification object TwoStepLinSpec such that synchronisation linearisation with respect to SyncSpec is equivalent to two-step linearisation with respect to TwoStepLinSpec. The definition we choose is not the simplest possible (and the two-step linearisation specification object is certainly more complex than the synchronisation linearisation specification object) but it is convenient for the testing framework we use in Section 5.

The definition of TwoStepLinSpec is in Figure 3. We assume that each thread has an identity in some range $[0\,..\,\mathsf{NumThreads})$. For simplicity, we arrange for this identity to be included in the call events for operations $\mathsf{op}_1$ and $\overline{\mathsf{op}}_1$.

The object TwoStepLinSpec encodes the automaton on the right. It requires that corresponding executions of $\mathsf{op}_1$ and $\mathsf{op}_2$ occur consecutively, which means that corresponding executions of $\mathsf{op}_1$ and $\mathsf{op}_2$ on the synchronisation object are linearised successively. However, it allows the corresponding $\overline{\mathsf{op}}_1$ to be later (but before the next operation execution by the same



```
type ThreadID = Int // Thread identifiers.
val NumThreads: Int = ... // Number of threads.
trait State
case object Zero extends State
case class One(t: ThreadID, x1: A1) extends State

object TwoStepLinSpec{
  private var state: State = Zero
  private val returns = new Array[Option[B1]](NumThreads)
  for(t <− 0 until NumThreads) returns(t) = None
  def op1(t: ThreadID, x1: A1): Unit = {
    require(state == Zero && returns(t) == None); state = One(t, x1); ()
  }
  def op2(x2: A2): B2 = {
    require(state.isInstanceOf[One]); val One(t, x1) = state
    val (y1, y2) = SyncSpec.sync(x1, x2); returns(t) = Some(y1); state = Zero; y2
  }
  def op1bar(t: ThreadID): B1 = {
    require(state == Zero && returns(t).isInstanceOf[Some[B1]])
    val Some(y1) = returns(t); returns(t) = None; y1
  }
}
```

Figure 3: The TwoStepLinSpec object.

thread).

TwoStepLinSpec uses an array returns, indexed by thread identities, to record the value that should be returned by an $\overline{\mathsf{op}}_1$ execution by each thread. Each execution of $\mathsf{op}_2$ calls SyncSpec.sync to obtain the values that should be returned for synchronisation linearisation; it writes the value for the corresponding $\overline{\mathsf{op}}_1$ into returns.

Formally, the object TwoStepLinSpec defines a set of legal histories, using events as in the previous subsection. In particular, in each legal history, the precondition of each operation is satisfied, so the history indeed corresponds to the above automaton. The following lemma identifies important properties of those histories. It follows immediately from the definition.

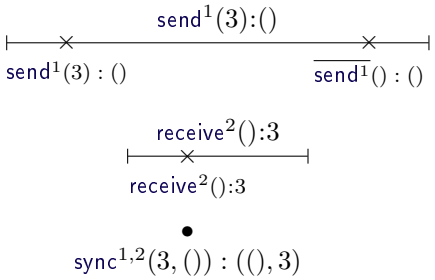
**Lemma 13.** Within any legal history of TwoStepLinSpec, events $\mathsf{op}_1$ and $\mathsf{op}_2$ alternate. Let $\mathsf{op}_1^{i_1}(t, x_1){:}()$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ be a consecutive pair of such events. Then $\mathsf{op}_2$ makes a call $\mathsf{SyncSpec.sync}(x_1, x_2)$ obtaining result $(y_1, y_2)$. The next event for thread $t$ (if any) will be $\overline{\mathsf{op}}_1^{i_1}(t){:}y_1$; and this will be later in the history than $\mathsf{op}_2^{i_2}(x_2){:}y_2$. Further, the corresponding history of events $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ is a legal history of SyncSpec.

Conversely, each history with events ordered in this way will be a legal history of TwoStepLinSpec if the corresponding history of events $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ is a legal history of SyncSpec.

The following proposition reduces synchronisation linearisation to two-step linearisation.

**Proposition 14.** Let SyncObj be a binary heterogeneous synchronisation object, SyncSpec a corresponding synchronisation specification object, and let TwoStepLinSpec be built from SyncSpec as above. Then each history $h$ of SyncObj is two-step linearisable with respect to TwoStepLinSpec if and only if it is synchronisation linearisable with respect to SyncSpec. Hence SyncObj is two-step linearisable with respect to TwoStepLinSpec if and only if it is synchronisation linearisable with respect to SyncSpec.

PROOF. **($\Rightarrow$).** Let $h$ be a concurrent history of SyncObj that is two-step linearisable with respect to TwoStepLinSpec. By assumption, there is an extension $h'$ of $h$, and a legal history $h_s$ of TwoStepLinSpec such that $h_s$ is a two-step linearisation of $h'' = \mathit{complete}(h')$. Build a history $h'_s$ of SyncSpec by replacing each consecutive pair $\mathsf{op}_1^{i_1}(x_1){:}(),\ \mathsf{op}_2^{i_2}(x_2){:}y_2$ in $h_s$ by the event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$, where $y_1$ is the value returned by the corresponding $\overline{\mathsf{op}}_1^{i_1}()$. This is illustrated by the example timeline below, where $h''$ is represented by the horizontal lines and the labels above; $h_s$ is represented by the "$\times$"s and the labels below; and $h'_s$ is represented by the "$\bullet$" and the label below.




The history $h'_s$ is legal for SyncSpec by Lemma 13. It is possible to interleave $h''$ and $h'_s$ by placing each event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ in the same place as the corresponding event $\mathsf{op}_2^{i_2}(x_2){:}y_2$ in the interleaving of $h''$ and $h_s$; by construction, this is between $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$, and between $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$. Hence $h'_s$ is a synchronisation linearisation of $h''$; and so $h$ is synchronisation-linearisable.

**($\Leftarrow$).** Let $h$ be a history of SyncObj that is synchronisation-linearisable with respect to SyncSpec. By assumption, there is an extension $h'$ of $h$, and a legal history $h_s$ of SyncSpec such that $h_s$ is a synchronisation linearisation of $h'' = \mathit{complete}(h')$. Build a history $h'_s$ of TwoStepLinSpec by replacing each event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ in $h_s$ by the three consecutive events $\mathsf{op}_1^{i_1}(x_1){:}(),\ \mathsf{op}_2^{i_2}(x_2){:}y_2,\ \overline{\mathsf{op}}_1^{i_1}(){:}y_1$.

The history $h'_s$ is legal for TwoStepLinSpec by Lemma 13. It is possible to interleave $h''$ and $h'_s$ by placing each triple $\mathsf{op}_1^{i_1}(x_1){:}(),\ \mathsf{op}_2^{i_2}(x_2){:}y_2,\ \overline{\mathsf{op}}_1^{i_1}(){:}y_1$ in the same place as the corresponding event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ in the interleaving of $h''$ and $h_s$; by construction, each $\mathsf{op}_1^{i_1}(x_1){:}()$ and $\overline{\mathsf{op}}_1^{i_1}(){:}y_1$ are between

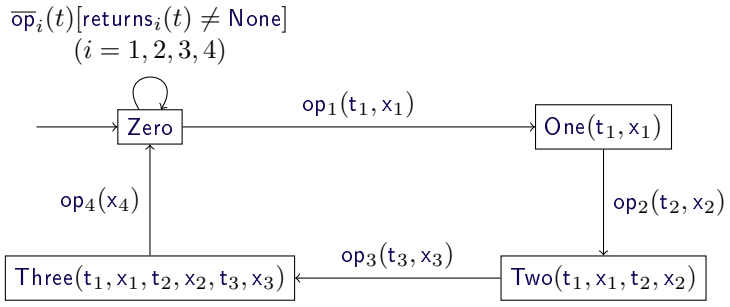
call.$\mathsf{op}_1^{i_1}(x_1)$ and return.$\mathsf{op}_1^{i_1}$:$y_1$; and each $\mathsf{op}_2^{i_2}(x_2)$:$y_2$ is between call.$\mathsf{op}_2^{i_2}(x_2)$ and return.$\mathsf{op}_2^{i_2}$:$y_2$. Hence $h'_s$ is a two-step linearisation of $h''$; and so $h$ is two-step-linearisable.

### *3.3. Variations*

We now discuss some variations on the case of binary heterogeneous synchronisations in the previous section. We give a general construction in the next section, but we consider the constructions in this section to be easier to understand: to a certain extent, this section can be seen as providing stepping stones towards that general construction.

The results of the previous subsections carry across to non-binary, fixed arity synchronisations, in a straightforward way. For a $k$-way synchronisation between distinct operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$, the corresponding two-step linearisation specification object has $2k-1$ operations, $\mathsf{op}_1, \ldots, \mathsf{op}_k, \overline{\mathsf{op}}_1, \ldots, \overline{\mathsf{op}}_{k-1}$. The definition of two-step linearisation is then the obvious adaptation of the binary case: each operation $\mathsf{op}_i$ of the synchronisation object is linearised by the composition of $\mathsf{op}_i$ and $\overline{\mathsf{op}}_i$ of the specification object, for $i = 1, \ldots, k-1$.

The construction of the previous subsection is easily adapted to the case of $k$-way synchronisations for $k > 2$. The specification object encodes an automaton with $k$ states. The figure below gives the automaton in the case $k = 4$.

