Understanding Synchronisation

Jonathan Lawrence and Gavin Lowe

June 6, 2023

Abstract

$\boxed{\cdots}$

1 Introduction

In many concurrent programs, it is necessary at some point for two or more threads to *synchronise*: each thread waits until other relevant threads have reached the synchronisation point before continuing; in addition, the threads can exchange data. Reasoning about programs can be easier when synchronisations are used: it helps us to reason about the states that different threads are in.

We study synchronisations in this paper: we formalise the requirements of synchronisations, and describe testing and analysis techniques.

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[A]{
  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.

[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 ().

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

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

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, a *barrier synchronisation* object of the following class

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

can be used to synchronise `n` threads: each thread calls `sync`, and no invocation returns until all `n` have called it. We say that the synchronisation has *arity n*.

A *combining barrier*, 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(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).

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

[2]The Scala type `(A,A) => A` represents functions from pairs of `A` to `A`.

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[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 invocation of each operation may or may not synchronise with an invocation of the other operation. Equivalently, unsuccessful instances of send and receive can be considered *unary* synchronisations.

Some implementations of synchronous channels allow the channel to be closed, say by an operation `close` (which does not synchronise with another invocation). 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 mixed modes of synchronisation, either successful or throwing an exception.

A *termination-detecting queue* can also be thought of as a stateful synchronisation object with multiple modes. Such an object 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 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[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 termination-detecting queue combines the functionality of a concurrent datatype and a synchronisation object.

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

In this paper, we consider what it means for one of these synchronisation objects to be correct. We also present 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 it talks about synchronisations between invocations, rather than single invocations: we call the property *synchronisation linearisation*. We also describe a corresponding progress property.

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 an operation of the synchronisation object may correspond to *two* operations of the object used to specify linearisation.

We then consider testing of synchronisation object implementations. Our techniques are based on the techniques for testing (standard) linearisation [Low16], which we sketch in Section 4: the basic idea is to record a history of threads using the object, and then to test whether that history is linearisable.

In Section 5 we show how the technique can be adapted to test for synchronisation linearisation, using the result of Section 3. Then in Section 6 we show how synchronisation linearisation can be tested more directly, and present various complexity results. $\boxed{\text{More here.}}$

In Section 8 we consider how the property of synchronisation linearisation can be analysed via model checking. $\boxed{\text{Cut this?}}$

$\boxed{\text{Contributions}}$

2 Specifying synchronisations

In this section we describe how synchronisations can be formally specified. For ease of exposition, we start by considering *heterogeneous binary* synchronisation in this section, i.e. where every synchronisation is between *two* invocations of *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 $k \neq 1$ 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, $\mathsf{state: S}$. 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 invocations 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 $\mathsf{op}_1(\mathsf{x}_1)$ and $\mathsf{op}_2(\mathsf{x}_2)$ synchronise, then the results y_1 and y_2 of the invocations are such that $\mathsf{sync}(\mathsf{x}_1, \mathsf{x}_2)$ could return the pair $(\mathsf{y}_1, \mathsf{y}_2)$. 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 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 invocations. The $\mathsf{require}$ 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 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 specification object with operation

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

The require clause specifies that 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[A]{
  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* [HW90], 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.

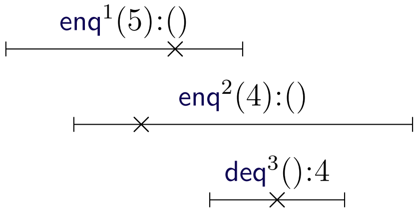


Figure 1: Timeline representing the linearisation example.

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.

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.

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

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

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. We say that h and h_s *correspond* if they contain 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 this order is consistent with the specification object.

Continuing the running example, the histories h and h_s are compatible, 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.

We say that a complete history h is *linearisable* with respect to *Spec* if there is a corresponding legal 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 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 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.

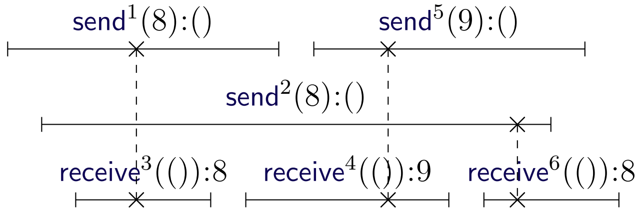


Figure 2: Timeline representing the synchronisation example.

2.2 Synchronisation linearisability

We now adapt the definition of linearisability to synchronisations. We consider a concurrent history of the synchronisation object *Sync*. The history contains call and return events, as in the previous subsection; in the case of binary synchronisation, the events correspond 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:

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

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 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. Informally, an event $\mathsf{sync}^{i_1,i_2}(x_1,x_2){:}(y_1,y_2)$ corresponds to a synchronisation between invocations $\mathsf{op}_1^{i_1}(x_1){:}y_1$ 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 specification 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 dashed 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 shown by the interleaving 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 $\mathsf{op}_j(\mathsf{x}_j\colon \mathsf{A}_j)\colon \mathsf{B}_j$ for $j = 1, \ldots, k$,

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

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

```
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 $j = 1,\ldots,k$. Note that the parameters of the k invocations could be passed to sync in $k!$ different orders (although in this case, if f is associative and commutative, the order makes no difference). 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[A]{
  def sync_s(x: A) = false
  def sync_r(u: Unit) = None
  def sync_s,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: in particular sync_s must occur between the call and return of send, and likewise for sync_r.

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

