Understanding Synchronisation

Jonathan Lawrence and Gavin Lowe

November 17, 2021

Abstract

. . .

1 Introduction

A common step of many concurrent programs involves two or more threads *synchronising*: each thread waits until other relevant threads have reached the synchronisation point before continuing; in addition, the threads can exchange data. We study synchronisations in this paper.

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. Such a channel might have signature[1]

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

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

Sometimes an invocation may synchronise with an invocation of the same operation. For example, an *exchanger* has the following signature.

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

[1] The type Unit is the type that contains a single value, the *unit value*, denoted ().

When two threads call exchange, they each receive the value passed in by the other. When invocations of two different operations synchronise, we use the term *heterogeneous*; where two invocations of the same operation synchronise, we use the term *homogeneous*.

For some synchronisation objects, synchronisations might involve more than two threads. For example, an object of the following class

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

can be used to synchronise n threads, known as a *barrier synchronisation*: each thread calls sync, and no invocation returns until all n have called it.

A *combining barrier* also allows each thread to submit a parameter, and for all to receive back some function of those parameters.[2]

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

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 associative and commutative, this result is independent of the order of the parameters).

In addition, we allow the synchronisations to be mediated by an object that maintains some state between synchronisations. As an example, consider a synchronous channel that, in addition, maintains a sequence counter, and such that both invocations receive the value of this counter.

```
class SyncChanCounter{
  private var counter: Int
  def send(x: A): Int
  def receive(): (A, Int)
}
```

Some synchronisation objects allow different modes of synchronisation. For example, consider a synchronous channel with timeouts: each invocation might synchronise with another invocation, or might timeout without synchronisation. Such a channel might have a signature as follows.

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

[2] The Scala type (A,A) => A represents functions from pairs of A to 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 invocation of each operation may or may not synchronise with an invocation of the other operation.

A *termination-detecting queue* can also be thought of as a stateful synchronisation object with multiple modes. Such an object acts like a standard concurrent queue, except if all the threads are attempting to dequeue, but the queue is empty, then they all return a special value to indicate this fact. In many concurrent algorithms, such as a concurrent graph search, this latter outcome indicates that the algorithm should terminate. Such a termination-detecting queue might have the following signature, where a dequeue returns the value None to indicate the termination case.

```
class TerminationDetectingQueue(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 termination-detecting queue combines the functionality of a concurrent datatype and a synchronisation object.

In this paper, we consider what it means for one of these synchronisation objects to be correct, and techniques for testing correctness.

In Section 2 we describe how to specify a synchronisation object. The definition has similarities with the standard definition of *linearisation* for concurrent datatypes except is talks about synchronisations between invocations, rather than single invocations: we call the property *synchronisation linearisation*.

In Section 3 we consider the relationship between synchronisation linearisation and (standard) linearisation. We show that the two notions are different; but we show that synchronisation linearisation corresponds to a small adaptation of linearisation, where one of the operations of the synchronisation object corresponds to *two* operations of the object used to specify linearisation.

In Section 4 we consider how the property of synchronisation linearisation can be analysed via model checking.

In Section 5 we consider how to build testing frameworks for synchronisation objects. More here.

[3] The type Option[A] contains the union of such values.

2 Specifying synchronisations

In this section we describe how synchronisations can be formally specified. For ease of exposition, we start by *heterogeneous binary* synchronisation in this section, where every synchronisation is between invocations of two different operations. We generalise at the end of this section.

We assume that the synchronisation object has two operations, each of which has a single parameter, with signatures as follows.

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

(We can model a concrete operation that takes $k > 1$ parameters by an operation that takes a k-tuple as its parameter; we can model a concrete operation that takes no parameters by an operation that takes a Unit parameter.) In addition, the synchronisation object might have some state, state: S. Each invocation of op_1 must synchronise with an invocation of op_2, and vice versa. The result of each invocation may depend on the two parameters x_1 and 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 invocations (which implies that the invocation must overlap): we refer to this point as the *synchronisation point*.

Each synchronisation object 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 invocations $op_1(x_1)$ and $op_2(x_2)$ synchronise, then the results y_1 and y_2 of the invocations are such that $sync(x_1, x_2)$ could return the pair (y_1, y_2). The specification object might have some private state that is accessed and updated within sync. Note that invocations of sync occur *sequentially*.

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 typical definition of the specification obejct 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 invocations. The `require` clause of `sync` specifies a precondition for the synchronisation to take place. The values res_1 and res_2 represent the results that should be returned by the corresponding invocations of op_1 and 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 as follows, with empty state

```
class SyncChanSpec{
  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{
  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 specification object with operation

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

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

2.1 Linearisability

We formalise below precisely the allowable behaviours captured by a particular synchronisation specification object. Our definition has much in common with the well known notion of *linearisation* [?], used for specifying concurrent datatypes; so we start by reviewing that notion. There are a number of equivalent ways of defining linearisation: we choose a way that will be convenient subsequently.

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

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

In each case, op is an operation of o. Here i is a *invocation identity*, used to identify a particular invocation, and to link the call and corresponding return. In order to be well formed, each invocation identity must appear on at most one call event and at most one return event; and for each event $\mathsf{return}.op^i{:}y$, the history must contain an earlier event $\mathsf{call}.op^i(x)$, i.e. for the same operation and invocation 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 invocation identifier. A concurrent history is *complete* if for every call event, there is a matching return event, i.e. no invocation is still pending at the end of the history.

For example, consider the following complete concurrent history of a concurrent object 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: here, time runs from left to right; each horizontal line represents an operation invocation, with the left-hand end representing the call event, and the right-hand end representing the return event.

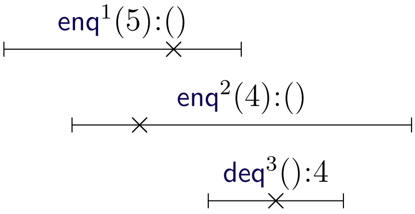


Figure 1: Timeline representing the linearisation example.

