Understanding Synchronisation

Jonathan Lawrence and Gavin Lowe

September 24, 2024

Abstract

We study *synchronisation objects*: objects that allow two or more threads to synchronise. We define a correctness condition for such synchronisation objects, which we call *synchronisation linearisation*: informally, the synchronisations appear to take place in a one-at-a-time order, consistent with the invocations of operations on the object. We also define a progress condition, which we call *synchronisation progressibility*: informally, invocations don't get stuck unnecessarily.

We consider testing of implementations of synchronisation objects. The basic idea is to run several threads that use the object, record the history of operation calls and returns, and then to test whether the resulting history satisfies synchronisation linearisation and progressibility. We present algorithms for this last step, and also present results concrening the complexity of the problem.

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 to keep threads in consistent stages of the program, and so makes it easier to reason about the states of different threads.

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

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

Perhaps the most common form of synchronisation object is a synchronous channel. 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.

Each synchronisation of a synchronous channel involves two invocations of *different* operations; we say that the synchronisation is *heterogeneous*. By contrast, sometimes two invocations of the *same* operation may synchronise; we say that the synchronisation is *homogeneous*. For example, an *exchanger* has the following signature.

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

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

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

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

can be used to synchronise n threads. Each thread is assumed to have an integer thread identifier in the range [0 .. n). Each thread me calls sync(me), 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
}
```

[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 ().

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

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

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.

So far, our example synchronisation objects have been *stateless*: they maintain no state from one synchronisation to another. By contrast, some synchronisation objects are *stateful*: they maintain some state between synchronisations, which might affect the availability of the results of synchronisations. As a toy 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      // result is sequence number
  def receive(): (A, Int)  // result is (value received, sequence number)
}
```

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

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

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

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

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

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

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

A *terminating queue* can also be thought of as a stateful synchronisation object with multiple modes. Such an object 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 terminating queue might have the following signature, where a dequeue returns the value None to indicate the termination case.

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

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

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

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*; informally, the synchronisations appear to take place in a one-at-a-time order, consistent with the invocations of operations on the object.

We also define a progress condition, which we call *synchronisation progressibility*: informally, invocations don't get stuck unnecessarily.

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

We 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: we describe algorithms that test whether a history of a synchronisation object is synchronisation linearisable. We also present various complexity results: testing whether a history is synchronisation linearisable is NP-complete in general, but can be solved in polynomial time in the case of binary (heterogeneous or homogeneous) stateless synchronisation objects.

In Section 7 we describe experiments to determine the effectiveness of the testing techniques. We sum up in Section 8.

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 allow stateful synchronisation objects (which includes stateless objects as degenerative cases). We generalise later in this section.

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

def $\mathsf{op}_1(\mathsf{x}_1\colon \mathsf{A}_1)\colon \mathsf{B}_1$
def $\mathsf{op}_2(\mathsf{x}_2\colon \mathsf{A}_2)\colon \mathsf{B}_2$

(We can model a concrete operation that takes $k \neq 1$ parameters by an operation that takes a k-tuple as its parameter. We identify a 0-tuple with the unit value, but will sometimes omit that value in examples.) In addition, the

synchronisation object might have some state. Each 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*.

Synchronisation linearisation is defined in terms of a *synchronisation specification object*: we define these specification objects in the next subsection. In Section 2.2, we review the notion of linearisation, on which synchronisation linearisation is based. We then define synchronisation linearisation, for binary heterogeneous synchronisation objects, in Section 2.3. We generalise to other classes of synchronisation objects in Section 2.4. We present our progress property, synchronisation progressibility, in Section 2.5.

2.1 Synchronisation specification objects

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

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

The idea is that if two 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 $(\mathsf{y}_1, \mathsf{y}_2)$. (We allow sync to be nondeterministic; but in all our examples it will be deterministic.) The specification object might have private state, which can be 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 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. We assume the specification object is deterministic: f_1, f_2 and update are functions.

For example, consider a synchronous channel with operations

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

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

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

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

As another example, consider a filtering channel.

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

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

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

The require clause specifies that 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))
```