```
class ClosableChannelSpec[A]{
  private var isClosed = false
  def close(u: Unit) = { isClosed = true; () }
  def sync(x: A, u: Unit) = { require(!isClosed); ((), x)
  def sendFail(x: A) = { require(isClosed); throw new Closed }
  def receiveFail(u: Unit) = { require(isClosed); throw new Closed }
}
```

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`.

2.4 Specifying progress

We now consider a progress condition for synchronisation objects.

We assume that each pending invocation is scheduled infinitely often, unless it is blocked (for example, trying to obtain a lock). Under this assumption, our progress condition can be stated informally as:

- If a set of pending invocations can synchronise, then some such set should eventually synchronise;
- Once a particular invocation has synchronised, it should eventually return.

Note that there might be several different synchronisations possible. For example, consider a synchronous channel, and suppose there are pending calls to `send(3)`, `send(4)` 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 progress condition allows *either* synchronisation to occur.

Our progress condition allows all pending invocations to block if no synchronisation is possible. For example, if every pending invocations on a synchronous channel is a `send`, then clearly none can return.

Note that our progress condition is somewhat different from the condition of *lock freedom* for concurrent datatypes [HS12]. That condition requires that, assuming invocations collectively are scheduled infinitely often, then eventually some invocation returns. Lock freedom makes no assumption about scheduling being fair. For example, if a particular invocation holds a lock then lock freedom allows the scheduler to never schedule that invocation; in most cases, this will mean that no invocation returns: any implementation that uses a lock in a non-trivial way is not lock-free.

By contrast, our assumption, that each unblocked pending invocation is scheduled infinitely often, reflects that modern schedulers *are* fair, and do not starve any single invocation. For example, if an invocation holds a lock, and is not in a potentially unbounded loop (or permanently blocked trying to obtain a second lock), then it will be scheduled sufficiently often, and so will eventually release the lock. Thus our progress condition can be satisfied by an implementation that uses locks. However, our assumption does allow invocations to be scheduled in an unfortunate order (as long as each is scheduled infinitely often), which may cause the progress condition to fail.

The following definitions make this notion precise.

Definition 1 We say that an infinite execution is *fair* if every invocation either returns or performs infinitely many steps.

Consider a synchronisation object in a particular state st. We say that the object can *eventually return* if (1) no execution leads to a deadlocked state, and (2) for every fair infinite execution from st that contains no call event, there is a return event.

The following definition describes the circumstances under which it is acceptable for an object to block, and so does not eventually return.

Definition 2 Let $Sync$ be a synchronisation object that is synchronisation-linearisable with respect to specification object $Spec$. Let h be a history of $Sync$. We say that $Sync$ *may block* after h if there is a legal history h_s of $Spec$, such that:

- $complete(h)$ and h_s are compatible; and
- There is no proper extension h_e of h (adding one or more return events, but no call events) and extension h'_s of h_s such that h'_s is a legal history of $Spec$, and $complete(h_e)$ and h'_s are compatible.

The first condition says that for each synchronisation in h_s, there is a corresponding return event in h: there is no invocation that has synchronised but not yet returned. The second condition says that no more synchronisations are possible: such a synchronisation would correspond to a synchronisation event $sync$.

We give two examples, both for a synchronous channel.

Example 3 Consider $h = \langle \mathsf{call.send}^1(3), \mathsf{call.receive}^2(), \mathsf{return.receive}^2{:}3\rangle$. There is no history h_s of $Spec$ such that $complete(h) = \langle \mathsf{call.receive}^2(), \mathsf{return.receive}^2{:}3\rangle$ and h_s are compatible. (However, the extension $h_e = h \frown \langle \mathsf{return.send}^1{:}()\rangle$ is compatible with $h_s = \langle \mathsf{sync}^{1,2}(3, ()){:}((), 3)\rangle$). Informally, the channel may not block after h because a synchronisation has occurred, and so there is a pending return of the send invocation.

Example 4 Now consider $h = \langle \mathsf{call.send}^1(3), \mathsf{call.receive}^2()\rangle$. We have that $complete(h) = \langle\rangle$ is compatible with the history $h_s = \langle\rangle$ of $Spec$ (and no other). But the extension $h_e = h \frown \langle \mathsf{return.send}^1{:}(), \mathsf{return.receive}^2{:}3\rangle$ is compatible with the extension $h'_s = \langle \mathsf{sync}^{1,2}(3, ()){:}((), 3)\rangle$ of h_s. Hence the channel may not block after h. Informally, the two pending invocations can synchronise and then return.

We now give an example where blocking is allowed.

Example 5 Let $h = \langle \mathsf{call.send}^1(3) \rangle$. Then *complete*$(h) = \langle\rangle$ is compatible with the history $h_s = \langle\rangle$ of *Spec*. But the only proper extension of h is $h_e = \langle \mathsf{call.send}^1(3), \mathsf{return.send}^1{:}() \rangle$, and no history of *Spec* is compatible with *complete*$(h_e) = h_e$.

Definition 6 Let *Sync* be a synchronisation object that is synchronisation-linearisable with respect to specification object *Spec*. We say that *Sync* is *progressable* if for every history h, if it is not the case that *Sync* may block after h, then *Sync* can eventually return from each state reached after h.

Relate testing to progress.

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 synchronise, so each operation of the synchronisation specification object corresponds to a single operation of the concurrent object. Put another way: standard linearisation is an instance of synchronisation linearisation with just unary synchronisations.

However, linearisability and synchronisation linearisability are not equivalent in general: we show that, given a synchronisation linearisability specification object SyncSpec, it is not always possible to find a linearisability 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 linearisation with respect to linearisability specification Spec. Consider the history

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

This is synchronisation linearisable with respect to SyncChanSpec. By the assumption, it must also be linearisable with respect to Spec; so 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 $\mathsf{receive}$, i.e.

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

But the history

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

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

3.1 Two-step linearisability

We now show that binary heterogeneous synchronisation linearisability corresponds to a small adaptation of linearisability, where one of the operations on the synchronisation 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. We generalise to synchronisations of more than two threads, and to the homogeneous case in Section 3.3. In the definitions below, we describe just the differences from standard linearisation, to avoid repetition.

Given a synchronisation object with signature

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

we will consider a linearisability specification object with signature