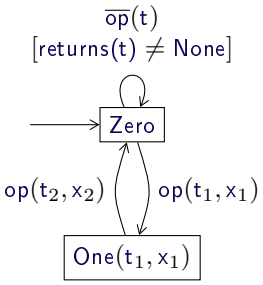



The final op operation, $\mathsf{op}_4$ in the above figure, applies the sync operation of the synchronisation specification object to the parameters $\mathsf{x}_1, \ldots, \mathsf{x}_k$ to obtain the results $\mathsf{y}_1, \ldots, \mathsf{y}_k$; it stores the first $k-1$ in appropriate $\mathsf{returns}_i$ arrays, and returns $\mathsf{y}_k$ itself. In the case $k = 4$, it has definition:

```
def op4(x4: A4): B4 = {
  require(state.isInstanceOf[Three]); val Three(t1, x1, t2, x2, t3, x3) = state
  val (y1, y2, y3, y4) = SyncSpec.sync(x1, x2, x3, x4)
  returns1(t1) = Some(y1); returns2(t2) = Some(y2); returns3(t3) = Some(y3)
  state = Zero; y4
}
```

Each $\overline{\mathsf{op}}_i$ operation retrieves the result from the corresponding $\mathsf{returns}_i$ array.

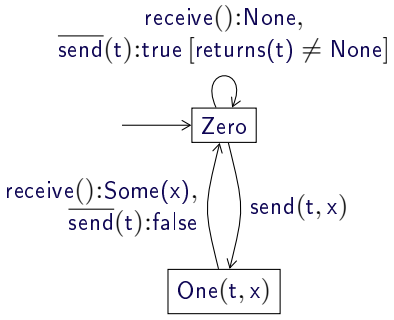
We now consider the homogeneous case. For simplicity, we describe the binary case; synchronisations of more than two operation executions are handled similarly. Suppose we have a synchronisation object with a single operation **def** op(x: A): B. All executions of op have to be treated similarly, so we associate *each* with two operations op and $\overline{\text{op}}$ of the specification object. The specification object is below, and encodes the automaton on the right. The second execution of op in any synchronisation (from the One state of the automaton) writes the results of the execution into the returns array. Each execution of $\overline{\text{op}}$ returns the stored value.




```
class TwoStepHomoLinSpec{
  private var state: State = Zero
  private val returns = new Array[Option[B1]](NumThreads)
  for(t <− 0 until NumThreads) returns(t) = None
  def op(t: ThreadID, x: A): Unit = {
    require(returns(t) == None)
    state match{
      case Zero => state = One(t, x)
      case One(t1, x1) =>
        val (y1, y2) = SyncSpec.sync(x1, x)
        returns(t1) = Some(y1); returns(t) = Some(y2); state = Zero
    }
  }
  def op‾(t: ThreadID): B = {
    require(state.isInstanceOf[Zero] && returns(t).isInstanceOf[Some])
    val Some(y) = returns(t); returns(t) = None; y
  }
}
```

Recall that some operations have multiple modes of synchronisations: different executions of the operation may have synchronisations with different arities. For example, in a timeout channel, an execution of the send and receive operations may synchronise with an execution of the other operation, or may timeout corresponding to a unary synchronisation.

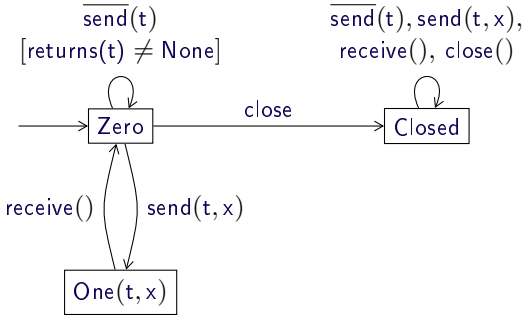
The figure to the right gives the automaton for a timeout channel, where we treat send as corresponding to $\text{op}_1$ (we omit concrete code in the interests of brevity). The automaton is similar to that for a standard channel. The receive operation can happen from either state: if it happens from the One state, then a synchronisation has occurred and the execution returns a value of the form Some(x); but if it happens from the Zero state, there



has been no corresponding send, and so the execution returns None, indicating a timeout. Likewise, the $\overline{\text{send}}$ operation can happen from either state; if it happens from the Zero state, then a synchronisation has occurred and the execution returns true; but if it happens from the One state, there has been no corresponding receive, and so the execution returns false, indicating a timeout.

We now consider stateful specification objects. In general, we can simply augment the Zero and One states of the automaton to include the state of the specification object. Different transitions may be available based upon that state. However, it can be clearer and simpler to introduce different named states into the automaton.

The figure below gives the specification automaton for a closeable channel.




The automaton has an additional state, Closed, corresponding to the channel being closed. All executions are unary synchronisations in this state. This automaton represents a simplification over the general approach discussed above: we do not need two separate states after closing. Note that a $\overline{\text{send}}$(t) transition from the Closed state may succeed if the corresponding synchronisation happened before the channel was closed, in which case returns(t) ≠ None.

### *3.4. The general construction*

We now give a general construction to show how two-step linearisation can capture synchronisation linearisation. Fix a synchronisation object SyncObj, a synchronisation specification object SyncSpec, and synchronisation abstraction function $\mathcal{A}$. Let the operations of the SyncObj be

**def** $\text{op}_i$(x: $\text{A}_i$): $\text{B}_i$ for $i = 1, \ldots, n$.

The two-step linearisation tester has operations $\text{op}_i$ and $\overline{\text{op}}_i$ for $i = 1, \ldots, n$. The idea is that for a mapping

$$\langle \text{op}_{i_1}(x_1), \ldots, \text{op}_{i_a}(x_a) \rangle \mapsto \text{sync}(x_1, \ldots, x_a) \in \mathcal{A},$$

corresponding to a synchronisation of arity $a$, is linearised in the following order:

1. $\text{op}_{i_1}, \ldots, \text{op}_{i_a}$, in order;
2. $\overline{\text{op}}_{i_i}$, for some $i$, which calls SyncSpec.sync$(x_1, \ldots, x_a)$, and stores the results for other threads;

```
trait Op
case class Op_i(t: ThreadID, x: A_i) // for i = 1, ..., n.
type State = List[Op]
type B = ⋃_{i=1,...,n} B_i

class TwoStepLinSpec{
  private var state: State = List()
  private val returns = new Array[Option[B]](NumThreads)
  for(i <− 0 until NumThreads) returns(i) = None

  def op_i(t: ThreadID, x: A_i): Unit = { // for i = 1, ..., n.
    require(∃ops ∈ dom 𝒜 • opsFor(state)⌢⟨op_i(x)⟩ ≤ ops)
    state = state :+ Op_i(t,x)
  }

  def ōp_i(t: ThreadID): B_i = { // for i = 1, ..., n.
    if(opsFor(state) ∈ dom 𝒜){
      val (y_1,...,y_a) = SyncSpec.𝒜(opsFor(state))
      for(m <− 1 to a) returns(state(m).t) = Some(y_m)
      state = List()
    }
    else (require(state == List() && returns(t).isInstanceOf[Some])
    val Some(y) = returns(t); returns(t) = None; y
  }
}
```

Figure 4: A schema for the general construction of a two-step linearisability tester.

3. The remaining $\overline{\mathsf{op}}_{i_j}$, in some order, which retrieve their results.

No other operation occurs between the start of stage 1 until after stage 2; but other operations may occur during stage 3.

During stage 1, the two-step linearisabilty tester records the prefix of $\langle \mathsf{op}_{i_1}(x_1), \ldots, \mathsf{op}_{i_a}(x_a)\rangle$ performed so far. This is done by defining a case class $\mathsf{Op}_i$ corresponding to each such operation $\mathsf{op}_i$, as at the top of Figure 4. The following function maps such a sequence to the corresponding sequence of operations, for use with $\mathcal{A}$.

$$opsFor(state) \quad = \quad \langle \mathsf{op}_j(x_j) \mid \mathsf{Op}_j(t_j, x_j) \leftarrow state\rangle.$$

A schema (using pseudo-Scala) for the general construction is in Figure 4. The variable state records the sequence of $\mathsf{op}_i$ linearised so far: this satisfies an invariant that it is a prefix of an element of $\mathrm{dom}\,\mathcal{A}$. For each thread t, returns(t) stores the value to be returned by t as part of its most recent uncompleted synchronisation, if any.

The operation $\mathsf{op}_i$ appends an appropriate $\mathsf{Op}_i$ object to state. The precondition ensures that state still corresponds to a prefix of an element of $\mathrm{dom}\,\mathcal{A}$.

The operation $\overline{\mathsf{op}}_i$ proceeds as follows. In the case that state corresponds to an element of $\mathrm{dom}\,\mathcal{A}$, it calls the corresponding operation on SyncSpec, stores each result in the corresponding entry of returns, and resets state. Otherwise, it requires that some other operation has called the operation on SyncSpec, and that state holds its initial value. In either case, the operation retrieves and clears its result from the appropriate entry in returns.

It is then straightforward to check that operations of TwoStepLinSpec can be linearised only as described earlier. The proof that two-step linearisability corresponds to synchronisation linearisability is then much as in the proof for the binary case in Proposition 14.

Figure 4 uses preudo-Scala. For a concrete implementation, we need a separate implementation of $\mathsf{op}_i$ and $\overline{\mathsf{op}}_i$ for each $i$. Then each calculation involving $\mathcal{A}$ can be implemented directly, for example using a case analysis.

## 4. Linearisability testing

In the following two sections, we describe techniques for testing whether the implementation of a synchronisation object is synchronisation linearisable with respect to a synchronisation specification object. The techniques are influenced by the techniques for testing (standard) linearisation [14, 15], so we begin by sketching those techniques.