```
  }
}
```

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

2.2 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.

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.

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:

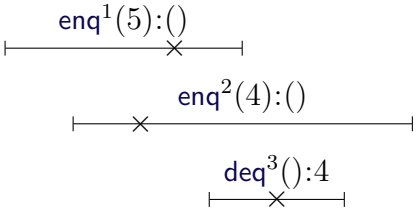


Figure 1: Timeline representing the linearisation example. Time runs from left to right; each horizontal line represents an operation invocation, with the left-hand end representing the call event, and the right-hand end representing the return event

- $op^i(x){:}y$ 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*. We assume that the specification object is deterministic: after a particular history, there is a unique value that can be returned by each invocation.

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

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

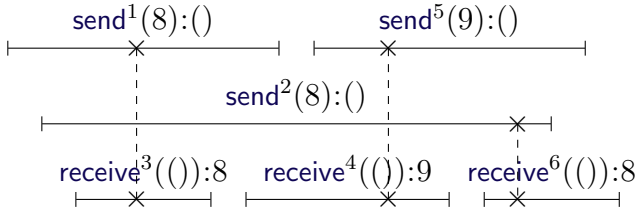


Figure 2: Timeline representing the synchronisation example.

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 $\mathit{complete}(h)$ 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 $\mathit{complete}(h')$ is linearisable. We say that a concurrent object is linearisable if all of its histories are linearisable.

2.3 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 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 $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ and $\mathsf{return}.\mathsf{op}_1^{i_1}{:}y_1$, and between $\mathsf{call}.\mathsf{op}_2^{i_2}(x_2)$ and $\mathsf{return}.\mathsf{op}_2^{i_2}{:}y_2$. In the running example, 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 $\mathit{complete}(h')$ is synchronisation-linearisable. We say that a synchronisation object is synchronisation-linearisable if all of its histories are synchronisation-linearisable.

2.4 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(x_1: A, ..., x_n: A) = {
    val result = f(x_1, f(x_2 ,... f(x_{n-1}, x_n )...)); (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 $\mathsf{call}.\mathsf{op}_1^{i_j}(x_j)$ and $\mathsf{return}.\mathsf{op}_j^{i_j}{:}y_j$ for each $j = 1, \ldots, k$. The definition of synchronisation-linearisability follows in the obvious way.

Note that if several of the op_j are the same operation, there is a choice as to the order in which their parameters are passed to sync. (However, in the case of the combining barrier, if f is associative and commutative, the order makes no difference.)

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        // send times out
  def sync_r(u: Unit) = None      // receive times out
  def sync_{s,r}(x: A, u: Unit) = (true, Some(x))  // synchronisation
}
```

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  // is the channel closed?
  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.

2.5 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); in other words, the scheduler is fair to each invocation. 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 *eventually returns* 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' 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 $complete(h')$ 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 in h'_s and return event in h'.

We give two examples, both for a synchronous channel.

Example 3 Consider $h = \langle \mathsf{call.send}^1(3), \mathsf{call.receive}^2(), \mathsf{return.receive}^2{:}3 \rangle$. Note that h has a pending invocation, and $\mathit{complete}(h) = \langle \mathsf{call.receive}^2(), \mathsf{return.receive}^2{:}3 \rangle$. It is clear that there is no history h_s of *Spec* such that $\mathit{complete}(h)$ and h_s are compatible, because $\mathit{complete}(h)$ is missing the send invocation. Definition 2 says that the channel may not block after h. Informally, a synchronisation has occurred, and so the pending return of the send invocation should occur.

Example 4 Now consider $h = \langle \mathsf{call.send}^1(3), \mathsf{call.receive}^2() \rangle$. We have that $\mathit{complete}(h) = \langle\rangle$ is compatible with the history $h_s = \langle\rangle$ of *Spec* (and no other). But the extension $h' = h^\frown\langle \mathsf{return.send}^1{:}(), \mathsf{return.receive}^2{:}3 \rangle$ is compatible with the extension $h'_s = \langle \mathsf{sync}^{1,2}(3, ()){:}((), 3) \rangle$ of h_s. Hence Definition 2 says that 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 $\mathit{complete}(h) = \langle\rangle$ is compatible with the history $h_s = \langle\rangle$ of *Spec*. But the only proper extension of h is $h' = \langle \mathsf{call.send}^1(3), \mathsf{return.send}^1{:}() \rangle$, and no history of *Spec* is compatible with $\mathit{complete}(h') = h'$.

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

3 Comparing synchronisation linearisation and standard linearisation

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

It is clear that synchronisation linearisation is equivalent to standard linearisation in the (rather trivial) case that no operations actually synchronise,

so each operation of the synchronisation specification object corresponds to a single operation of the concurrent 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 concurrent 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