```
class TwoStepLinSpec{
  def op1(x1: A1): Unit
  def op̅1(): B1
  def op2(x2: A2): B2
}
```

The idea is that the operation op_1 on $\mathsf{SyncObj}$ will be linearised by the composition of the two operations op_1 and $\overline{\mathsf{op}}_1$; but operation op_2 on $\mathsf{SyncObj}$ 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 legal history h_s of such a two-step specification object much as in Section 2.1, with the addition that for each event $\overline{\mathsf{op}}_1^i(){:}y$ in h_s, we require that there is an earlier event $\mathsf{op}_1^i(x){:}()$ 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 $\mathsf{call}.\mathsf{op}_1^i(x)$ and $\mathsf{return}.\mathsf{op}_1^i{:}y$ in h, h_s contains $\mathsf{op}_1^i(x){:}()$ and $\overline{\mathsf{op}}_1^i(){:}y$; and vice versa;

- For every $\mathsf{call}.\mathsf{op}_2^i(x)$ and $\mathsf{return}.\mathsf{op}_2^i{:}y$ in h, h_s contains $\mathsf{op}_2^i(x){:}y$; 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

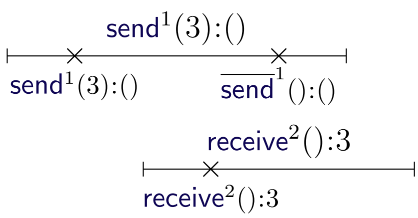
- Each $\mathsf{op}_1^i(x){:}()$ and $\overline{\mathsf{op}}_1^i(){:}y$ occur between $\mathsf{call}.\mathsf{op}_1^i(x)$ and $\mathsf{return}.\mathsf{op}_1^i{:}y$, in that order;

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

For example, consider a synchronous channel, with send corresponding to op_1, and $\mathsf{receive}$ 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.

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

This is represented by the following timeline, where the horizontal lines and the labels above represent the interval between the call and return events of the synchronisation object, and the "×"s and the labels below represent the corresponding operations of the specification object.



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

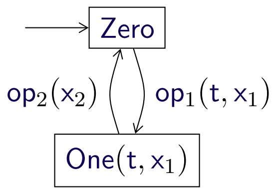
3.2 Proving the relationship

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 we choose is not the simplest possible, but it is convenient for the testing framework we use in Section 5.

The definition of TwoStepLinSpec is below. We assume that each thread has an identity in some range [0 .. NumThreads). For simplicity, we arrange for this identity to be included in the call events for operations op_1 and $\overline{op}_1$.

The object TwoStepLinSpec requires that corresponding invocations of op_1 and op_2 are linearised consecutively: it does this by encoding the automaton on the right. However, it allows the corresponding $\overline{op}_1$ 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 the value that should be returned by an $\overline{op}_1$ invocation by each thread. Each invocation of op_2 calls SyncSpec.sync to obtain the values that should be returned for synchronisation linearisation; it writes the value for the corresponding $\overline{op}_1$ into returns.

```
type ThreadID = Int                 // Thread identifiers
val NumThreads: ThreadID = ... // 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 o̅p̅1(t: ThreadID): B1 = {
    require(state == Zero && returns(t).isInstanceOf[Some[B1]])
    val Some(y1) = returns(t); returns(t) = None; y1
  }
}
```

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

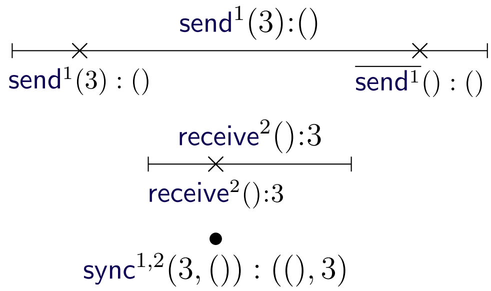
Lemma 7 Within any legal history of TwoStepLinSpec, events op_1 and op_2 alternate. Let $\mathsf{op}_1^{i_1}(t, x_1){:}()$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ be a consecutive pair of such events. Then op_2 makes a call $\mathsf{SyncSpec.sync}(x_1, x_2)$ obtaining result (y_1, y_2). The next event for 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 linearisability to two-step linearisability.

Proposition 8 Let SyncObj be a binary heterogeneous synchronisation object, SyncSpec a corresponding synchronisation specification object, and let TwoStepLinSpec be built from SyncSpec as above. Then 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. By assumption, there is an extension h' of h, and a legal history h_s of TwoStepLinSpec such that $h'' = \mathit{complete}(h')$ and h_s are two-step-compatible. 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 "×"s and the labels below; and h'_s is represented by the "•" and the label below.



The history h'_s is legal for SyncSpec by Lemma 7. 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 $\mathsf{call.op}_1^{i_1}(x_1)$ and $\mathsf{return.op}_1^{i_1}{:}y_1$, and between $\mathsf{call.op}_2^{i_2}(x_2)$ and $\mathsf{return.op}_2^{i_2}{:}y_2$. Hence h'' and h_s are synchronisation-compatible; so h'' is synchronisation-linearisable; and so h is synchronisation-linearisable.

($\Leftarrow$). Let h be a complete history of SyncObj. By assumption, there is an extension h' of h, and a legal history h_s of SyncSpec such that $h'' = \mathit{complete}(h')$ and h_s are synchronisation compatible. 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 7. 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 $\mathsf{call.op}_1^{i_1}(x_1)$ and $\mathsf{return.op}_1^{i_1}{:}y_1$; and each $\mathsf{op}_2^{i_2}(x_2){:}y_2$ is between $\mathsf{call.op}_2^{i_2}(x_2)$ and $\mathsf{return.op}_2^{i_2}{:}y_2$. Hence h'' and h_s are two-step-compatible; so h'' is two-step-linearisable; and so h is two-step-linearisable. $\square$

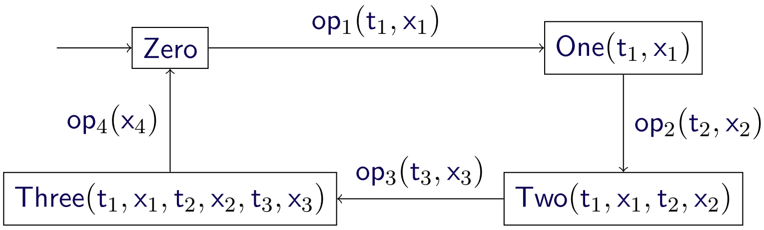
The two-step linearisation specification object can often be significantly simplified from the template definition above. Here is such a specification object for a synchronous channel.