The idea of linearisability testing is as follows. We run several *worker threads*, performing operations (typically chosen randomly) upon the concurrent datatype that we are testing, and logging the calls and returns. More precisely, a thread that performs a particular operation $\mathsf{op}^i(x)$: (1) writes $\mathsf{call}.\mathsf{op}^i(x)$ into the log; (2) performs $\mathsf{op}(x)$ on the synchonisation object, obtaining result $y$, say; (3) writes $\mathsf{return}.\mathsf{op}^i{:}y$ into the log. Further, the logging associates each operation execution with an execution $\mathsf{op}(x)$ of the corresponding operation on the specification object.

Once all worker threads have finished, we can use an algorithm to test whether the history is linearisable with respect to the specification object. The algorithm searches for an order to linearise the executions, consistent with the log, and such that the order represents a legal history of the corresponding executions on the specification object. See [14, 15] for details of several algorithms. All these algorithms assume that the specification object is deterministic.

This process can be repeated many times, so as to generate and analyse many histories. Our experience is that the technique works well. It seems effective at finding bugs, where they exist, typically within a few seconds; for example, we used it to find an error in the concurrent priority queue of [18], which we believe had not previously been documented. Further, the technique is easy to use: we have taught it to undergraduate students, who have used it effectively.

This testing concentrates upon the safety property of linearisation, rather than liveness properties such as deadlock-freedom. However, if the concurrent object can deadlock, it is likely that the testing will discover this. Related to this point, it is the responsibility of the tester to define the worker threads in

a way that all executions will eventually return, so the threads terminate. For example, consider a partial stack where a pop operation blocks while the stack is empty; here, the tester would need to ensure that threads collectively perform at least as many pushes as pops, to ensure that each pop does eventually return.

Note also that there is potentially a delay between a worker thread writing the call event into the log and actually calling the operation; and likewise there is potentially a delay between the operation returning and the thread writing the return event into the log. However, these delays do not generate false errors: if a history without such delays is linearisable, then so is a corresponding history with delays. We believe that it is essential that the technique does not give false errors: an error reported by testing should represent a real error; testing of a correct implementation should be able to run unsupervised, maybe for a long time. Further, our experience is that the delays do not prevent the detection of bugs when they exist (although might require performing the test more times). This means that a failure to find any bugs, after a large number of tests, can give us good confidence in the correctness of the concurrent datatype.

## 5. Adapting the linearisability framework

In this section we investigate how to use the existing linearisation testing framework for testing synchronisation linearisation, using the ideas of Section 3. We require the synchronisation specification object to be deterministic, reflecting the fact that the linearisation testing framework requires its specification object to be deterministic.

This is not a use for which the framework was intended, so we consider it a hack. However, it has the advantage of not requiring the implementation of any new algorithms. (We do not consider progressibility in this section.)

We adapt the idea of two-step linearisation from Section 3. We start by considering the case of binary heterogeneous synchronisation. We aim to obtain a log history that can be tested for (standard) linearisation against TwoStepLinSpec.

As with standard linearisability testing, we run several worker threads, calling operations on the synchronisation object, and logging the calls and returns.

- A thread $t_1$ that performs the concrete operation $\mathsf{op}_1(x_1)$: (1) writes $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ into the log, associating it with a corresponding execution $\mathsf{op}_1(t_1, x_1)$ on the specification object; (2) performs $\mathsf{op}_1(x_1)$ on the synchonisation object, obtaining result $y_1$, say; (3) writes $\mathsf{return}.\mathsf{op}_1^{i_1}:()$ into the log; (4) writes $\mathsf{call}.\overline{\mathsf{op}}_1^{i_1}()$ into the log, associating it with a corresponding execution $\overline{\mathsf{op}}_1(t)$ on the specification object; (5) writes $\mathsf{return}.\overline{\mathsf{op}}_1^{i_1}:y_1$ into the log.

- A thread $t_2$ that performs operation $\mathsf{op}_2$, acts as for standard linearisability testing. It: (1) writes $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ into the log, associating it with a corresponding execution $\mathsf{op}_2(x_2)$ on the specification object; (2) performs $\mathsf{op}_2(x_2)$ on the synchonisation object, obtaining result $y_2$, say; (3) writes $\mathsf{return}.\mathsf{op}_2^{i_2}:y_2$ into the log

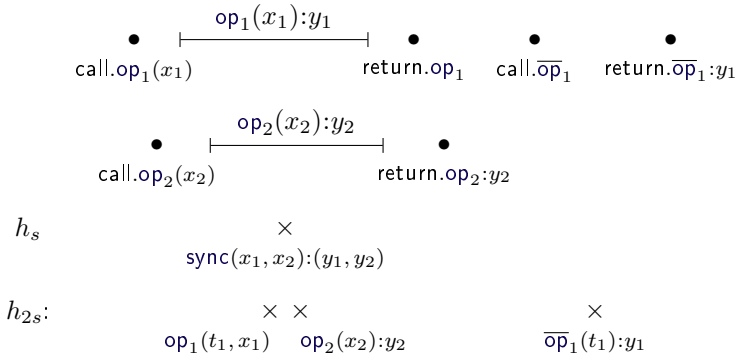



Figure 5: Illustration of two-step linearisation testing. The operation executions are represented by the horizontal lines with labels above (denoted "$h$" in Proposition 16). The log entries are represented by the bullets with labels below (denoted "$h_l$" in Proposition 16). Linearisation points are represented by crosses with labels below: the penultimate row, labelled "$h_s$", is a synchronisation linearisation; the bottom row, labelled "$h_{2s}$", is a linearisation of the two-step synchronisation object. Execution identifiers and null arguments and returns are omitted, for clarity.

The top half of Figure 5 illustrates a possible run, containing a single synchronisation, together with the log history.

Once all threads have finished, we test whether the log history is linearisable (i.e. standard linearisation) with respect to TwoStepLinSpec from Section 3. Figure 5 gives an example linearisation, denoted $h_{2s}$.

Note that we have three related concepts here: (1) synchronisation linearisation of the concrete history of operation executions with respect to SyncSpec; (2) two-step linearisation of the concrete history with respect to TwoStepLinSpec; and (3) linearisation of the log history with respect to TwoStepLinSpec. Proposition 14 shows that the first two of these are equivalent. We need to show that these imply (3), so the technique does not give false errors. (The converse might not hold, because of delays in writing to the log.)

Since the linearisation algorithm receives a log history, rather than a concrete history, we need to describe the relationship.

**Definition 15.** Let $h$ be a complete history of a binary heterogeneous synchronisation object, and let $h_l$ be a log history for the same object. We say that the two histories *correspond* if there is some way of interleaving them such that

- Each call.$\mathsf{op}_1^{i_1}(x_1)$, from $h_l$, precedes the call and return of $\mathsf{op}_1^{i_1}(x_1)$:$y_1$ from $h$, which precede return.$\mathsf{op}_1^{i_1}$:(), call.$\overline{\mathsf{op}}_1^{i_1}$() and return.$\overline{\mathsf{op}}_1^{i_1}$:$y_1$, from $h_l$, in that order.
- Each call.$\mathsf{op}_2^{i_2}(x_2)$, from $h_l$, precedes the call and return of $\mathsf{op}_2^{i_2}(x_2)$:$y_2$ from $h$, which precede return.$\mathsf{op}_2^{i_2}$:$y_2$, from $h_l$.

**Proposition 16.** Let $h$ be a complete history of a binary heterogeneous synchronisation object, and let $h_l$ be a corresponding log history for the same object. Let SyncSpec be a synchronisation specification object, and TwoStepSyncSpec the corresponding two-step synchronisation specification object, constructed as in Section 3.2. Suppose $h$ is synchronisation-linearisable with respect to SyncSpec. Then $h_l$ is linearisable with respect to TwoStepSyncSpec.

PROOF. Since $h$ is synchronisation-linearisable, there is a legal history $h_s$ of SyncSpec such that $h_s$ is a synchronisation linearisation of $h$. Consider the interleaving of $h_s$ and $h$, that demonstrates this, and interleave $h_l$ with it, consistent with the interleaving of $h$ and $h_l$ that demonstrates that they correspond. Figure 5 illustrates such an interleaving.

We build a history $h_{2s}$ of TwoStepSyncSpec, and interleave it with $h_l$ as follows. In the interleaving of the previous paragraph, replace each event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ (from $h_s$) by immediately consecutive events $\mathsf{op}_1^{i_1}(x_1){:}()$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$, and add $\overline{\mathsf{op}}_1^{i_1}(){:}y_1$ between $\mathsf{call}.\overline{\mathsf{op}}_1^{i_1}()$ and $\mathsf{return}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$ (from $h_l$). Again, Figure 5 illustrates such an interleaving. This is a legal history of TwoStepSyncSpec, by Lemma 13. Further, each event of $h_{2s}$ is between the corresponding call and return events of $h_l$, by construction. Hence $h_{2s}$ is a linearisation of $h_l$.

This approach generalises to non-binary synchronisations, homogeneous synchronisations, and stateful specification objects as in Section 3.3.

As with standard linearisation, the tester needs to define the worker threads so that all executions will eventually return, i.e. so that each will be able to synchronise. For a binary heterogeneous synchronisation with no precondition, we can achieve this by half the threads calling one operation, and the other half calling the other operation (with the same number of calls by each). For a binary homogeneous synchronisation, this approach might not work if every worker does more than one operation: one worker might end up with two operations to perform, when all others have terminated; instead, we arrange for an even number of workers to each perform a single operation.

*Variable-arity synchronisations.* It turns out that it is not, in general, possible to capture variable-arity synchronisations using this technique, in particular where the arity of a synchronisation depends upon the relative timing of executions, as opposed to the state of the specification object. This is a result of two things: that the logging of operations, in particular the $\overline{\mathsf{op}}_1$, can be arbitrarily delayed; and that it can be nondeterministic whether or not two executions synchronise, which is at odds with the fact that each operation on the specification object needs to be deterministic.