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

- $op^i(x) : y$ representing an invocation of operation *op* with parameter x, returning result y; again i is an invocation 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 invocation, the precondition is satisfied, and the return value is as for the definition of the operation in *Spec*.

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 corresponding to the same invocations, 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 and h_s are *compatible* 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 invocations of h appeared to take place in the order described by h_s, and that that order is consistent (in terms of the satisfaction of preconditions and values returned) with the specification object.

Continuing the running example, the histories h and h_s are compatible, as evidenced by the interleaving

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

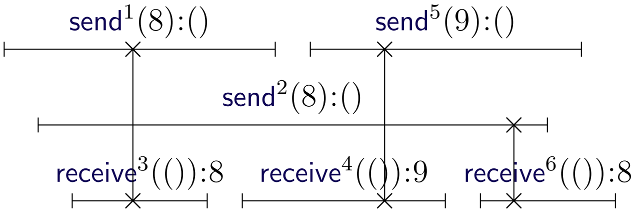


Figure 2: Timeline representing the synchronisation example.

which is again illustrated in Figure 1.

We say that a complete history h is *linearisable* with respect to *Spec* if there is a corresponding valid history h_s of *Spec* such that h and h_s are compatible.

A concurrent history might not be complete, i.e. it might have some pending invocations. An *extension* of a history h is formed by adding zero or more return events corresponding to pending invocations. We write $complete(h)$ for the subsequence of h formed by removing all call events corresponding to pending invocations. We say that a (not necessarily complete) concurrent history h is *linearisable* if there is an extension h' of h such that $complete(h')$ is linearisable. We say that a concurrent object is linearisable if all of its histories are linearisable.

2.2 Synchronisation linearisability

We now adapt the definition of linearisability to synchronisations. We consider a concurrent history of the synchronisation object *Sync*, as with linearisability; in the case of binary synchronisation, this will contain events corresponding to the operations op_1 and op_2.

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

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

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 invocation of sync with parameters (x_1,x_2) and result (y_1,y_2). The event's invocation identity is (i_1,i_2): each of i_1 and i_2 must appear at most once in the history.

Such a history is *legal* if is consistent with *Spec*. Informally, a sync event with identity (i_1, i_2) represents a synchronisation between the invocations $op_1^{i_1}$ and $op_2^{i_2}$ in a history of the corresponding synchronisation object.

For example, the following is a legal history of $\mathsf{SyncChanSpec}$.

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

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 vertical line. This particular synchronisation specification object is stateless, so in fact any permutation of this history 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.

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:

- For each sync event with identity (i_1, i_2) in h_s, h contains an invocation of op_1 with identity i_1 and an invocation of op_2 with identity i_2;
- For each invocation of op_1 with identity i_1 in h, h_s contains a sync event with identity (i_1, i_2) for some i_2;
- For each invocation of op_2 with identity i_2 in h, h_s contains a sync event with identity (i_1, i_2) for some i_1.

We say that a complete history h of *Sync* and a corresponding legal history h_s of *Spec* are *synchronisation compatible* if there is some way of interleaving them such that each event $\mathsf{sync}^{i_1,i_2}(x_1, x_2) : (y_1, y_2)$ occurs between $\mathsf{call}.op_1^{i_1}(x_1)$ and $\mathsf{return}.op_1^{i_1} : y_1$, and between $\mathsf{call}.op_2^{i_2}(x_2)$ and $\mathsf{return}.op_2^{i_2} : y_2$.

In the running example, the histories h and h_s are synchronisation compatible, as evidenced by the interleaving illustrated in Figure 2.

We say that a complete history h of *Sync* is *synchronisation linearisable* if there is a corresponding legal history h_s of *Spec* such that h and h_s are synchronisation compatible.

We say that a (not necessarily complete) concurrent history h is *synchronisation linearisable* if there is an extension h' of h such that $complete(h)$ is synchronisation linearisable. We say that a synchronisation object is synchronisation linearisable if all of its histories are synchronisation linearisable.

Is the definition compositional? I think so.

2.3 Variations

Above we considered heterogeneous binary synchronisations, i.e. two invocations of different operations, with a single mode of synchronisation.

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

```
  def opj(xj: Aj): Bj   for j = 1, . . . , k,
```

where the op_j might not be distinct. The specification object will have a corresponding operation

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

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

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

A history of the specification object will have corresponding events $\mathsf{sync}^{i_1,\ldots,i_k}(x_1, \ldots, x_k) : (y_1, \ldots, y_k)$. The definition of synchronisation compatibility is an obvious adaptation of earlier: in the interleaving of the complete history of the synchronisation history and the history of the specification object, each $\mathsf{sync}^{i_1,\ldots,i_k}(x_1, \ldots, x_k) : (y_1, \ldots, y_k)$ occurs between $\mathsf{call}.op_1^{i_j}(x_j)$ and $\mathsf{return}.op_j^{i_j} : y_j$ for each $j = 1, \ldots, k$. The definition of synchronisation linearisability follows in the obvious way.

It is also straightforward to adapt the definitions to deal with multiple modes of synchronisation: the specification object has a different operation for each mode. For example, recall the TimeoutChannel from the Introduction, where sends and receives may timeout and return without synchronisation. The corresponding specification object would be:

```
class TimeoutChannelSpec{
  def syncs(x: A) = false
  def syncr(u: Unit) = None
  def syncs,r(x: A, u: Unit) = (true, Some(x))
}
```

The operation sync_s corresponds to a send returning without synchronising; likewise sync_r corresponds to a receive returning without synchronising; and $\mathsf{sync}_{s,r}$ corresponds to a send and receive synchronising. The formal definition of synchronisation linearisation is the obvious adaptation of the earlier definition.

3 Relating synchronisation and linearisation

In this section we describe the relationship between synchronisation linearisation and standard linearisation. Below, we show that the two notions are different: synchronisation linearisation cannot, in general, be captured directly as standard linearisation. However, we show that synchronisation linearisation corresponds to a small adaptation of linearisation, where one of the operations of the synchronisation object corresponds to *two* operations of the linearisation specification object.