```
object SyncChanTwoStepLinSpec{
  private var state = 0            // Takes values 0, 1, cyclically
  private var threadID = −1        // Current thread ID when state = 1
  private val canReturn =          // which senders can return?
    new Array[Boolean](NumThreads)
  private var value: A = _         // The current value being sent
  def send(t: ThreadID, x: A): Unit = {
    require(state == 0 && !canReturn(t)); value = x; threadID = t; state = 1 }
  def receive(u: Unit): A = {
    require(state == 1); canReturn(threadID) = true; state = 0; value }
  def send‾(t: ThreadID): Unit = {
    require(state == 0 && canReturn(t)); canReturn(t) = false }
}
```

3.3 Generalisations

The results of the previous subsections carry across to non-binary synchronisations, in a straightforward way. For a synchronisation object with k 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 op_i of the synchronisation object is linearised by the composition of 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 $k = 4$.



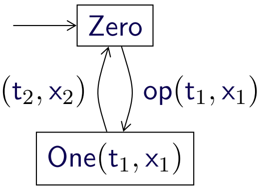
The final op operation, op_4 in the above figure, applies the sync method of the synchronisation specification object to the parameters $\mathsf{x}_1, \ldots, \mathsf{x}_k$ to obtain the results $\mathsf{y}_1, \ldots, \mathsf{y}_k$; it stores the first $k-1$ in appropriate $\mathsf{returns}_i$ arrays, and returns 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 invocations are handled similarly. Suppose we have a synchronisation object with a single operation **def** op(x: A): B. All invocations of op have to be treated similarly, so we associate *each* with two operations op and $\overline{\mathsf{op}}$ of the specification object. The specification object is below, and encodes the automaton on the right. The second invocation of op in any synchronisation (from the One state of the automaton) writes the results of the invocation into the returns array. Each invocation of $\overline{\mathsf{op}}$ returns the stored value.