To illustrate this point, consider a timeout channel. Without loss of generality, let the send operation correspond to $\mathsf{op}_1$, and the receive operation correspond to $\mathsf{op}_2$.

The top-half of Figure 6 gives a timeline illustrating a successful send(3) and receive. This corresponds to the history

$$\langle \mathsf{send}(3){:}(),\ \mathsf{receive}(){:}\mathsf{Some}(3),\ \overline{\mathsf{send}}(){:}\mathsf{true}\rangle$$

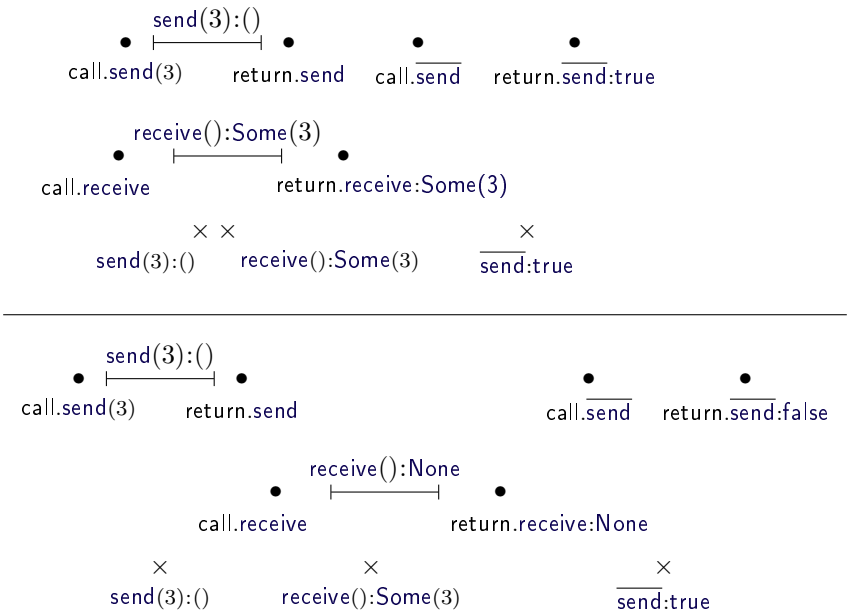



Figure 6: Figure showing why two-step linearisation cannot be used for a timeout channel. Conventions are as in Figure 5.

of the specification object.

The bottom-half side of Figure 6 gives a timeline illustrating an unsuccessful send(3) and receive, and where the logging of $\overline{\text{send}}$ is delayed. None of the executions overlap, so they must necessarily be linearised in the same order as in the previous history. The specification object is deterministic, so the operations must return the same results as in the previous history. But, in the cases of receive and $\overline{\text{send}}$, those returned values, Some(3) and true, do not agree with the corresponding values in the log history, None and false. Hence the history would be flagged as an error, despite being valid.

The difference between this situation and the discussion in Section 3.3 is that the logging of operations, in particular the $\overline{\text{send}}$, can be arbitrarily delayed. However, in the earlier section we allowed the $\overline{\text{send}}$ anywhere within the corresponding concrete operation. This means that a history like in the bottom-half of Figure 6 could be linearised by the history

$$\langle \text{send}(3){:}(),\ \overline{\text{send}}(3){:}\text{false},\ \text{receive}(){:}\text{None} \rangle$$

of the two-step specification object. Here the operations take place in a different order than for Figure 6, and so this is consistent with a deterministic specification object.

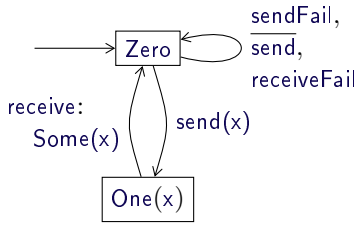
A similar problem arises with a timeout exchanger.

We have investigated an alternative approach, which involves worker threads

adapting their logging behaviour based on the outcome of their operations. For the timeout channel:

- A thread that sends a value $x$: (1) writes $\mathsf{call.send}^{i_1}(x)$ into the log; (2) performs $\mathsf{send}(x)$ on the channel; (3) writes $\mathsf{return.send}^{i_1}{:}()$ into the log; (4) if the send is successful, associates the log entries with an operation $\mathsf{send}(x)$ on the specification object, and otherwise associates them with an operation $\mathsf{sendFail}(x)$; (5) if the send is successful, writes $\mathsf{call.}\overline{\mathsf{send}}^{i_1}()$ and $\mathsf{return.}\overline{\mathsf{send}}_1^{i_1}{:}()$ into the log, associating them with an operation $\overline{\mathsf{send}}()$ on the specification object (and otherwise does nothing).
- A thread that performs a receive: (1) writes $\mathsf{call.receive}^{i_2}$ into the log; (2) performs $\mathsf{receive}$ on the channel, receiving result $r$, say; (3) writes $\mathsf{return.receive}^{i_1}{:}r$ into the log; (4) if the receive was successful, associates the log entries with an operation $\mathsf{receive}$ on the specification, and otherwise associates them with an operation $\mathsf{receiveFail}$.

The specification object then encodes the automaton to the right. Thus a successful synchronisation is linearised by the sequence $\mathsf{send(x)}$, $\mathsf{receive{:}Some(x)}$, $\overline{\mathsf{send}}$, with the former two events consecutive, as for other binary heterogeneous synchronisations. Unsuccessful sends and receives are each linearised by a single event.




A similar technique can be used for a timeout exchanger. We consider this approach convoluted, and we do not advocate it. We include it only for completeness.

## 6. Direct testing of synchronisation linearisation and progressibility

We now consider how to test for synchronisation linearisation more directly. We also consider how to test for synchronisation progressibility. We perform logging precisely as for standard linearisation: a thread that performs a particular operation $\mathsf{op}^i(x)$: (1) writes $\mathsf{call.op}^i(x)$ into the log; (2) performs $\mathsf{op}(x)$ on the synchonisation object, obtaining result $y$, say; (3) writes $\mathsf{return.op}^i{:}y$ into the log.

When testing for synchronisation linearisation, we again make it the responsibility of the tester to define the worker threads in a way that ensures that all operation executions will be able to synchronise, so all threads will eventually terminate.

When testing for progress, we remove the requirement on the tester to ensure that all operation executions can synchronise. Indeed, in some cases, in order to find failures of progress, it is necessary that not all executions can synchronise: we have examples of incorrect synchronisation objects where (for example) if there are two executions of $\mathsf{op}_1$ and one of $\mathsf{op}_2$, then it's possible that *neither* execution of $\mathsf{op}_1$ returns, signifying a failure of progressibility; but if there

were a second execution of $\mathtt{op}_2$, it would unblock both executions of $\mathtt{op}_1$, so all executions would return, and the failure of progressibility would be missed.

Instead, we run threads performing operations, typically chosen at random; and after a suitable duration, we interrupt any threads that have not yet returned. The duration before the interrupts needs to be chosen so that if any threads have not returned by that point, then (almost certainly) they really are stuck: otherwise this approach it likely to produce false positives. We discuss this point further in Section 8.

In the remainder of this section we consider algorithms for determining whether the resulting log history is synchronisation linearisable, and whether it is synchronisation progressible. These algorithms again assume that the synchronisation specification object is deterministic. In Section 6.1 we present a general algorithm for this problem, based on depth-first search. We also show how the algorithm can be extended to test for synchronisation progressibility.

We then consider the complexity of this problem. We show, in Section 6.2, that, for a *stateful* synchronisation object, the problem of deciding whether a history is synchronisation-linearisable is NP-complete in general: this is not surprising, given a similar result for standard linearisation.

Things are more interesting in the stateless case. (In practice, linearisation is only ever considered in the stateful case; for a stateless object, deciding linearisability amounts to checking each operation execution in isolation, which can be decided in linear time.) We show that in the case of *binary* synchronisations with a *stateless* specification object, synchronisation linearisability can be decided in polynomial time: we consider the heterogeneous case in Section 6.3, and the homogeneous case in Section 6.4. Nevertheless, in Section 6.5 we show that for synchronisations of three or more executions, the problem is again NP-complete, even in the stateless case.

The table below summarises the complexity results for deciding synchronisation linearisation for a history of length $n$.

| | |
|---|---|
| Stateful | NP-complete |
| Stateless, binary, heterogeneous | $O(n^2)$ |
| Stateless, binary, homogeneous | $O(n^4)$ |
| Stateless, non-binary | NP-complete |

### *6.1. The general case*

We describe an algorithm for deciding whether a given complete history $h$ is synchronisation linearisable with respect to a given synchronisation specification object. We transform the problem into a graph-search algorithm as follows.

We define a search graph, where each node is a *configuration* comprising:

- An index $i$ into the log;
- A set *pending* of operation executions that were called in the first $i$ events of the log and that have not yet been linearised;
- A set *linearised* of operation executions that were called in the first $i$ events of the log and that have been linearised, but have not yet returned;

- The state *spec* of the specification object after the synchronisations linearised so far.

From such a configuration, there are edges to configurations as follows:

Synchronisation. If some set of executions in *pending* can synchronise, giving the same results as required by the specification *spec*, then there is an edge to a configuration where the synchronising executions are moved into *linearised*, and *spec* is updated corresponding to the synchronisation;

Call. If the next event in the log is a call event, then there is an edge where that event is added to *pending*, and $i$ is advanced;

Return. If the next event in the log is a return event, and the corresponding execution is in *linearised*, then there is an edge where that execution is removed from *linearised*, and $i$ is advanced.

The initial configuration has $i$ at the start of the log, *pending* and *linearised* empty, and *spec* the initial state of the specification object. Target configurations have $i$ at the end of the log, and *pending* and *linearised* empty.