It is clear that standard linearisation is equivalent to synchronisation linearisation in the (rather trivial) case that no operations synchronise, so each operation of the synchronisation specification object corresponds to a single operation of the concurrent object.

We show that, given a synchronisation linearisability specification object *SyncSpec*, it is not, in general, possible to find a linearisability syncronisation specification *Spec* such that for every history h, h is synchronisation linearisable with respect to *SyncSpec* if and only if h is linearisable with respect to *Spec*.

For example, consider the example of a synchronous channel from Section 2, where synchronisation linearisation is captured by SyncChanSpec. Assume (for a contradiction) that the same property can be captured by linerisation with respect to linearisability specification *Spec*. Consider the history

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

This is synchronisation linearisable with respect to SyncChanSpec. By the assumption, there must be a legal history h_s of *Spec* such that h and h_s are compatible. Without loss of generality, suppose the send in h_s occurs before the receive, i.e.

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

But the history

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

is also compatible with respect to 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 linearisability specification *Spec* exists.

3.1 Two-step linearisability

Note: I don't think we want both this and the construction in Section 5.2.

We now show that synchronisation linearisability corresponds to a small adaptation of linearisability, but where one of the operations on the concurrent object corresponds to *two* operations of the linearisability specification object. We define what we mean by this, and then prove the correspondence in the next subsection. In the definitions below, we describe just the differences from standard linearisation, to avoid repetition.

Given a synchronisation object with operations op_1 and op_2, we will consider a linearisability specification object with signature

```
class TwoStepLinSpec{
  def $\mathsf{op}_1(\mathsf{x}_1\text{: }\mathsf{A}_1)$: Unit
  def $\overline{\mathsf{op}}_1()$: $\mathsf{B}_1$
  def $\mathsf{op}_2(\mathsf{x}_2\text{: }\mathsf{A}_2)$: $\mathsf{B}_2$
}
```

The idea is that the operation op_1 on the concurrent object will be linearised by the composition of the two operations op_1 and $\overline{\mathsf{op}}_1$; but operation op_2 on the concurrent object will be linearised by just the operation op_2 of the specification object, as before. We call such an object a *two-step linearisability specification object.*

We define a history h_s of such a two-step specification object much as in Section 2.1, with the addition that for each event $\overline{\mathsf{op}}^i_1():y$ in h_s, we require that there is an earlier event $\mathsf{op}^i_1(x):()$ in h_s with the same invocation identity; other than in this regard, invocation 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 corresponding to the same invocations in the following sense:

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

We say that h and h_s are *two-step compatible* if there is some way of interleaving the two histories such that

- Each $\mathsf{op}^i_1(x):()$ and $\overline{\mathsf{op}}^i_1():y$ occur between $\mathsf{call.op}^i_1(x)$ and $\mathsf{return.op}^i_1:y$, in that order;
- Each $\mathsf{op}^i_2(x):y$ occurs between $\mathsf{call.op}^i_2(x)$ and $\mathsf{return.op}^i_2:y$.

For example, consider a synchronous channel, with send corresponding to op_1, and receive corresponding to op_2. Then the following would be an interleaving of two-step compatible histories of the synchronisation object and the corresponding specification object.

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

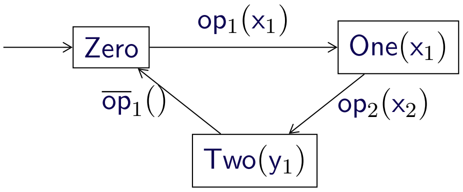
The definition of two-step linearisability then follows from this definition of two-step compatability, precisely as in Section 2.1.

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 is below: the specification's behaviour is described by the automaton on the right.[4]

```
trait State
case class Zero extends State
case class One($x_1$: $A_1$) extends State
case class Two($y_1$: $B_1$) extends State
```



```
class TwoStepLinSpec{
  private var state: State = Zero
  def $op_1$($x_1$: $A_1$): Unit = {
    require(state.isInstanceOf[Zero]); state = One($x_1$)
  }
  def $op_2$($x_2$: $A_2$): $B_2$ = {
    require(state.isInstanceOf[One]); val One($x_1$) = state
    val ($y_1$, $y_2$) = SyncSpec.sync($x_1$, $x_2$); state = Two($y_1$); $y_2$
  }
  def $\overline{op}_1$(): $B_1$ = {
    require(state.isInstanceOf[Two]); val Two($y_1$) = state; state = Zero; $y_1$
  }
}
```

[4] Defining the subclasses of State as case classes allows pattern matching against such values. For example, the statement val One(x_1) = state succeeds only if state has type One, and binds the name x_1 to the value of the x_1 field of state.

The definition forces the operations to take place in the order described by the automaton. In addition, the op_2 operation calls the sync method on $\mathsf{SyncSpec}$, to calculate the return values and to update $\mathsf{SyncSpec}$'s state; it stores op_1's result in the state.

The following lemma follows immediately from the construction of $\mathsf{TwoStepLinSpec}$.

Lemma 1 *Each history of $\mathsf{TwoStepLinSpec}$ is the concatenation of triples of events of the form $\mathsf{op}_1^{i_1}(x_1){:}()$, $\mathsf{op}_2^{i_2}(x_2){:}y_2$, $\overline{\mathsf{op}}_1^{i_1}(){:}y_1$ such that $\mathsf{SyncSpec}$ has a corresponding legal history of events $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$, and vice versa.*

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

Proposition 1.1 *Let $\mathsf{SyncObj}$ be a synchronisation object, $\mathsf{SyncSpec}$ be a synchronisation specification object, and let $\mathsf{TwoStepLinSpec}$ be built from $\mathsf{SyncSpec}$ as above. Then $\mathsf{SyncObj}$ is two-step linearisable with respect to $\mathsf{TwoStepLinSpec}$ if and only if it is synchronisation linearisable with respect to $\mathsf{SyncSpec}$.*

Proof: (⇒). Let h be a concurrent history of $\mathsf{SyncObj}$. By assumption, there is an extension h' of h, and a legal history h_s of $\mathsf{TwoStepLinSpec}$ such that $h'' = \mathit{complete}(h')$ and h_s are two-step compatible. Build a history h'_s of $\mathsf{SyncSpec}$ by replacing 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 h_s by the event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$. The history h'_s is legal by Lemma 1. 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'' and h_s are synchronisation compatible, so h'' is synchronisation lineariable, and so h is synchronisation linearisable.