```
class TwoStepLinSpec{
  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 o̅p̅(t: ThreadID): B = {
    require(state.isInstanceOf[Zero] && returns(t).isInstanceOf[Some])
    val Some(y) = returns(t); returns(t) = None; y
  }
}
```



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. Most of the techniques are influenced by the techniques for testing (standard) linearisation [Low16], 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 $\mathsf{call.op}^i(x)$ 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. Further, the logging associates each invocation with an invocation $\mathsf{op}(x)$ of the corresponding operation on the specification object.

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 corresponding invocations on the specification object. See [Low16] for details of several 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 [ST05], 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.

Note that this testing concentrates upon the safety property of linearisability, 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, 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 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 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. 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.

Recall, from the introduction of Section 3, that a straightforward approach won't work. Instead we adapt the idea of two-step linearisation from later in that section. We start by considering the case of binary heterogeneous synchronisation. We aim to obtain a log history that can be tested for

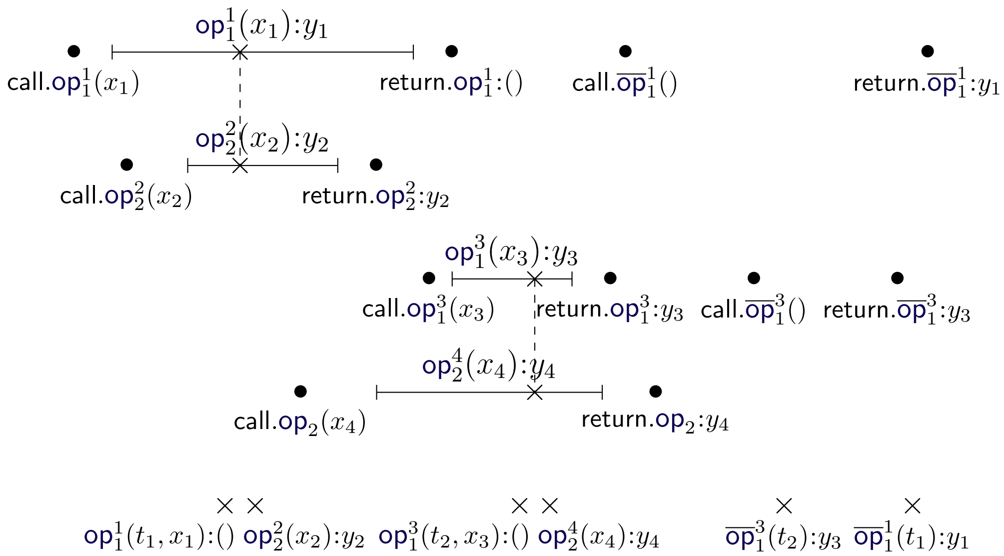


Figure 3: Illustration of a log for two-step synchronisation linearisation testing. Horizontal lines represent the operation calls themselves. Bullets (read from left to right) represent the log history. The crosses on operation calls, linked by dashed lines, represent the synchronisation points. The bottom row illustrates the history h_t of the two-step synchronisation object constructed in the proof of Lemma 9.

(standard) linearisability against $\mathsf{TwoStepLinSpec}$.

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

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

- A thread that performs operation op_2 acts as for standard linearisability testing.

Figure 3 illustrates a possible log. Note there might be delays involved in writing to the log. We refer to the *log history*, to distinguish it from the history of calls and returns on the synchronisation object.

As with standard linearisation, the tester needs to define the threads so that all invocations will eventually return, i.e. that each will be able to synchronise. For a binary 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).

Once all threads have finished, we test whether the log history is linearisable (i.e. standard linearisation) with respect to TwoStepLinSpec from Section 3. The following lemma shows that this approach does not find false errors.

Lemma 9 Suppose the synchronisation object is synchronisation-linearisable with respect to SyncSpec. Then each history obtained by the above process is linearisable with respect to TwoStepLinSpec.

Proof: Let h by a history of actual calls and returns, and let h_l be a corresponding log history. By assumption, h is synchronisation-linearisable, so let h_s be the history of SyncSpec that is synchronisation-compatible with h. Consider the interleaving of all three: i.e. h and h_l interleaved corresponding to temporal ordering; and h and h_s interleaved as required for synchronisation compatibility. Figure 3 gives an example.

We construct a history h_t of TwoStepLinSpec such that h_l and h_t are compatible. We define h_t by interleaving it with the previous histories as follows.

- For each synchronisation point $s = \mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ in h_s, we add an event $\mathsf{op}_1^{i_1}(t, x_1){:}()$ immediately before s, and an event $\mathsf{op}_2^{2}(x_2){:}y_2$ immediately after s (i.e. such that there is no other event between these two events).
- For each pair of events $\mathsf{call}.\overline{\mathsf{op}}_1^{i_1}()$ and $\mathsf{return}.\overline{\mathsf{op}}_1^{i_1}{:}y_1$, we insert an event $\overline{\mathsf{op}}_1^{i_1}(t){:}y_1$ at an arbitrary place in between (but not between a pair of events inserted under the previous item).

The bottom row of Figure 3 illustrates this construction.

The histories h_l and h_t are compatible by construction: in the above interleaving, each event of h_t is between the corresponding call and return events of h_l, and has the appropriate parameter and return value. Further, h_t is a legal history of TwoStepLinSpec, by Lemma 7 and the fact that h_s is a legal history of SyncSpec. Hence h_l is linearisable with respect to TwoStepLinSpec. $\square$

The converse of the above lemma does not hold. A history of the synchronisation object might not be synchronisation-linearisable, but the corresponding log history might be linearisable with respect to TwoStepLinSpec. This is because of delays in logging: two invocations might not overlap in

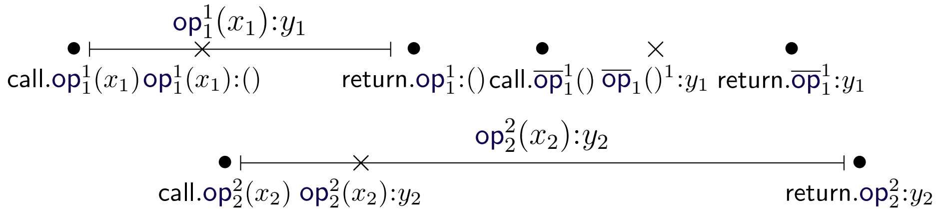


Figure 4: Illustration of the construction in the proof of Lemma 10. Horizontal lines represent the operation calls themselves. Bullets represent the log history. Crosses represent the linearisation points of the two-step synchronisation object (we omit thread identities).

reality, but might appear to overlap in the log, and so appear to be a valid synchronisation. Alternatively, the delays in logging might make it appear that two synchronisations can occur in the opposite order to what is possible with the actual history. We suspect such cases are rare in practice.

Nevertheless, the following lemma shows that any non-synchronisation-linearisable history may give rise to a non-linearisable log history, informally if the logging is done fast enough.

Lemma 10 Let h be a complete history of operation invocations that is not synchronisation-linearisable with respect to SyncSpec. Then there is a corresponding log history h_l that is not linearisable with respect to TwoStepLinSpec.

Proof: We construct h_l by interleaving with h, so that each event of h_l occurs as close as possible to the corresponding call or return in h, i.e.:

- Each $\mathsf{call.op}_j^i(x)$ in h_l occurs immediately before the corresponding call in h, for $j = 1, 2$;
- Each $\mathsf{return.op}_1^i{:}()$, $\mathsf{call.}\overline{\mathsf{op}}_1^i()$ and $\mathsf{return.}\overline{\mathsf{op}}_1^i{:}y_1$ in h_l occur immediately after the corresponding return in h;
- Each $\mathsf{return.op}_2^i{:}y_2$ occurs immediately after the corresponding return in h.

Here "immediately before" or "immediately after" means there are no intermediate events. Figure 4 gives an illustrative example.

We show that h_l is not linearisable with respect to TwoStepLinSpec. We argue by contradiction: we assume that h_l is linearisable, and deduce that h is synchronisation-linearisable. So let h_t be a history of TwoSpecLinSpec such that h_l and h_t are compatible.

We interleave h_t with the interleaving of h_l and h, by inserting each event of h_t in a way that is consistent with the interleaving of h_l and h_t, but consistent with the above construction of h_l, maintaining the "immediately before" and "immediately after" properties, so not between corresponding call/return events from h and h_l. This means:

- Each $\mathsf{op}_1^{i_1}(x_1){:}()$ from h_t occurs between the corresponding call and return events of op_1 in h;
- Each $\mathsf{op}_2^{i_2}(x_2){:}y_2$ from h_t occurs between the corresponding call and return events of op_2 in h;
- Each $\overline{\mathsf{op}}_1^{i_1}(){:}y_1$ from h_t occurs between the corresponding $\mathsf{call}.\overline{\mathsf{op}}_1^1()$ and $\mathsf{return}.\overline{\mathsf{op}}_1^1{:}y_1$, with these three events being consecutive.

Further, for a matching pair i_1 and i_2 of invocations (using Lemma 7):

- $\mathsf{op}_2^{i_2}(x_2){:}y_2$ occurs after the corresponding $\mathsf{op}_1^{i_1}(x_1){:}()$ event, and so after the corresponding call of op_1 in h.
- $\mathsf{op}_2^{i_2}(x_2){:}y_2$ occurs before the corresponding $\overline{\mathsf{op}}_1^{i_1}(){:}y_1$ event, and so before the corresponding return of op_1 in h.

Figure 4 illustrates. We linearise each synchronisation at the point of the $\mathsf{op}_2^{i_2}(x_2){:}y_2$ event. We have shown that this is within the period of each invocation. Further, by Lemma 7, this represents a legal history of SyncSpec. Hence h is synchronisation-linearisable with respect to SyncSpec: we have reached our contradiction. □

Generalisations

6 Direct testing of synchronisation linearisation

We now consider how to test for synchronisation linearisation more directly. We perform logging precisely as for standard linearisation: 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.