Any path from the initial configuration to a target configuration clearly represents an interleaving of a history of the specification object with $h$, as required for synchronisation linearisation. We can therefore search this graph using a standard algorithm. Our implementation uses depth-first search.

It is straightforward to adapt the search algorithm to also test for progress. We change the definition of a target configuration to have $i$ at the end of the log, *linearised* empty, and such that no set of executions in *pending* can synchronise: this ensures that we are dealing with a *maximal* synchronisation linearisation.

*6.1.1. Partial-order reduction*

We have investigated a form of partial-order reduction, which we call *ASAP linearisation*. The idea is that we try to linearise executions *as soon as possible*.

**Definition 17.** Let $h$ be a complete history of a synchronisation object, and let $h_s$ be a legal history of the corresponding specification object; and consider an interleaving, as required for synchronisation linearisation. We say that the interleaving is an *ASAP interleaving* if every event in $h_s$ appears either: (1) directly after the call event of one of the corresponding executions from $h$; or (2) directly after another event from $h_s$.

The following lemma shows that it suffices to consider ASAP interleavings.

**Lemma 18.** Let $h$ be a complete history of a synchronisation object, and let $h_s$ be a legal history of the corresponding specification object. If $h_s$ is a synchronisation linearisation of $h$, then there is an ASAP interleaving of them.

Proof. Consider an interleaving of $h$ and $h_s$, as required for synchronisation linearisation. We transform it into an ASAP interleaving as follows. Working

forwards through the interleaving, we move every event of $h_s$ earlier in the interleaving, as far as possible, without it moving past any of the corresponding call events, nor moving past any other event from $h_s$. This means that subsequently each such event follows either a corresponding call event or another event from $h_s$.

Each event from $h_s$ is still between the call and return events of the corresponding executions. Further, we do not reorder events from $h_s$ so the resulting interleaving is still an interleaving of $h$ and $h_s$.

Thus the resulting interleaving is an ASAP interleaving.

Our approach, then, is to trim the search graph by removing synchronisation edges that do not correspond to an ASAP linearisation: after a call edge, we attempt to linearise a synchronisation corresponding to that call, and then, if successful, to linearise an arbitrary sequence of other synchronisations; but we do not otherwise allow synchronisations.

Our experience is that this tactic is moderately successful. In some cases, it can reduce the total time to check histories by over 30%; although in some cases the gains are smaller, sometimes negligible. The gains seem highest in examples where there can be a reasonably large number of pending executions.

### *6.2. Complexity*

Consider the problem of testing whether a given concurrent history is synchronisation linearisable with respect to a given synchronisation specification object. We show that this problem is NP-complete in general.

We make use of a result from [19] concerning the complexity of the corresponding problem for linearisation. Let Variable be a linearisation specification object corresponding to an integer variable with get and set operations. Then the problem of deciding whether a given concurrent history is linearisable with respect to Variable is NP-complete in general.

Since standard linearisation is a special case of synchronisation linearisation (in the trivial case of unary synchronisations), this immediately implies that deciding synchronisation linearisation is NP-complete. However, even if we restrict to the non-trivial case of binary synchronisations, the result still holds. We consider concurrent synchronisation histories on an object with the following signature, which mimics the behaviour of a variable but via synchronisations.

```
object VariableSync{
  def op1(op: String, x: Int): Int
  def op2(u: Unit): Unit
}
```

The intention is that op1("get", x) acts like get(x), and op1("set", x) acts like set(x) (but returns -1). The op2 operation does nothing except synchronise with op1. This can be captured formally by the following synchronisation specification object.

```
object VariableSyncSpec{
  private var state = 0 // The value of the variable.
```

```
  def sync((op, x): (String, Int), u: Unit): (Int, Unit) =
    if(op == "get") (state, ()) else{ state = x; (−1, ()) }
}
```

Let ConcVariable be a concurrent datatype that represents an integer variable. Given a history $h$ of ConcVariable, we build a history $h'$ of VariableSync as follows. We replace every call or return of get(x) by (respectively) a call or return of $\mathsf{op}_1$("get", x); and we do similarly with sets. If there are $k$ calls of get or set in total, we prepend $k$ calls of $\mathsf{op}_2$, and append $k$ corresponding returns (in any order). Then it is clear that $h$ is linearisable with respect to Variable if and only if $h'$ is linearisable with respect to VariableSyncSpec. Deciding the former is NP-complete; hence the latter is also.

*6.3. The binary heterogeneous stateless case*

The result of the previous subsection used a *stateful* specification object. We now consider the *stateless* case. We show that for binary heterogeneous synchronisations, the problem of deciding whether a history is synchronisation linearisable can be decided in quadratic time. We consider the homogeneous case in the next subsection.

So consider a binary heterogeneous synchronisation object, whose specification object is stateless. In this case we do not need to worry about the order of synchronisations: if each individual synchronisation is correct, then any permutation will also be correct from the point of view of the specification object; and we can order the synchronisations in a way that is compatible with the concurrent history. Informally, the idea is to find matching operation executions in the concurrent history that could correspond to a particular synchronisation; we therefore reduce the problem to that of finding a matching in a graph.

Define two complete operation executions to be *compatible* if they could be synchronised, i.e. they overlap and the return values agree with those for the specification object. For $n$ executions of operations this can be calculated in $O(n^2)$.

Consider the bipartite graph where the two sets of nodes are executions of $\mathsf{op}_1$ and $\mathsf{op}_2$, respectively, and there is an edge between two executions if they are compatible. A synchronisation linearisation then corresponds to a total matching of this graph: given a total matching, we build a synchronisation linearisation by including events $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ (in an appropriate order) whenever there is an edge between $\mathsf{op}_1^{i_1}(x_1){:}y_1$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ in the matching; and conversely, each synchronisation linearisation corresponds to a total matching.

Thus we have reduced the problem to that of deciding whether there is a total matching, for which standard algorithms exist. We use the Ford-Fulkerson method [20], which runs in time $O(n^2)$.

It is straightforward to extend this to a mix of binary and unary synchronisations, again with a stateless specification object: the executions of unary operations can be considered in isolation.

This approach can be easily extended to also test for progress. It is enough to additionally check that no two pending executions could synchronise.

*6.4. The binary homogeneous stateless case*

We now consider the case of binary *homogeneous* synchronisations with a stateless specification object. This case is almost identical to the case with heterogeneous synchronisations, except the graph produced is not necessarily bipartite. Thus we have reduced the problem to that of finding a total matching in a general graph. This problem can be solved using, for example, the blossom algorithm [21], which runs in time $O(n^4)$.

In fact, our experiments use a simpler algorithm. We attempt to find a matching via a depth-first search: we pick a node $n$ that has not yet been matched, try matching it with some unmatched compatible node $n'$, and recurse on the remainder of the graph; if that recursive search is unsuccessful, we backtrack and try matching $n$ with a different node. We guide this search by the standard heuristic of, at each point, expanding the node $n$ that has fewest unmatched compatible nodes $n'$.

In our only example of this category, the Exchanger from the Introduction, we can choose the values to be exchanged randomly from a reasonably large range (say size 100). Then we can nearly always find a node $n$ for which there is a unique unmatched compatible node: this means that the algorithm nearly always runs in linear time. We expect that similar techniques could be used for other examples in this category.

*6.5. The non-binary stateless case*

It turns out that for synchronisations of arity greater than 2, the problem of deciding whether a history is synchronisation linearisable is NP-complete in general, even in the stateless case. We prove this fact by reduction from the following problem, which is known to be NP-complete [22].

**Definition 19.** The problem of finding a total matching in a 3-partite hypergraph is as follows: given disjoint finite sets $X$, $Y$ and $Z$ of the same cardinality, and a set $T \subseteq X \times Y \times Z$, find $U \subseteq T$ such that each member of $X$, $Y$ and $Z$ is included in precisely one element of $U$.

Suppose we are given an instance $(X, Y, Z, T)$ of the above problem. We construct a synchronisation specification and a corresponding history $h$ such that $h$ is synchronisation linearisable if and only if a total matching exists. The synchronisations are between operations as follows:

```
def op1(x: X): Unit
def op2(y: Y): Unit
def op3(z: Z): Unit
```

The synchronisations are specified by:

```
def sync(x: X, y: Y, z: Z): (Unit, Unit, Unit) = {
  require((x,y,z) ∈ T); ((), (), ())
}
```

The history $h$ starts with calls of $\mathsf{op}_1(x)$ for each $x \in X$, $\mathsf{op}_2(y)$ for each $y \in Y$, and $\mathsf{op}_3(z)$ for each $z \in Z$ (in any order); and then continues with returns of the same executions (in any order). It is clear that any synchronisation linearisation corresponds to a total matching, i.e. the executions that synchronise correspond to the total matching $U$. Hence finding a synchronisation linearisation is NP-complete.

Our implementation for these cases uses a depth-first search to find a matching, very much like in the binary homogeneous case.

## 7. Implementation

We have implemented a testing framework (in Scala). The framework supports both two-step linearisation and the direct algorithms. We have used the framework to implement testers for particular synchronisation objects[5]. We consider the framework to be straightforward to use: most of the boilerplate code is encapsulated within the framework; defining a tester for a new synchronisation object normally takes just a few minutes. Below, we concentrate our discussion on the part of the framework using the direct algorithms.

Figure 7 gives a stripped-down tester for a synchronous channel. (The full version can be used to test several different implementations with the same interface, and replaces the numeric constants by parameters that can be set on the command line.)

The worker function defines a worker thread that performs operations on the channel c. The function also takes parameters representing the thread's identity and a log object that logs representations of sends and receives. Here, each worker with an even identity performs 10 receives: the statement log(me, c.receive(), Receive) logs the call, performs the receive, and then logs the return. Similarly, each worker with an odd identity performs 10 sends of random values. This definition is designed so that an even number of workers with contiguous identities will not deadlock.