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

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

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

But the history

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

is also 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 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 ~op1~(): B1
  def op2(x2: A2): B2
}
```

The idea is that the operation op_1 on SyncObj will be linearised by the composition of the two operations op_1 and $\overline{\mathsf{op}}_1$ of TwoStepLinSpec; but operation op_2 on SyncObj will be linearised by just the operation op_2 of TwoStepLinSpec, 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 2.2, 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.

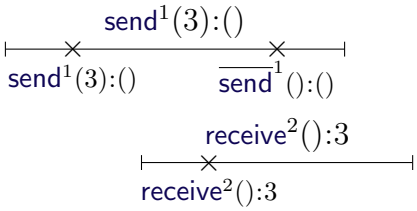
We say that h and h_s are *two-step-compatible* if there is some way of interleaving the two histories such that

- 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 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.

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

This is represented by the following timeline, where the horizontal lines and the labels above represent the 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 2.2.

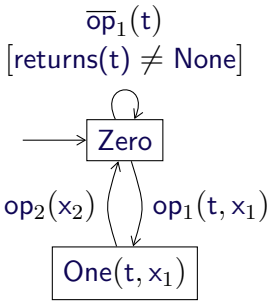
3.2 Proving the relationship

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 $\overline{\mathsf{op}}_1$.

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 $\overline{\mathsf{op}}_1$ 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 the value that should be returned by an $\overline{\mathsf{op}}_1$ 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 $\overline{\mathsf{op}}_1$ into returns.

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




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

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.

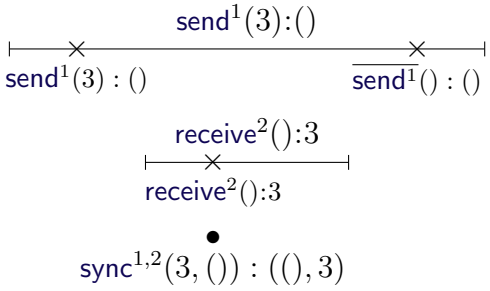
Conversely, each history with events ordered in this way will be a legal history of TwoStepLinSpec if the corresponding history of events $\mathsf{sync}^{i_1,i_2}(x_1, x_2){:}(y_1, y_2)$ is a legal history of SyncSpec.

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 each history h of SyncObj is two-step linearisable with respect to TwoStepLinSpec if and only if it is synchronisation linearisable with respect to SyncSpec. And hence SyncObj is two-step linearisable with respect to TwoStepLinSpec if and only if it is synchronisation linearisable with respect to SyncSpec.

Proof: ($\Rightarrow$). Let h be a concurrent history of SyncObj. 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 "$\times$"s and the labels below; and h'_s is represented by the "$\bullet$" and the label below.



The history h'_s is legal for SyncSpec by Lemma 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 $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ 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-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 $\mathsf{call}.\mathsf{op}_1^{i_1}(x_1)$ 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-linearisable; and so h is two-step-linearisable. □

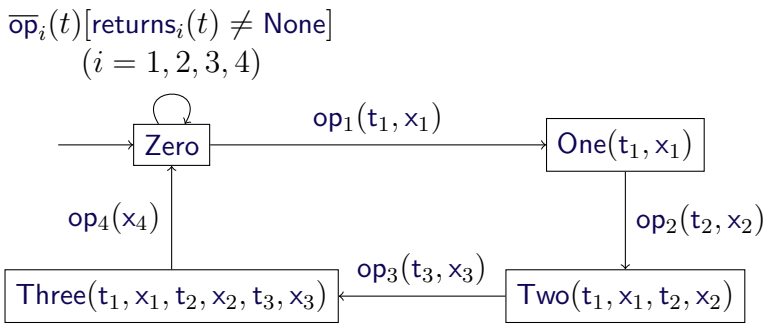
The two-step linearisation specification object can often be significantly simplified from the template definition above. Here is such a specification object for a synchronous channel.

```
object SyncChanTwoStepLinSpec{
  private var state = 0          // Takes values 0, 1, cyclically
  private var threadID = −1      // Current sender's thread ID when state = 1
  private var value: A = _       // The current value being sent when state = One
  private val canReturn =        // which senders can return?
    new Array[Boolean](NumThreads)
  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 k-ary synchronisation between distinct operations $\mathsf{op}_1, \ldots, \mathsf{op}_k$, the corresponding two-step linearisation specification object has $2k-1$ operations, $\mathsf{op}_1, \ldots, \mathsf{op}_k, \overline{\mathsf{op}}_1, \ldots, \overline{\mathsf{op}}_{k-1}$. The definition of two-step linearisation is then the obvious adaptation of the binary case: each operation 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 $k = 4$.