When not testing for progress, we make it the responsibility of the tester to define the threads in a way that ensures that all invocations will be able to synchronise, so all threads will eventually terminate. For example, for a binary heterogeneous synchronisation object, threads collectively should perform the same number of each operation.

When testing for progress, we remove the requirement on the tester to ensure that all invocations can synchronise. Indeed, in some cases, in order to find failures of progress, it is necessary that not all invocations can synchronise: we have examples of incorrect synchronisation objects where (for example) if there are two invocations of op_1 and one of op_2, then *neither* invocation of op_1 returns, signifying the failure of progressability; but if there were a second invocation of op_2, it would unblock both invocations of op_1, so all invocations would return, and the failure of progressability 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 it is highly likely that any threads that have not returned really are stuck: otherwise this approach it likely to produce false positives. In practice, we have found it easy to identify a suitable duration. A downside of this approach is that the duration needs to be chosen fairly conservatively, which increases the time that a given number of runs will take.

Discuss progress later?

In the remainder of this section we consider algorithms for determining if the resulting log history is synchronisation linearisable, and whether it also satisfies progressability. In Section 6.1 we present a general algorithm for this problem, based on depth-first search. We then consider the complexity of this problem. We show, in Section 6.2, that, in the case of a stateful synchronisation object, the problem of deciding whether a history is synchronisation linearisable is NP-complete in general. However, we show that in the case of binary synchronisations with a stateless specification object the problem can be solved 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 invocations, the problem is again NP-complete, even in the stateless case.

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 invocations that were called in the first i events of the log and that have not yet been linearised;

- A set *linearised* of operation invocations 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 invocations in *pending* can synchronise, giving results compatible with *spec*, then there is an edge to a configuration where the synchronising invocations are moved into *linearised*, and the specification object 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 invocation is in *linearised*, then that invocation 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 compatibility. We can therefore search this graph using a standard algorithm. Our implementation uses depth-first search.

6.1.1 Progress

It is straightforward to adapt the search algorithm to also test for progress. It is enough to change configurations as follows, following the definition of progressability.

- The definition of synchronisation edges is changed so that they involve only invocations that do subsequently return.

- The definition of target configurations is changed so that *pending* may be non-empty, but must contain no set of invocations that can synchronise according to *spec* (i.e. satisfying the precondition in *spec*). (However, *linearised* must still be empty.) This ensures that there no further synchronisations are possible at the end.

$\boxed{??}$ Why not leave synchronisation edges unchanged, since we require *linearised* to be empty?

6.1.2 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 invocations as soon as possible.

Definition 11 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 compatibility. 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 invocations from h; or (2) directly after another event from h_s.

Lemma 12 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 and h_s are synchronisation-compatible, then there is an ASAP interleaving of them.

Proof: Consider an interleaving of h and h_s, as required for synchronisation compatibility. 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.

Note that each event from h_s is still between the call and *return* events of the corresponding invocations. 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 linearisations.

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

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 [GK97] concerning the complexity of the corresponding problem for linearisability. Let Variable be a linearisability 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 Variable is NP-complete.

Since standard linearisation is a special case of synchronisation linearisation (in the trivial case of no 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 $\mathsf{op}_1(\text{"get"}, \mathsf{x})$ acts like $\mathsf{get}(\mathsf{x})$, and $\mathsf{op}_1(\text{"set"}, \mathsf{x})$ acts like $\mathsf{set}(\mathsf{x})$ (but returns -1). The op_2 invocations do nothing except synchronise with invocations of op_1. 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 history h of ConcVariable, we build a history h' of VariableSync as follows. We replace every call or return of $\mathsf{get}(\mathsf{x})$ by (respectively) a call or return of $\mathsf{op}_1(\text{"get"}, \mathsf{x})$; 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. 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 for binary heterogeneous synchronisations. We show that in this case the problem of deciding whether a history is synchronisation linearisable can be decided in quadratic time.

So consider a binary synchronisation object, whose specification object is stateless. Note that in this case we do not need to worry about the order of synchronisations: if each individual synchronisation is correct, then any permutation of them will be synchronisation-linearisable.

Define two complete 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 operations this can be calculated in $O(n^2)$.

Consider the bipartite graph where the two sets of nodes are invocations of op_1 and op_2, respectively, and there is an edge between two invocations if they are compatible. A synchronisation linearisation then corresponds to a total matching of this graph: given a total matching, we build a synchronisation-compatible history of the synchronisation specification object by including events $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ (in an arbitrary 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-compatible history corresponds to a total matching.

Thus we have reduced the problem to that of deciding whether a total matching exits, for which standard algorithms exist. We use the Ford-Fulkerson method, 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 invocations 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 invocations 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 maximum matching in a general graph, which can be solved using, for example, the blossom algorithm [Edm65], 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 in 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 [Kar72].

Definition 13 The problem of finding a complete 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 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 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 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.

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

7 Examples

Category	Arity	Stateful?	Heterogeneous?
Synchronous channel	2	N	Y
Filter channel	2	N	Y
Men & Women	2	N	Y
Exchanger	2	N	N
Channel with counter	2	Y	Y
Two families	2	Y	Y
One family	2	Y	N
ABC	3	N	Y
Barrier	n	N	N
Timeout channel	1, 2	N	Y
Timeout exchanger	1, 2	N	N
ABC with counter	3	Y	Y
Barrier with counter	n	Y	N
Terminating queue	1, n	Y	N

8 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 [Ros10] and its model checker FDR [GRABR15, FDR20]. 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 of a concurrent object 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 [Low19].

We assume that the model includes the following events:

- $\mathsf{call}.t.op.x$ to represent thread t calling operation op with parameter x;
- $\mathsf{return}.t.op.y$ 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 construct a specification process (Spec, below) that allows precisely traces of call and return events that are synchronisation linearisable. We construct this specification process from several components.

We build a process SyncSpec corresponding to the specification object. We assume this process uses events of the form $\mathsf{sync}.t_1.t_2.x_1.x_2.y_1.y_2$ 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.