SyncChanSpec is the synchronisation specification object from earlier. The way executions synchronise is captured by the function matching, which is analogous to the synchronisation abstraction function from Section 2.4. This is a partial function whose domain defines which operation executions can synchronise together, and, in that case, the value each should return: here send(x) and receive() can synchronise, giving a result as defined by the synchronisation specification object. (Alternatively, the call to SyncChanSpec.sync could be in-lined.)

The function doTest performs a single test. This uses a BinaryStatelessTester object from the testing framework, which encapsulates the search from Section 6.3. Here, the tester runs 4 worker threads, and tests the resulting history against matching. If a non-synchronisation-linearisable history is recorded, it displays this for the user, indicating the first operation that couldn't be linearised.

[5]The implementation is available from `https://www.cs.ox.ac.uk/people/gavin.lowe/Synchronisation/`.

```
object ChanTester extends Tester{
  trait Op // Representation of operations within the log.
  case class Send(x: Int) extends Op
  case object Receive extends Op

  def worker(c: SyncChan[Int])(me: Int, log: HistoryLog[Op]) =
    for(i <- 0 until 10)
      if(me%2 == 0) log(me, c.receive(), Receive)
      else{ val x = Random.nextInt(100); log(me, c.send(x), Send(x)) }

  object SyncChanSpec{
    def sync(x: Int, u: Unit) = ((), x)
  }

  def matching: PartialFunction[(Op,Op), (Any,Any)] = {
    case (Send(x), Receive) => SyncChanSpec.sync(x, ()) // = ((), x).
  }

  /** Do a single test. Return true if it passes. */
  def doTest(): Boolean = {
    val c = new SyncChan[Int]
    new BinaryStatelessTester[Op](worker(c), 4, matching)()
  }

  def main(args: Array[String]) = {
    var i = 0; while(i < 5000 && doTest()) i += 1
  }
}
```

Figure 7: A simple tester for a synchronous channel.

The main function runs doTest either 5000 times or until an error is found. The tester can be adapted to test for synchronisation progressibility by passing a timeout duration to the BinaryStatelessTester.

Other classes of testers are similar. Details and examples can be found in the manual [23]. In the case of a stateful specification, the matching function takes the specification object as a parameter, and also returns the new value of the specification object. The framework directly supports two-step linearisation testing for binary synchronisations, but for other forms of synchronisation, the programmer has to define the appropriate automaton.

If a tester finds a history that is not synchronisation linearisable, it displays it. For example, Figure 8 gives such a history for an exchanger. Here executions 2 and 3 have correctly synchronised and exchanged their values, 76 and 58. Execution 1 has received execution 0's value, 13. However, execution 0 has received execution 3's value, rather than execution 1's. This does not, of course, identify what the bug is; but it does give some indication as to what has gone wrong, namely that execution 0 has been delayed and so failed to pick up the

```
0:   Call of Exchange(13)
1:   Call of Exchange(70)
1:   Return of 13 from Exchange(70)
2:   Call of Exchange(76)
3:   Call of Exchange(58)
3:   Return of 76 from Exchange(58)
2:   Return of 58 from Exchange(76)
0:   Return of 58 from Exchange(13)
Invocation 0 does not synchronise with any other operation.
```

Figure 8: A faulty history for an exchanger, showing a failure of synchronisation linearisability. The left-hand column gives an index for each operation execution.

```
0:   Call of Send(48)
1:   Call of Receive
2:   Call of Send(92)
3:   Call of Receive
1:   Return of 48 from Receive
0:   Return of () from Send(48)
Pending invocations 2 and 3 should have synchronised.
```

Figure 9: A faulty history for a synchronous channel, showing a failure of progressibility.

correct value.

Similarly, if a tester finds a failure of synchronisation progressibility, it displays the history. Figure 9 gives an example for a faulty synchronous channel. Here executions 0 and 1 have successfully synchronised. However, executions 2 and 3 have failed to synchronise when they should have done.

## 8. Experiments

In this section we describe experiments based on our testing framework.

We consider synchronisation objects implementing a number of interfaces, summarised in Figure 10. Most of the interfaces were described in earlier sections (namely synchronous channel, filter channel, exchanger, counter channel, barrier, enrollable barrier, timeout channel, timeout exchanger, closeable channel, and terminating queue).

The *men and women* problem involves two families of threads, known as men and women: each thread wants to pair off with a thread of the other type; each passes in its own identity, and expects to receive back the identity of the thread with which it has paired. In the *two families* problem, there are two families of threads, with $n$ threads of each family; each thread calls an operation $n$ times, and each execution should synchronise with a thread of the opposite family, but with a *different* thread each time. In the *one family* problem, there are $n$ threads, each of which calls an operation $n-1$ times, and each time should

| Category | Arity | Stateful? | Heterogeneous? |
|---|---|---|---|
| Synchronous channel | 2 | N | Y |
| Filter channel | 2 | N | Y |
| Men and women | 2 | N | Y |
| Exchanger | 2 | N | N |
| Counter channel | 2 | Y | Y |
| Two families | 2 | Y | Y |
| One family | 2 | Y | N |
| ABC | 3 | N | Y |
| Barrier | $n$ | N | N |
| Enrollable barrier | $1..n$, 1 | Y | N |
| Timeout channel | 2, 1 | N | Y |
| Timeout exchanger | 2, 1 | N | N |
| Closeable channel | 2, 1 | Y | Y |
| Terminating queue | 1, $n$ | Y | N |
| Atomic broadcast | $n+1$ | N | N |

Figure 10: Example interfaces of synchronisation objects.

synchronise with a *different* thread. Finally, the $ABC$ problem can be thought of as a ternary version of the men and women problem: there are three types of threads, A, B and C; each synchronisation involves one thread of each type. The *atomic broadcast* problem allows a single sender to broadcast a message to $n$ receivers synchronously.

For each interface, we have implemented a tester using the two-step approach from Section 4, and also a tester using the direct algorithm from Section 6.

For each interface, we have produced a correct implementation. Most implementations are quite short, about 30 lines of code on average. An exception is the implementation of a synchronous channel from the SCL library [24] (which supports closing, timeouts and alternation), which is a few hundred lines. However, as noted in the Introduction, despite most implementations begin quite short, they can be hard to get right.

For most interfaces, we have also implemented one or more faulty versions that fail to achieve either synchronisation linearisation or progressibility. The faulty versions mostly have realistic mistakes: a few are genuine bugs; others are similar to bugs we have seen from students. Some of the bugs are caused by an operation failing to wait for the previous synchronisation to finish, which can lead to interference between the two synchronisations. Other bugs allow threads to run in an unexpected order (because of unfortunate scheduling), which falsifies the intended logic. A particularly subtle bug concerns a previous implementation of the channel in SCL: here, if three threads run concurrently, one sending, one receiving, and one closing the channel, it is possible for the receiver to return as if it has synchronised, then for the closing to take effect, and then for the receiver to find the channel closed and signal a failure; this is

an error, since either both threads should think they synchronised, or neither.

We describe various experiments below. The purpose of testing is to find bugs. We therefore concentrate on the time taken to find bugs. If the technique is fast to find bugs when they exist, then the failure to find bugs on other examples should give us reasonable confidence that none exists.

We consider the following questions.

- Which works better, the direct algorithm or the two-step algorithm? The experiments show that the direct algorithms are faster in most cases. Further, the direct algorithms scale much better as the number of operation executions per run increases.

- How should we choose parameters (number of threads to run, number of iterations performed by each thread, etc.) for testing? The experiments suggest that, for both types of tester, it is best to use a small number of threads (typically two to four), each performing a small number of operations (typically about four).

- Is this approach effective at finding bugs? The experiments suggest it is. For each of the incorrect examples we considered, each approach found an error for synchronisation linearisation within about a second on average, and somewhat faster on most examples. For synchronisation progressibility, they were typically a few hundred milliseconds slower.

The experiments were performed on a dedicated eight-core machine (two 2.40GHz Intel(R) Xeon(R) E5620 CPUs, with 12GB of RAM, but limited to 4GB of heap space).

In each experiment below, we consider a synchronisation object with a bug that causes a failure of synchronisation linearisation, but does not lead to a deadlock. We performed a number of *runs* of a tester on the synchronisation object. In each run, a particular number of threads performed a particular number of operation calls on the synchronisation object; the relevant algorithm was then used to decide whether the log history was synchronisation linearisable or two-step linearisable.

Each *observation* performed multiple runs until an error was detected, and recorded the time taken. Each observation was performed as a separate operating system process, with the aim of making observations independent, avoiding dependencies caused by, for example, garbage collection, caching behaviour, and just-in-time compilation. Thus each observation was as close as possible to a normal use case.

For each data point in the experiments, we performed 100 observations. We give the average time to find an arror, and a 95%-confidence interval for that average (following [25]). The number of observations is chosen so as to obtain a reasonably small confidence interval, but avoiding excessively long experiments.

We start with the second of the questions above, how to choose parameters for testing. Each of the graphs in Figures 11 and 12 concerns a particular tester. Each data point represents a particular number $p$ of worker threads (given in the

key), and a particular number of operation executions per run by each thread (given on the $X$-axis). The $Y$-axis gives the time in milliseconds. (The graph for the two-step tester applied to a barrier omits a plot for $p = 10$, with times of around 5000ms, which does not fit on the axes.)

The experiments suggest that both types of tester work best with a fairly small number of worker threads (typically two to four), each performing a fairly small number of operations per run (typically about four).