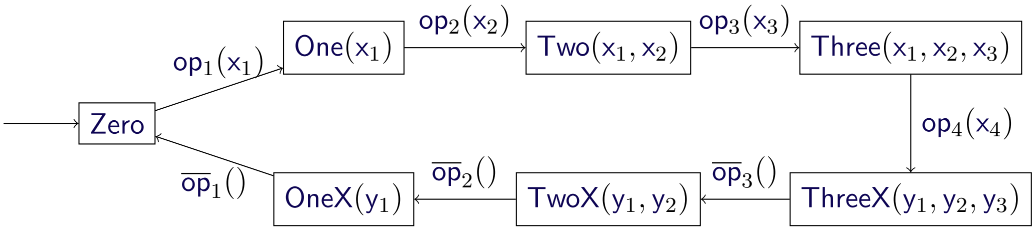
(⇐). Let h be a complete history of $\mathsf{SyncObj}$. By assumption, there is an extension h' of h, and a legal history h_s of $\mathsf{SyncSpec}$ such that $h'' = \mathit{complete}(h')$ and h_s are synchronisation compatible. Build a history h'_s of $\mathsf{TwoStepLinSpec}$ by replacing each event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ in h_s by the three 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 by Lemma 1. 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 $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$; and each $\mathsf{op}_2^{i_2}(x_2){:}y_2$ is between $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$. Hence h'' and h_s are two-step compatible, so h'' is two-step lineariable, and so h is two-step linearisable. □

The two-step linearisation specification object can often be significantly simplified from the template definition above. Here is such a specification object for a synchronous channel.

```
object SyncChanTwoStepLinSpec{
  private var state = 0          // Takes values 0, 1, 2, cyclically
  private var value: A = _       // The current value being sent
  def send(x: A): Unit = { require(state == 0); value = x; state = 1 }
  def receive(u: Unit): A = { require(state == 1); state = 2; value }
  def s̅e̅n̅d̅(): Unit = { require(state == 2); state = 0 }
}
```

3.3 Variations

The results of this section carry across to non-binary synchronisations, in a straightforward way. For a synchronisation object with k operations, op_1, …, op_k, the operations op_1, …, op_{k-1} are each linearised in two steps. The two-step linearisation specification object has $2k-1$ operations, op_1, …, op_k, $\overline{\mathsf{op}}_1$, …, $\overline{\mathsf{op}}_{k-1}$. The specification object encodes an automaton with $2k-1$ states. The figure below gives the automaton in the case $k = 4$.



The middle operation, op_4 in the above figure, applies the sync method 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 the state, and returns y_k.

4 Model checking for synchronisation linearisation

In this section we describe how to analyse a synchronisation object using model checking, to gain assurance that it satisfies synchronisation linearisation. We present our approach within the framework of the process algebra CSP [?] and its model checker FDR [?, ?]. We assume some familiarity with the syntax of CSP.

In particular, we use checks within the traces model of CSP. This model represents a process P by its traces, denoted $traces(P)$, i.e. the finite sequences of visible events that P can perform. Given processes P and Q, FDR can test whether $traces(P) \subseteq traces(Q)$. Here P is typically a model of some system that we want to analyse, and Q is a specification process that has precisely the traces that correspond to the desired property.

$\boxed{\text{Limitations}}$ of model checking.

We describe how to test for synchronisation linearisation within this framework. We start with the case of heterogeneous binary synchronisations; we describe how to generalise at the end of this section.

We build a CSP model of the synchronisation object. Such modelling is well understood, so we don't elaborate in detail. Typically CSP processes representing threads perform events to read or write shared variables, acquire or release locks, etc. The shared variables, locks, etc., are also represented by CSP processes. An example for a synchronous channel can be found in [**?**].

We assume that the model includes the following events:

- call.$t.op.x$ to represent thread t calling operation op with parameter x;
- return.$t.op.y$ to represent thread t returning from operation op with result y.

We assume that all other events, describing the internal operation of the synchronisation object, are hidden, i.e. converted into internal events.

We now describe how to test whether the model satisfies synchronisation linearisation with respect to a specification object. We build a process SyncSpec corresponding to the specification object. We assume this process uses events of the form sync.$t_1.t_2.x_1.x_2.y_1.y_2$ to represent a synchronisation between threads t_1 and t_2, calling $op_1(x_1)$ and $op_2(x_2)$, and receiving results y_1 and y_2, respectively. For example, for the synchronous channel, we would have

```
SyncSpec = sync?t1?t2?x?u!u!x → SyncSpec
```

If the synchronisation object or specification object has unbounded state, we have no chance of modelling it using finite-state model checking. However, we can often build approximations. For example, we could approximate (in an informal sense) the synchronous channel with sequence counter by one where the sequence counter is stored mod 5. Then the specification object can be modelled by

```
SyncSpec = SyncSpec'(1)
SyncSpec'(ctr) = sync?t1?t2?x?u!ctr!(x,ctr) → SyncSpec'((ctr+1)%5)
```

We then build a *lineariser* process for each thread as follows.

$$
\begin{aligned}
&\mathsf{Lineariser\ (t)}\ = \\
&\quad \mathsf{call\ .t.op_1\,?x_1} \to \mathsf{sync.t\,?t_2\,!x_1\,?x_2\,?y_1\,?y_2} \to \mathsf{return.t.op_1.y_1} \to \mathsf{Lineariser(t)} \\
&\quad \Box \\
&\quad \mathsf{call\ .t.op_2\,?x_2} \to \mathsf{sync\,?t_1\,!t\,?x_1\,!x_2\,?y_1\,?y_2} \to \mathsf{return.t.op_2.y_2} \to \mathsf{Lineariser(t)} \\
&\mathsf{alpha(t)}\ = \{\!|\ \mathsf{call.t,\ \ return.t,\ \ sync.t.t_1,\ \ sync.t_1.t}\ \mid\ \mathsf{t_1} \leftarrow \mathsf{ThreadID},\ \mathsf{t_1} \neq \mathsf{t}\ |\!\}
\end{aligned}
$$

This ensures that between each call and return event of t, there is a corresponding sync event.