```
Lineariser(t) =
  call.t.op₁?x₁ → sync.t?t₂!x₁?x₂?y₁?y₂ → return.t.op₁.y₁ → Lineariser(t)
  □
  call.t.op₂?x₂ → sync?t₁!t?x₁!x₂?y₁?y₂ → return.t.op₂.y₂ → Lineariser(t)
alpha(t) = {| call.t, return.t, sync.t.t₁, sync.t₁.t | t₁ ← ThreadID, t₁ ≠ t |}
```

This process 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 SyncSpec, $\mathsf{Lineariser(t_1)}$ and $\mathsf{Lineariser(t_2)}$.

```
Spec₀ = SyncSpec [| {| sync |} |] (|| t ← ThreadID • [alpha(t)] Lineariser (t))
```

Every trace of $\mathsf{Spec_0}$ represents an interleaving between a possible history of the concurrent object (call and return events) and a compatible legal history of the specification object (sync events).

Finally, we hide the sync events.

```
Spec = Spec₀ \ {| 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 this 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 specification) between k linearisers 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 (for $k = 2$).

```
Lineariser (t) =
  let others = ThreadID−{t} within
  call .t.op?x → (
    sync.t?t':others ! x?x'?y?y' → return.t.op.y → Lineariser(t)
    □
    sync?t':others ! t?x' ! x?y' ! y → return.t.op.y → Lineariser(t)
  )
```

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

8.1 Progress conditions

A simple adaptation of the above check allows us to capture an interesting progress condition, which we now describe. We make the assumption that the scheduler in the implementation schedules each operation infinitely often. This is different from the assumption corresponding to the standard property of lock freedom [HS12], which allows threads to be suspended forever; however, it is consistent with how real schedulers behave. Under this assumption, we require that if a synchronisation is possible, such a synchronisation can happen, and the relevant threads are able to return: in other words, the return events become available.

Part of our progress check is that the model of the system is divergence-free, which can be tested by FDR. Recall, that a divergence (in CSP) is an infinite sequence of consecutive internal events. In the case of the model of a synchronisation object, this would represent a livelock, i.e. where one or more threads perform infinitely many steps without reaching a point where they can return. The check forbids such livelocks.

The other part of our progress check concerns stable failures. Recall that a stable failure of a process is a pair (tr, X) representing that the process can perform trace tr to reach a stable state (i.e. where no internal event is possible), where no event from X can be performed. We test whether the stable failures of the model of the synchronisation object are a subset of the stable failures of the above Spec process. We explain the property this test captures via examples.

Consider a model of the synchronous channel, and the trace $\langle \mathsf{call.t_1.send}.4, \mathsf{call.t_2.receive.unit}\rangle$. After this trace, Spec (internally) performs $\mathsf{sync.t_1.t_2.4.unit.unit.4}$, and reaches a state where both $\mathsf{return.t_1.unit}$ and $\mathsf{return.t_2.4}$ are available. The test of the previous paragraph requires that both of these events are also available in the model of the system, i.e. both threads are able to return.