Some bugs are exhibited only when the number of threads exceeds the arity of a synchronisation: for example, in the ABC problem, two different synchronisations interfere to produce the error. We therefore recommend including enough threads to find such bugs.

On the other hand, the number of operations performed by each thread should not be too small. For example, for the ABC problem, the testers are rather slow to find the bug when each thread performs only two operations per run. The reason in this case seems to be that on most runs some of the threads finish their operations before others start, which removes the possibility of interference between synchronisations.

Using rather short runs has an additional advantage: if the tester does find an erroneous history, a shorter history is normally easier to interpret than a longer one.

The results also suggest that the direct algorithms scale better than the two-step tester as the number of threads increases. Figure 13 investigates this further. Each graph considers one type of synchronisation object, with the two plots representing the two testers. Each data point considers a particular number of threads (given on the $X$-axis). The $Y$-axis again gives the time in milliseconds. We omit results for cases where the two-step tester sometimes ran out of memory.

The results confirm that the two-step approach does not scale well with the number of threads. In each case, the running time increases dramatically at a particular point, and tests often fail. The running time also becomes erratic (as indicated by the wide confidence intervals): some runs take an extremely long time. By contrast, the direct algorithms scale well.

We believe the reason the two-step approach scales poorly is as follows. The linearisation tester tries to find a linearisation order for operation executions, via a depth-first search. At each step, it picks a particular operation execution to try to linearise next. If it picks wrongly, it might have to consider many nodes of the search graph before backtracking. This can be the case with two-step testing, because if it picks the wrong $\mathsf{op}_1$ to try to linearise, it will only discover this fact when it reaches the corresponding $\overline{\mathsf{op}}_1$ and finds the wrong value is returned. This latter event might be much later in the history. To reach it, the tester has to consider many possibilities for ordering other operation executions.

Figure 14 gives times to find various bugs with the two testers. Based on the earlier experiments, in most cases we ran four threads, each executing four operations; for the ABC testers, we ran six threads, each executing four operations; for the (untimed) exchanger, we ran eight threads, each performing a single operation (recall from Section 3 that this avoids deadlocks); for the two-

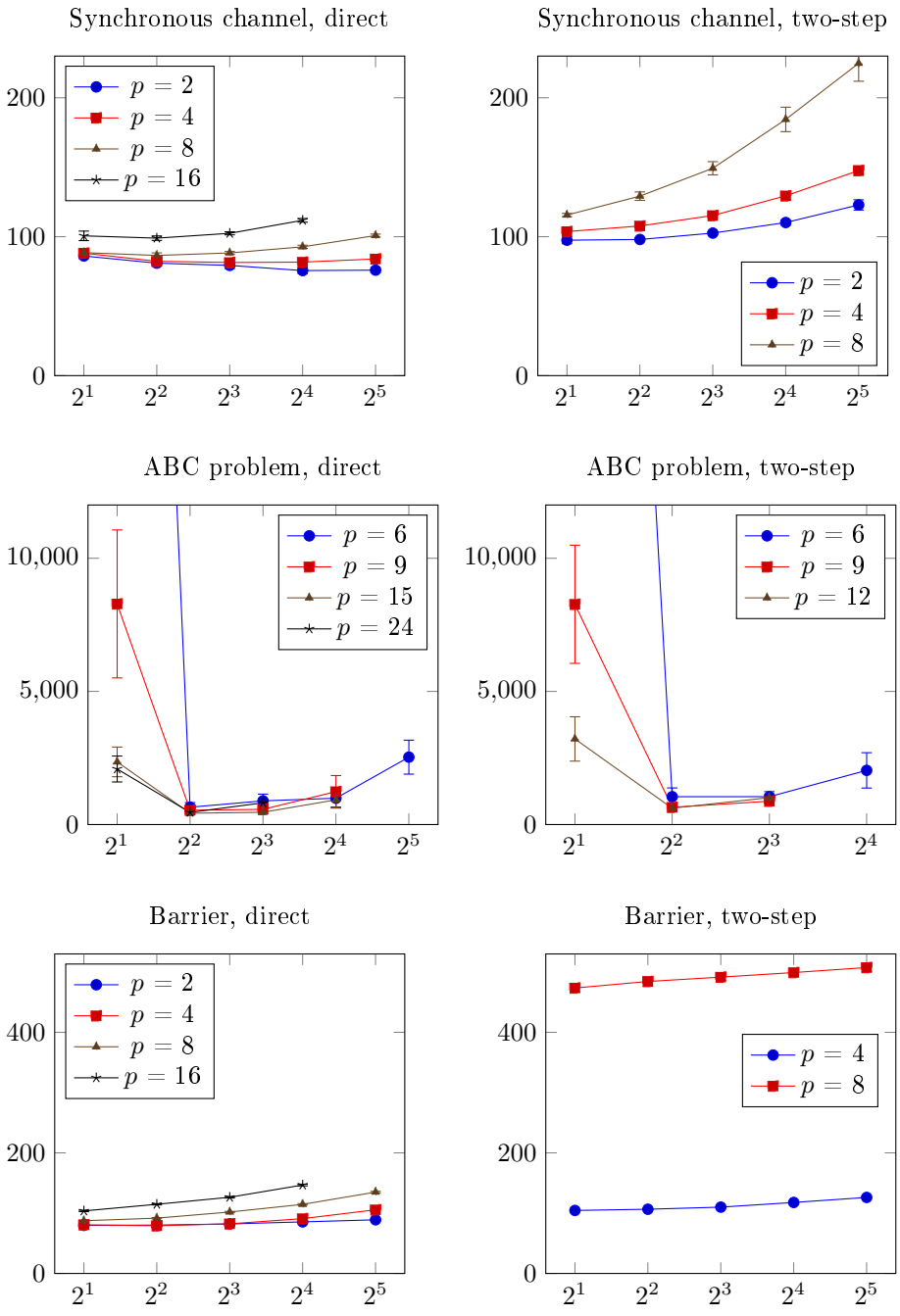



Figure 11: Effect of choices of parameters for testers for a synchronous channel the ABC problem, and a barrier.

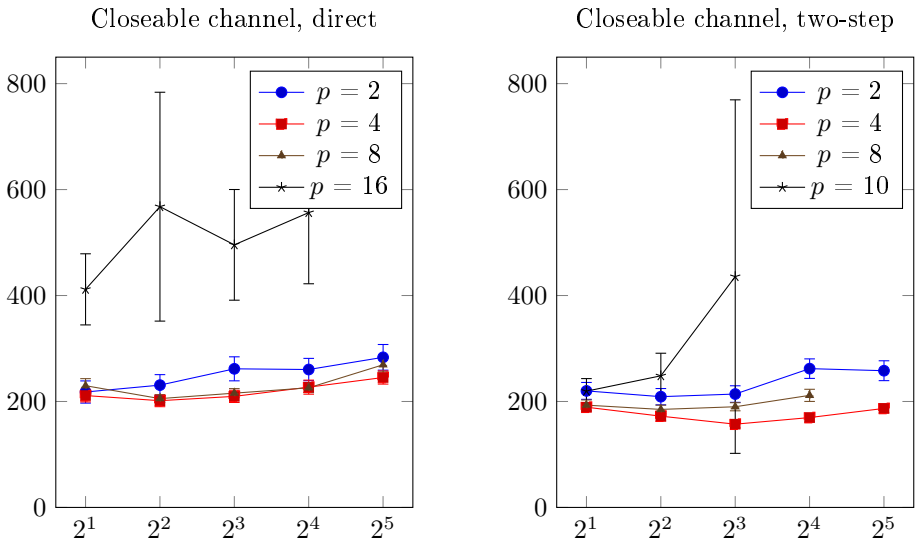



Figure 12: Effect of choices of parameters for testers for a closeable channel.

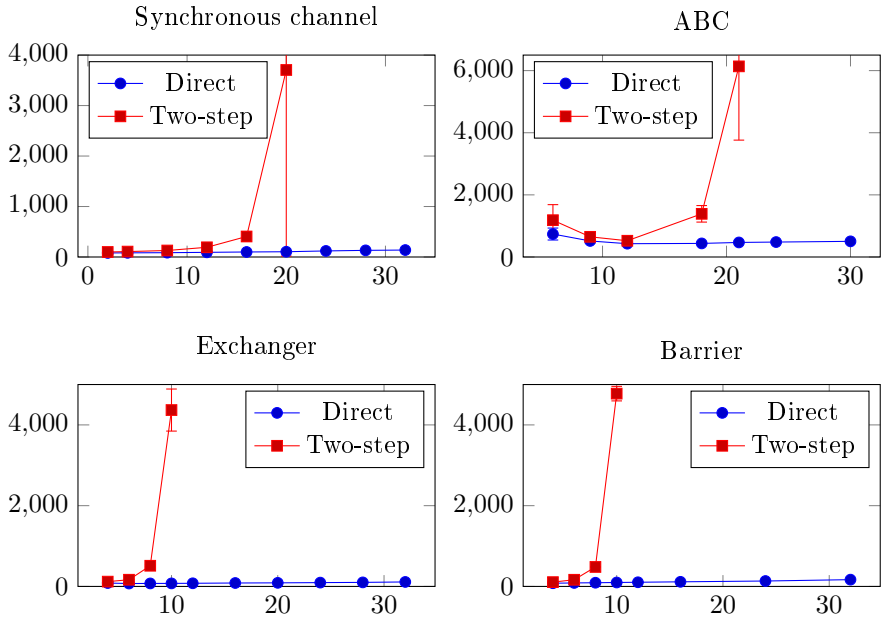



Figure 13: The effect of increasing the number of threads on the time to find bugs.

| Synchronisation object | Direct | Two-step |
|---|---|---|
| Synchronous channel | $85 \pm 3$ | $109 \pm 2$ |
| Filter channel | $77 \pm 1$ | $108 \pm 3$ |
| Men and women | $75 \pm 1$ | $108 \pm 4$ |
| Exchanger | $73 \pm 1$ | $511 \pm 25$ |
| Counter channel | $94 \pm 2$ | $106 \pm 1$ |
| Two families | $299 \pm 34$ | $267 \pm 25$ |
| One family | $336 \pm 27$ | $437 \pm 39$ |
| ABC | $717 \pm 225$ | $928 \pm 216$ |
| Barrier | $80 \pm 1$ | $106 \pm 1$ |
| Enrollable barrier | $132 \pm 5$ | $149 \pm 6$ |
| Timeout channel | $121 \pm 5$ | $135 \pm 5$ |
| Timeout exchanger | $288 \pm 64$ | $231 \pm 18$ |
| Closeable channel | $191 \pm 11$ | $173 \pm 9$ |
| Terminating queue | $103 \pm 1$ | $105 \pm 1$ |

Figure 14: Times to find bugs affecting synchronisation linearisation.

families object, we ran four threads (two from each family), each performing two operations; and for the one-family object, we ran four threads, each performing three operations. The table gives average times in milliseconds to detect the bug, with 95%-confidence intervals.