We then combine together the specification process with the linearisers, synchronising on shared events: this means that each $\mathsf{sync.t_1.t_2}$ event will be a three-way synchronisation between $\mathsf{SyncSpec}$, $\mathsf{Lineariser(t_1)}$ and $\mathsf{Lineariser(t_2)}$.

$$\mathsf{Spec_0} = \mathsf{SyncSpec}\ [|\ \{\!|\ \mathsf{sync}\ |\!\}\ |]\ (\|\ \mathsf{t} \leftarrow \mathsf{ThreadID} \bullet [\mathsf{alpha(t)}]\ \ \mathsf{Lineariser\ (t)})$$

Every trace will represent an interleaving between a possible history of the concurrent object and a legal history of the specification object. Finally, we hide the sync events.

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

Each trace of the resulting process represents a history for which there is a compatible legal history of the specification object; i.e. it has precisely the traces that correspond to histories that are synchronisation linearisable. It is therefore enough to test whether the traces of the model of the synchronisation object are a subset of the traces of Spec, which can be discharged using FDR.

We now generalise this approach. For a synchronisation involving k threads, the corresponding sync event contains k thread identities, k parameters, and k return values; each such event will be a synchronisation (in the CSP model) between k threads and the specification process.

For homogeneous synchronisations the identities of the threads (and corresponding parameters and return values) may appear in either order within the sync events. The following definition of the lineariser allows this.

$$
\begin{aligned}
&\mathsf{Lineariser\ (t)}\ = \\
&\quad \mathsf{call\ .t.op\,?x} \to (\\
&\qquad \mathsf{sync.t\,?t'\,!\,x\,?x'\,?y\,?y'} \to \mathsf{return.t.op.y} \to \mathsf{Lineariser(t)} \\
&\qquad \Box \\
&\qquad \mathsf{sync\,?t'\,!\,t\,?x'\,!\,x\,?y'\,!\,y} \to \mathsf{return.t.op.y} \to \mathsf{Lineariser(t)} \\
&\quad)
\end{aligned}
$$

Finally, for synchronisation objects with multiple synchronisation modes, the specification process should have a different branch (with different sync events) for each mode.

5 Testing algorithms

5.1 Linearisability testing

Most of the techniques that we describe for testing synchronisation linearisation are influenced by the techniques for testing (standard) linearisation testing [?], so we begin by sketching those techniques.

The idea of linearisability testing is as follows. We run several 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.

Once all threads have finished, we can use an algorithm to test whether the history is linearisable with respect to the specification object. Informally, the algorithm searches for an order to linearise the invocations, consistent with what is recorded in the log, and such that the order represents a legal history of the specification object. See [?] for details of the algorithms.

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 [?], which we believe had not previously been documented. Further, the technique is easy to use: we have taught it in our undergraduate Concurrent Programming course at Oxford, and students have used it effectively.

Note that 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 threads in a way that all invocations will eventually return. 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.

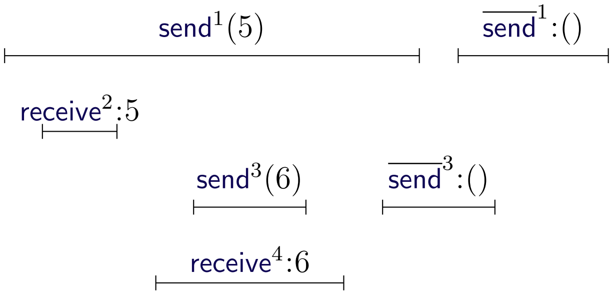
Another thing to note is that there is potentially a delay between a thread writing the call event into the log and actually calling the operation; and likewise there is potentially a delay between the operation returning and the thread writing the return event into the log. However, these delays do 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.

5.2 Hacking the linearisablity framework

In this section we investigate how to use the existing linearisation testing framework for testing synchronisation linearisation, using the ideas of Section 3.2. 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.

It turns out that we cannot use linearisability testing directly with the specification object TwoLinSpec from Section 3.2, because it gives false errors caused by delays in writing to the log. We describe in more detail how such testing might be done, and then explain the cause of the false errors. A thread that performs the concrete operation $\mathsf{op}_1(x_1)$: (1) writes $\mathsf{call}.\mathsf{op}_1^i(x_1)$ into the log; (2) performs $\mathsf{op}_1(x_1)$ on the synchonisation object, obtaining result y_1; (3) writes $\mathsf{return}.\mathsf{op}_1^i{:}()$, $\mathsf{call}.\overline{\mathsf{op}}_1^i()$ and $\mathsf{return}.\overline{\mathsf{op}}_1^i{:}y_1$ into the log. A thread that performs operation op_2 acts as for standard linearisation testing. Once all threads have finished, we could use the existing algorithms for testing whether the history is linearisable with respect to TwoStepLinSpec.

This approach does not work, because it gives false errors. For example, the timeline below depicts a log that could be obtained from a correct synchronous channel using the above approach, where we treat send as op_1.

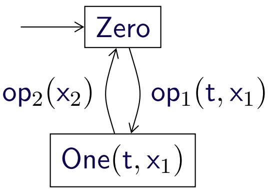


Here, the invocations in the top two rows synchronise to transmit 5, and then the invocations in the bottom two rows synchronise to transmit 6. However, the thread for the top row is slow to write its last three events into the log. The above history is not linearisable with respect to TwoStepLinSpec: it is clear that $\mathsf{send}^1(5)$ and $\mathsf{receive}^2{:}5$ would need to be linearised first; but this would require $\overline{\mathsf{send}}^1$ to be linearised before $\mathsf{send}^3(6)$, which is inconsistent with the history. Hence the approach would generate a false error.