In some cases, it might be nondeterministic which synchronisation, out of two or more possibilities, occurs. For example, consider the synchronous channel, again, and the trace $\langle \mathsf{call.t_1.send}.4, \mathsf{call.t_2.send.5}, \mathsf{call.t_3.receive.unit}\rangle$. After this trace, Spec may nondeterministically perform either $\mathsf{sync.t_1.t_3.4.unit.unit.4}$ or $\mathsf{sync.t_2.t_3.5.unit.unit.5}$. Subsequently, either $\mathsf{return.t_1.unit}$ and $\mathsf{return.t_3.4}$ or, respectively, $\mathsf{return.t_2.unit}$ and $\mathsf{return.t_3.5}$ are available. The check ensures that in each case $\mathsf{t_3}$ can return, and that either $\mathsf{t_1}$ or $\mathsf{t_2}$ can return (with $\mathsf{t_3}$ returning the corresponding value).

8.2 Alternative approach

The approach described above, using lineariser processes to ensure that the sync events are between the relevant call and return events, can be expensive. However, we can do better in some cases.

By way of an analogy, testing a concurrent datatype for (standard) linearisation is often easier when one can identify explicit linearisation points: the specification can be written in terms of those linearisation points. We use a similar technique with synchronisation linearisation.

Suppose we are considering a binary synchronisation object involving operations op_1 and op_2. Our approach requires the analyst to identify points p_1 and p_1' within op_1, and a point p_2 within op_2, which we call *signal points*. These signal points must satisfy the following conditions (which the test below verifies):

1. When particular invocations of op_1 and op_2 synchronise, point p_1 is reached before point p_2, and point p_2 is reached before point p_1' (for the corresponding signal points);
2. The return values of the two invocations are available at points p_1' and p_2, respectively;
3. No other invocation reaches a signal point between points p_1 and p_1'.

Typically, p_1 will be at or before op_1 signals to op_2; p_2 will be at or after op_2 receives that signal, and at or before it signals back to op_1; and p_1' will be at or after op_1 receives that signal back. Figure 5 gives an example.

Note that condition 1 and the fact that the signal points occur within the corresponding invocations imply that p_2 occurs within *both* invocations. Thus we can use p_2 as the synchronisation point.

We augment the CSP models of the threads with the following events:

- $\mathsf{signal}_1.\mathsf{t}_1.\mathsf{x}_1$ performed by thread t_1 at point p_1, where x_1 is its parameter;
- $\mathsf{signal}_2.\mathsf{t}_2.\mathsf{x}_2.\mathsf{y}_2$ performed by thread t_2 at point p_2, where x_2 is its parameter and y_2 is its return value;
- $\mathsf{signal}_1'.\mathsf{t}_1.\mathsf{y}_1$ performed by thread t_1 at point p_1', where y_1 is its return value.

Note that the events representing the calls and returns of operations are no longer necessary.

We can then test whether the model of the synchronisation object refines the following specification (with a suitable initial state).

```
object SyncChan[T]{
  private var slot: A = _
  private val mutex = new Semaphore; mutex.up
  private val signal1, signal2 = new Semaphore // initially down

  def send(x: A) = {
    mutex.down; slot = x
    signal1.up        // signal point p1
    signal2.down      // signal point p1'
    mutex.up
  }

  def receive = {
    signal1.down
    val result = slot // signal point p2
    signal2.up; result
  }
}
```

Figure 5: An implementation of a synchronous channel, using semaphores. Signal points are indicated by comments.

```
SyncSpec(state) =
  signal1?t1?x1 → signal2?t2?x2!f2(state,x1,x2) →
  signal1'.t1 ! f1(state,x1,x2) → SyncSpec(update(state,x1,x2))
```

where f_1 and f_2 give the expected return values for the two invocations, and update describes how the state is updated. The specification ensures that the above condition 1 is satisfied. Hence, as described above, this ensures that the synchronisations can be linearised in the order of the corresponding signal_2 events.

The above condition 2 is necessary to ensure the return values can be included in the signal events. If this is not true of op_2, we could arrange for the CSP model of this operation to perform a later signal, on channel signal'_2 with that value, and to use the following specification:

```
SyncSpec(state) =
  signal1?t1?x1 → signal2?t2?x2 →
  signal1'.t1 ! f1(state,x1,x2) → signal2'.t2 ! f2(state,x1,x2) →
  SyncSpec(update(state,x1,x2))
```

Condition 3 is necessary to avoid false positives with the above specification process. The specification process handles the signals for a single synchronisation at a time.

This approach can be extended to a synchronisation between $k > 2$ operations, with signal points occurring in the order

$$p_1, p_2, \ldots p_{k-1}, p_k, p'_{k-1}, p'_{k-2}, \ldots p'_1,$$

(where the subscripts correspond to the indices of the operations), or maybe some other permutation of the p'_i events.

References

[Edm65] Jack Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17:449–467, 1965.

[FDR20] University of Oxford. *FDR Manual*, 2020. `https://dl.cocotec.io/fdr/fdr-manual.pdf`.

[GK97] P. B. Gibbons and E. Korach. Testing shared memories. *SIAM Journal of Computing*, 26(4):1208–1244, 1997.

[GRABR15] Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A. W. Roscoe. FDR3: a parallel refinement checker for CSP. *International Journal on Software Tools for Technology Transfer*, 2015.

[HS12] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2012.

[HW90] M. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[Kar72] Richard M. Karp. Reducibility among combinatorial problems. In Raymond E. Miller, James W. Thatcher, and Jean D. Bohlinger, editors, *Complexity of Computer Computations*, pages 85–103. Springer US, 1972.

[Low16] Gavin Lowe. Testing for linearizability. *Concurrency and Computation: Practice and Experience*, 29(14), 2016.

[Low19] Gavin Lowe. Discovering and correcting a deadlock in a channel implementation. *Formal Aspects of Computing*, 31:411–419, 2019.

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

[ST05] Hakan Sundell and Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *Journal of Parallel and Distributed Computing*, 65(5):609–627, 2005.