All the testers work well, with the average time to find each bug below one second. Of course, other bugs might be harder to find, because they are triggered on fewer runs. However, our results do suggest that our techniques are effective at finding most bugs.

In most cases, the direct tester is faster than the two-step one; and the two-step tester is never significantly faster. We therefore recommend the direct algorithms. This approach has two additional advantages. Our experience is that it is easier to create the testing program based on the direct algorithms, whereas using the two-step approach involves designing and encoding the appropriate automaton, which can be somewhat tricky. Further, error histories found by the direct algorithms tend to be easier to understand: the corresponding two-step history is longer, because of the second step of some operations, and this can be distracting.

We now consider synchronisation progressibility. Our experience is that if a synchronisation object does not satisfy progressibility, then this can lead to a total deadlock. Thus, in most cases, testing for synchronisation linearisability will also detect progressibility bugs. However, this is no guarantee.

Recall that our approach is to interrupt threads after a suitable duration. This duration should be chosen so that if any threads have not returned by this point, then (almost certainly) they really are stuck. Our informal experiments suggest that a duration of 100ms is appropriate (at least, on the architecture we were using): we have not observed any false positives with this duration, but

| | |
|---|---|
| Synchronous channel | $323 \pm 39$ |
| Filter channel | $338 \pm 50$ |
| Men and women | $232 \pm 19$ |
| One family | $1098 \pm 278$ |
| ABC | $1080 \pm 186$ |
| Barrier | $169 \pm 3$ |

Figure 15: Times to find bugs affecting synchronisation progressibility.

did with a shorter duration of 80ms. If an error is found (and the cause is not immediately apparent), it is straightforward to increase the duration and see whether the error still occurs. Of course, a larger value for this duration does increase the time that a given number of runs will take: a tester may take the view that it's easiest to use a larger duration, and just leave the tests running for longer.

Figure 15 gives results for the time to find various bugs that lead to a failure of progressibility. Each uses the relevant direct algorithm: recall that two-step linearisation cannot be used to test progressibility.

Again, each bug is found quickly, within about a second. Testing for progressibility is normally a bit slower than for linearisation, because of the need to interrupt the worker threads, but only after allowing enough time that we can be confident that they really have got stuck.

One class of errors that we believe our testing framework will be less successful at finding is so-called *spurious wake-ups*. Scala inherits a wait/notify mechanism from Java. A thread that calls `wait()` is supposed to suspend until another thread calls `notify()`. Unfortunately, a waiting thread may spuriously wake-up and continue without being notified. Programmers are expected to guard against spurious wake-ups — but sometimes they don't, and this leads to bugs. However, spurious wake-ups happen sufficiently rarely that they might not be found by testing in a reasonable amount of time.

## 9. Conclusions

In this paper we have studied synchronisation objects. We have proposed the correctness conditions of synchronisation linearisation and synchronisation progressibility. We have shown how to reduce synchronisation linearisation to standard synchronisation, where some operations are linearised in two steps.

We have studied how to carry our testing on implementations of synchronisation objects. The approach is effective: the testing code is easy to write; and the testers normally find errors quickly. We have also studied the complexity of algorithms for deciding whether a history is synchronisation linearisable.

### *9.1. Related work*

As noted earlier, synchronisation linearisation is equivalent to Neiger's set linearisation [12], and Hemed et al.'s concurrency-aware linearisation [13].

Interval linearizability

Scherer and Scott

Scherer et al. [26] consider a setting where an operation may be linearised in several steps, similar to our two-step linearisation. Their interest is in partial concurrent datatypes, where an operation may have a nontrivial precondition. Such an operation is linearised by (1) an initial request (where the operation registers itself); (2) some number of unsuccessful follow-ups (where the precondition is found not to be satisfied); (3) a successful follow-up (where the precondition holds, and the operation takes effect).

Our analysis technique in this paper has been software testing of implementations of synchronisation objects. However, one can also apply model checking to the problem. The companion paper [24] analyses a library of communication primitives (including a closeable channel with timed operations, and alternation), using CSP [17] and its model checker FDR [27]. Synchronisation linearisation is captured by presenting a generic way of capturing the traces (equivalent to histories in this paper) that satisfy the property. Synchronisation linearisation is a safety property, so is tested in the traces model. It turns out that testing against the same specification, but in the failures-divergences model, captures synchronisation progressibility. The library includes several different synchronisation objects that are analysed in this way. An error is identified on a previous version of the library; but the revised version is shown to be synchronisation linearisable and progressible.

*Acknowledgements*

We would like to thank the anonymous reviewers for their useful comments and suggestions.

## References

[1] C. A. R. Hoare, Communicating sequential processes, Communications of the ACM 21 (8) (1978) 666–677.

[2] INMOS Limited, occam Programming Manual, Prentice Hall, 1983.

[3] G. Andrews, Concurrent Programming: Principles and Practice, Benjamin/Cummings, 1991.

[4] P. Welch, N. Brown, J. Moores, K. Chalmers, B. Sputh, Integrating and extending JCSP, in: Proceedings of Communicating Process Architectures, 2007.

[5] B. Sufrin, Communicating Scala Objects, in: Proceedings of Communicating Process Architectures, 2008.

[6] A. A. A. Donovan, B. W. Kernighan, The Go Programming Language, Addison-Wesley, 2020.

[7] D. Hendler, N. Shavit, L. Yerushalmi, A scalable lock-free stack algorithm, in: Proceedings of the Sixteenth Annual Symposium on Parallelism in Algorithms and Architectures, 2004, pp. 206–215.

[8] M. Herlihy, N. Shavit, The Art of Multiprocessor Programming, Morgan Kaufmann, 2012.

[9] G. Lowe, Scala Concurrency Library (SCL), online repository at `https://github.com/GavinLowe1967/Scala-Concurrency-Library` (2025).

[10] P. Welch, N. Brown, J. Moores, K. Chalmers, B. Sputh, Alting barriers: synchronisation with choice in Java using JCSP, Concurrency and Computation: Practice and Experience 22 (8) (2010).

[11] M. Herlihy, J. M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.

[12] G. Neiger, Set linearizability (brief announcement), in: Proceedings of the 13th Symposium on Principles of Distributed Computing, ACM Press, 1994, p. 396.

[13] N. Hemed, N. Rinetzky, V. Vafeiadis, Modular verification of concurrency-aware linearizability, in: Proceedings of the 29th Symposium on Principles of Distributed Computing, Vol. 9363 of LNCS, Springer, 2015, pp. 371–387.

[14] J. M. Wing, C. Gong, Testing and verifying concurrent objects, Journal of Parallel and Distributed Computing 17 (1993) 164–182.

[15] G. Lowe, Testing for linearizability, Concurrency and Computation: Practice and Experience 29 (14) (2016).

[16] D. Hendler, N. Shavit, L. Yerushalmi, A scalable lock-free stack algorithm, in: SPAA, 2004.

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

[18] H. Sundell, P. Tsigas, Fast and lock-free concurrent priority queues for multi-thread systems, Journal of Parallel and Distributed Computing 65 (5) (2005) 609–627.

[19] P. B. Gibbons, E. Korach, Testing shared memories, SIAM Journal of Computing 26 (4) (1997) 1208–1244.

[20] L. R. Ford, Jr., D. R. Fulkerson, Maximal flow through a network, Canadian Journal of Mathematics 8 (1956) 399–404.

[21] J. Edmonds, Paths, trees, and flowers, Canadian Journal of Mathematics 17 (1965) 449–467. doi:10.4153/CJM-1965-045-4.

[22] R. M. Karp, Reducibility among combinatorial problems, in: R. E. Miller, J. W. Thatcher, J. D. Bohlinger (Eds.), Complexity of Computer Computations, Springer US, 1972, pp. 85–103.

[23] G. Lowe, Testing Synchronisation Objects, available via `https://www.cs.ox.ac.uk/people/gavin.lowe/UnderstandingSynchronisation/` (2025).

[24] G. Lowe, Analysing a library of concurrency primitives using CSP, Formal Aspects of ComputingForthcoming (2026).

[25] A. Georges, D. Buytaert, L. Eeckhout, Statistically rigorous Java performance evaluation, in: Proceedings of OOPSLA '07, 2007.

[26] W. N. Scherer III, D. Lea, M. L. Scott, Scalable synchronous queues, Communicatons of the ACM 52 (5) (2009) 100–111.

[27] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, A. W. Roscoe, FDR3: a parallel refinement checker for CSP, International Journal on Software Tools for Technology Transfer (2015).