Instead we use a technique that is robust against delays in logging. We start by considering the binary heterogeneous case, i.e. each synchronisation is between precisely two threads, which have called different operations. We assume that each thread has an identity in some range $[0 \mathinner{..} \mathsf{NumThreads})$. We arrange for this identity to be included in the call events written to the log for operations op_1 and $\overline{\mathsf{op}}_1$, but otherwise threads act as above; in particular, for each thread, calls to op_1 and $\overline{\mathsf{op}}_1$ alternate.

We then test whether the history is linearisable with respect to the specification object below. This object requires that corresponding invocations of op_1 and op_2 are linearised consecutively: it encodes the automaton on the right. However, it allows the corresponding $\overline{\mathsf{op}}_1$ to be linearised later (but before the next operation invocation by the same thread). It uses an array returns, indexed by thread identities, to record values that should be returned by a $\overline{\mathsf{op}}_1$ operation.

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



```
object TwoStepDelayedLinSpec{
  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.isInstanceOf[Zero]); 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 $\overline{op}_1$(t: ThreadID): B1 = {
    require(returns(t).isInstanceOf[Some]); val Some(y1) = returns(t)
    returns(t) = None; y1
  }
}
```

The following lemma identifies important properties of TwoStepDelayedLinSpec. It follows immediately from the definition.

Lemma 2 *Within any legal history of* $\mathsf{TwoStepDelayedLinSpec}$*, events* op_1 *and* 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* op_2 *makes a call* $\mathsf{SyncSpec.sync}(x_1, x_2)$ *obtaining result* (y_1, y_2)*. Under the assumptions about threads within the test, the next event for thread* t *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* $\mathsf{SyncSpec}$*.*

Conversely, each history with events ordered in this way will be a legal history of $\mathsf{TwoStepDelayedLinSpec}$ *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* $\mathsf{SyncSpec}$*.*

In order to argue for correctness, we need to distinguish between:

- the *invocation history*, in terms of actual calls and returns of op_1 and op_2; and
- the corresponding *log history*, which might contain delays.

For clarity, we annotate events with "*inv*" or "*log*". The invocation history uses events of the form $\mathsf{call}^{inv}.\mathsf{op}_1^{i_1}(x_1)$, $\mathsf{return}^{inv}.\mathsf{op}_1^{i_1}{:}y_1$, $\mathsf{call}^{inv}.\mathsf{op}_2^{i_2}(x_2)$, and $\mathsf{return}^{inv}.\mathsf{op}_2^{i_2}{:}y_2$. The log history uses events of the form $\mathsf{call}^{log}.\mathsf{op}_1^{i_1}(t, x_1)$ (note the additional thread identity parameter), $\mathsf{return}^{log}.\mathsf{op}_1^{i_1}{:}()$ (note the unit return value), $\mathsf{call}^{log}.\overline{\mathsf{op}}_1^{i_1}(t)$, $\mathsf{return}^{log}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$ (note the transferred return value), $\mathsf{call}^{log}.\mathsf{op}_2^{i_2}(x_2)$, and $\mathsf{return}^{log}.\mathsf{op}_2^{i_2}{:}y_2$

We can consider the interleaving of the invocation and log histories, following real-time order. In the interleaving, for op_1 and op_2, call^{log} events will be earlier than the corresponding call^{inv} events; and return^{log} events will be later than the corresponding return^{inv} events. However, the two histories agree on the relative order of the op_1 and op_2 events of each individual thread.

The proposition below shows that this testing method does not generate any false errors.

Proposition 2.1 *Suppose synchronisation object* $\mathsf{SyncObj}$ *is linearisable with respect to* $\mathsf{SyncSpec}$*. Then each complete log history* h_l *of* $\mathsf{SyncObj}$ *is linearisable with respect to* $\mathsf{TwoStepDelayedLinSpec}$*.*

Proof: Consider a complete log history h_l of $\mathsf{SyncObj}$, and a corresponding invocation history h; and consider their real-time interleaving. By assumption, there is a legal history h_s of $\mathsf{SyncSpec}$ such that h and h_s are synchronisation compatible. Thus h_s may be interleaved with the interleaving of h and h_l, so that each sync event from h_s is between the corresponding call^{inv} and return^{inv} events from h, and hence also between the corresponding call^{log} and return^{log} events from h_l.

We build an interleaving of h, h_l and a legal history h'_s of $\mathsf{TwoStepDelayedLinSpec}$ from the interleaving of h, h_l and h_s, as follows.

1. We replace each $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ by the two (consecutive) events $\mathsf{op}_1^{i_1}(t, x_1){:}()$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$, where t is the identity of the thread that makes the corresponding call of op_1 in h_l.
2. We insert an event $\overline{\mathsf{op}}_1^{i_1}(t){:}y_1$ between every $\mathsf{call}^{log}.\overline{\mathsf{op}}_1^{i_1}(t)$ and $\mathsf{return}^{log}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$ (from h_l).

Note that each inserted event is between the corresponding call^{log} and return^{log} events, by construction. Let h'_s be these inserted events; we show that h'_s is a legal history of $\mathsf{TwoStepDelayedLinSpec}$.

- The events inserted in step 1 alternate between op_1 and op_2, as required by $\mathsf{TwoStepDelayedLinSpec}$. Further, they are in the same order, and have the same values for x_1, x_2, y_2, i_1 and i_2 as the corresponding sync events from h_s. Hence each inserted op_2 event has the return value y_2 as required by $\mathsf{TwoStepDelayedLinSpec}$. Further, the value for t matches that in the corresponding $\mathsf{call}^{log}.\mathsf{op}_1^{i_1}(t, x_1)$ event; so the value y_1 written into $\mathsf{returns}(t)$ (by op_2 in $\mathsf{TwoStepDelayedLinSpec}$) matches the value returned by the corresponding call to sync.
- For the events from step 2, because of the way logging is done, each value of y_1 returned by $\overline{\mathsf{op}}_1$ must match the value returned by the previous invocation of op_1 on the synchronisation object by thread t. Since h is synchronisation linearisable, this y_1 must match the value returned by the corresponding call of sync. And this matches the last value written into $\mathsf{returns}(t)$ (by the previous item), as required by $\mathsf{TwoStepDelayedLinSpec}$.

This demonstrates that h_l is linearisable with respect to $\mathsf{TwoStepDelayedLinSpec}$. □

In theory, the delays in logging can mean that an invocation history that is not synchronisation linearisable is transformed into a log history that is linearisable with respect to $\mathsf{TwoStepDelayedLinSpec}$ (although this seems unlikely). We show that if the invocation history is not synchronisation linearisable with respect to $\mathsf{SyncSpec}$, then there is a corresponding log history that is not linearisable with respect to $\mathsf{TwoStepDelayedLinSpec}$.

Proposition 2.2 *Let h be a history of $\mathsf{SyncObj}$ that is not synchronisation linearisable with respect to $\mathsf{SyncObj}$. Then there is a corresponding log history h_l that is not linearisable with respect to $\mathsf{TwoStepDelayedLinSpec}$.*

Proof: Let h be as in the statement of the proposition. We build a corresponding log history h_l, and interleave it with h as follows. The construction

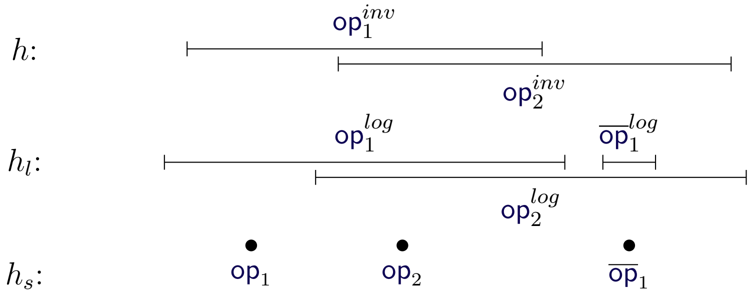


Figure 3: Representation of the construction in the proof of Proposition 2.2. calls of op_1 and op_2 might be in either order, as might their returns. The proof shows that both calls precede both returns.

is illustrated in Figure 3.

- We add an event $\mathsf{call}^{log}.\mathsf{op}_1^{i_1}(t, x_1)$ immediately before each $\mathsf{call}^{inv}.\mathsf{op}_1^{i_1}(x_1)$, where t is the identity of the thread making the call.
- We add events $\mathsf{return}^{log}.\mathsf{op}_1^{i_1}{:}()$, $\mathsf{call}^{log}.\overline{\mathsf{op}}_1^{i_1}(t)$, and $\mathsf{return}^{log}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$ immediately after each $\mathsf{return}^{inv}.\mathsf{op}_1^{i_1}{:}y_1$, where again t is the identity of the relevant thread.
- We add an event $\mathsf{call}^{log}.\mathsf{op}_2^{i_2}(x_2)$ immediately before each $\mathsf{call}^{inv}.\mathsf{op}_2^{i_2}(x_2)$.
- We add an event $\mathsf{return}^{log}.\mathsf{op}_2^{i_2}{:}y_2$ immediately after each $\mathsf{return}^{inv}.\mathsf{op}_2^{i_2}{:}y_2$.

Let h_l be these inserted events. We need to show that h_l is not linearisable with respect to $\mathsf{TwoStepDelayedLinSpec}$. We argue by contradiction: we suppose that h_l *is* linearisable, and deduce that h is synchronisation linearisable with respect to $\mathsf{SyncSpec}$.

So suppose h_l is linearisable, and let h_s be the corresponding legal history of $\mathsf{TwoStepDelayedLinSpec}$. Let $\mathsf{op}_1^{i_1}(t, x_1){:}y_1$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ be two consecutive events in h_s. Then:

1. In h_l, the event $\mathsf{call}^{log}.\mathsf{op}_1^{i_1}(t, x_1)$ is before $\mathsf{return}^{log}.\mathsf{op}_2^{i_2}{:}y_2$ (to satisfy linearisation). So in h, the event $\mathsf{call}^{inv}.\mathsf{op}_1^{i_1}(t, x_1)$ is before $\mathsf{return}^{inv}.\mathsf{op}_2^{i_2}{:}y_2$, by the way we have constructed h_l.
2. In h_s, the event $\mathsf{op}_2^{i_2}(x_2){:}y_2$ must be before $\overline{\mathsf{op}}_1^{i_1}(t){:}y_1$ (by Lemma 2). So in h_l, the event $\mathsf{call}^{log}.\mathsf{op}_2^{i_2}(x_2)$ must be before $\mathsf{return}^{log}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$ (to satisfy linearisation). Hence, by the way we have constructed h_l, the event $\mathsf{call}^{inv}.\mathsf{op}_2^{i_2}(x_2)$ is before $\mathsf{return}^{inv}.\mathsf{op}_1^{i_1}{:}y_1$.

To summarise the above two points: the invocations of $\mathsf{op}_1^{i_1}$ and $op_2^{i_2}$ overlap in h.

Further, consider two such pairs of invocations, $(\mathsf{op}_1^{i_1}, op_2^{i_2})$ and $(\mathsf{op}_1^{i'_1}, op_2^{i'_2})$. If the overlap in h of the first pair is before the overlap of the second pair, then the same is true in h_l (again by the way we have constructed h_l), so the corresponding events in h_s must be ordered in the same way. Informally, the order of the overlaps is consistent with h_s.

Let h'_s be the legal history of SyncSpec corresponding to h_s, implied by Lemma 2. Given the interleaving of h and h_l, and the interleaving of h_l and h_s, we build an interleaving of h and h'_s. We insert each event $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ from h'_s during the overlap between the corresponding invocations in h, and respecting the order of h_s and h'_s. This is possible by the above results.

Hence we have shown that h is synchronisation compatible with h'_s, and so is linearisable with respect to SyncSpec, as required. □

The above proposition included the possibility of an invocation history h of SyncObj not being synchronisation linearisable, but a corresponding log history h_l being linearisable with respect to TwoStepDelayedLinSpec, i.e. the incorrect history not being detected. Let us examine this possibility in more detail. So suppose h is not synchronistion linearisable, but h_l is linearisable. Let h_s be the corresponding legal history of TwoStepDelayedLinSpec, and let h'_s be the corresponding legal history of SyncSpec implied by Lemma 2. The fact that h is not synchronisation linearisable means that h and h'_s are not synchronisation compatible. .... Actually, there are lots of different ways that might happen, so it's not clear this discussion is going to be useful.

$\boxed{\text{Generalisations}}$: non-binary and/or non-homogeneous.

5.3 Case with state

Suppose the specification object has non-trivial state.

I think it will be more efficient to give a more direct implementation. Define a configuration to be: (1) a point in the log reached so far; (2) the set of pending operation invocations that have not synchronised; (3) the set of pending operation invocations that have synchronised (but not returned); and (4) the state of the sequential synchronisation object. In any configuration, can: synchronise a pair of pending operations (and update the synchronisation object); advance in the log if the next event is a return that is not pending; or advance in the log if the next event is a call. Then perform DFS.

Partial order reduction: a synchronisation point must follow either the call of one of the concurrent operations, or another synchronisation point.

Any synchronisation history can be transformed into this form, by moving synchronisation points earlier, but not before any of the corresponding call events, and preserving the order of synchronisations. This means that after advancing past the call of an invocation, we may synchronise that invocation, and then an arbitrary sequence of other invocations.

Alternatively, a synchronisation point must precede either the return of one of the concurrent operations, or another synchronisation point. This is more like the JIT technique in the linearisability testing paper. This means that before advancing in the log to the return of an invocation that has not synchronised, we synchronise some invocations, ending with the one in question. And we only synchronise in these circumstances.

My intuition is that the former is more efficient: in the latter, we might investigate synchronising other invocations even though the returning operation can't be synchronised with any invocation.

Complexity

Consider the problem of testing whether a given concurrent history has synchronisations consistent with a given sequential specification object.

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

Let ConcVariable be a concurrent object that represents a variable.

We consider concurrent synchronisation histories on an object with the following signature.

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

The intention is that op$_1$("get", x) acts like get(x), and op$_1$("set", x) acts like set(x) (but returns -1). The op$_2$ invocations do nothing except synchronise. This can be captured formally by the following synchronisation specification object.

```
object VariableSyncSpec{
  private var state = 0
  def sync((op, x): (String, Int), u: Unit): (Int, Unit) =
    if(op == "get") (state, ()) else{ state = x; (−1, ()) }
}
```

Let ConcVariable be a concurrent object that represents a variable. Given a concurrent history h of ConcVariable, we build a concurrent history h' of VaraibleSync as follows. We replace every call or return of get(x) by (respectively) a call or return of 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 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.

5.4 Stateless case

In the stateless case, a completely different algorithm is possible. Define two invocations 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 invocations of each operation (so a history of length $4n$), this can be calculated in $O(n^2)$. Then find if there is a total matching in the corresponding bipartite graph, using the Ford-Fulkerson method, which is $O(n^2)$.

6 Variations

Note: this section needs a complete re-write. Most needs to be moved into the testing section.

We've implicitly assumed that the operations op$_1$ and op$_2$ are distinct. I don't think there's any need for this. Example: exchanger.

Most definitions and results go through to the case of $k > 2$ invocations synchronising. Examples: ABC problem; barrier synchronisation. To capture the relationship with linearisation, we require $k - 1$ operations to be linearised by two operations of the specification object. Maybe give automaton for $k = 4$.

It turns out that for $k > 2$, 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 **??**.

Definition 2.1 *The problem of finding a complete matching in a 3-partite hypergraph is as follows: given 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 T.*

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 complete matching exists. The synchronisations are between operations as follows:

```
def op$_1$(x: X): Unit
def op$_2$(y: Y): Unit
def op$_3$(z: Z): Unit
```

The synchronisations are specified by:

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

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 invocations (in any order). It is clear that any synchronisation linearisation corresponds to a complete matching, i.e. the invocations that synchronise correspond to the complete matching U.

6.1 Different modes of synchronisation

Some synchronisation objects allow different modes of synchronisation. For example, consider a synchronous channel with timeouts: each invocation might synchronise with another invocation, or might timeout without synchronisation. Such a channel might have a signature as follows.

```
class TimeoutChannel{
  def send(x: A): Boolean
  def receive(u: Unit): 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 (the type Some[A] contains the union of such values). The possible synchronisations can be captured by the following specification object.

```
object TimeoutSpec{
  def sync$_{s,r}$(x: A, u: Unit): (Boolean, Option[A]) = (true, Some(x))
  def sync$_s$(x: A): Boolean = false
  def sync$_r$(u: Unit): Option[A] = None
}
```

The operation $\mathsf{sync}_{s,r}$ corresponds to where a send and receive synchronise, as previously. The operations sync_s and sync_r correspond, respectively, to where a send or receive fails to synchronise.

More generally, the specification object can have any number of operations of the form

```
def sync_{j_1,...,j_m}(x_1: A_1, ..., x_m: A_m): (B_1, ..., B_m)
```

This corresponds to the case of a synchronisation between the m invocations $\mathsf{op}_{j_1}(\mathsf{x}_1), \ldots, \mathsf{op}_{j_m}(\mathsf{x}_m)$. The formal definition is an obvious adaptation of the previous version: in the interleaved history, between the call and return of each $\mathsf{op}_j(\mathsf{x}) : \mathsf{y}$, there must be a corresponding $\mathsf{sync}_{j_1,\ldots,j_m}(\mathsf{x}_1, \ldots \mathsf{x}_m) : (\mathsf{y}_1, \ldots, \mathsf{y}_m)$ event, i.e. for some i, $j = j_i$, $\mathsf{x} = \mathsf{x}_i$, and $\mathsf{y} = \mathsf{y}_i$.

*** Can we capture the bounded buffer example in this framework?