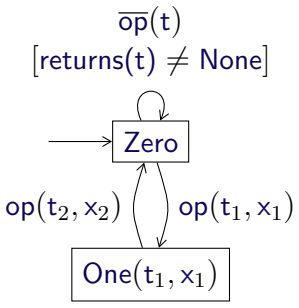
The final op operation, op_4 in the above figure, applies the sync method of the synchronisation specification object to the parameters $\mathsf{x}_1, \ldots, \mathsf{x}_k$ to obtain the results $\mathsf{y}_1, \ldots, \mathsf{y}_k$; it stores the first $k-1$ in appropriate $\mathsf{returns}_i$ arrays, 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 op̄(t: ThreadID): B = {
    require(state.isInstanceOf[Zero] && returns(t).isInstanceOf[Some])
    val Some(y) = returns(t); returns(t) = None; y
  }
}
```



Recall that some operations may have mixed modes of synchronisations: different invocations may have synchronisations with different arities. For

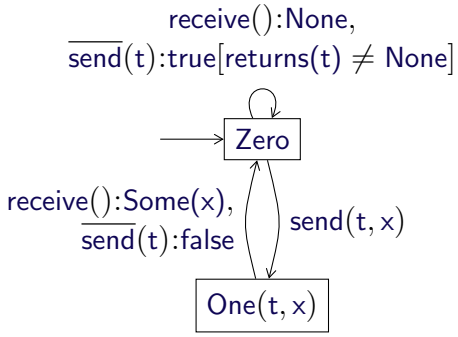


Figure 3: Automaton for capturing two-step linearisation for a timeout channel.

example, in a timed channel, an invocation of the send and receive operations may synchronise with an invocation of the other operation, or may timeout corresponding to a unary synchronisation.

We illustrate how to test for synchronisation linearisation of objects with mixed modes of synchronisation via an example. Figure 3 gives the automaton for a timeout channel, where we treat send as corresponding to op_1 (we omit concrete code in the interests of brevity). It is a small change from that for a standard channel. The receive operation can happen from either state: if it happens from the One state, then a synchronisation has occurred and the invocation returns a value of the form Some(x); but if it happens from the Zero state, there has been no corresponding send, and so the invocation returns None, indicating a timeout. Likewise, the $\overline{\text{send}}$ operation can happen from either state; if it happens from the Zero state, then a synchronisation has occurred and the invocation returns true; but if it happens from the One state, there has been no corresponding receive, and so the invocation returns false, indicating a timeout.

It is interesting to ask whether we can optimise the above construction, using just a single transition in the case of a failed send. It turns out that we cannot. To do so we require *two* send transitions from Zero, corresponding to successful and unsuccessful cases (leading to One and Zero states, respectively). This would make the automaton nondeterministic, contrary to our assumptions.

We now consider stateful specification objects. In general, we can simply augment the Zero and One states of the automaton to include the state of the specification object. Note that different transitions may be available based upon that state. For example, with a closeable channel, we could include a boolean field in the automaton states, indicating whether the channel is

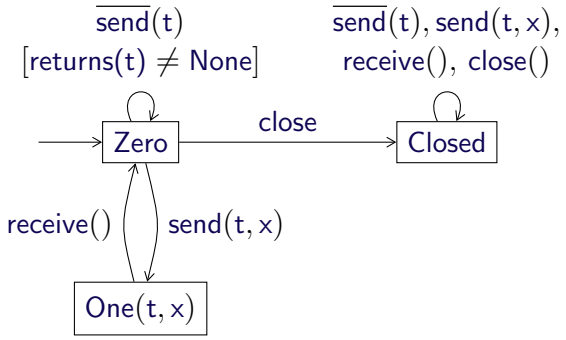


Figure 4: Automata for capturing two-step linearisation for a closeable channel.

closed. However, it can be clearer and simpler to introduce different named states into the automaton, as in Figure 4, where the automaton has an additional state, Closed, corresponding to the channel being closed. All transitions can be loops in this state, so this represents a simplification over the general approach discussed above. Note that a $\overline{\mathsf{send}}(\mathsf{t})$ transition from the Closed state may return true if the corresponding synchronisation happened before the channel was closed, in which case returns(t) $\neq$ None.

It is possible to perform a further optimisation in this case: we can capture a send that happens after the channel is closed by a *single* transition, rather than two. (The difference from the timeout channel is that these transitions happen from a different state, so the automaton is still deterministic.) *** I don't think it's worth discussing this.

We believe that the approach can be adapted to all the other synchronisation objects we have considered.

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

call.op$^i(x)$ into the log; (2) performs op(x) on the synchonisation object, obtaining result y, say; (3) writes return.op^i:y into the log. Further, the logging associates each invocation with an invocation 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 (standard) linearisability against 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 $\mathsf{call}.\mathsf{op}_1^i(x_1)$ 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 $\mathsf{return}.\mathsf{op}_1^i{:}()$ 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 $\mathsf{return}.\overline{\mathsf{op}}_1^i{:}y_1$ into the log.

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

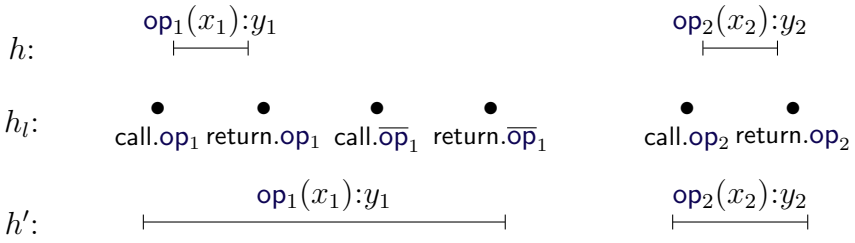
As with standard linearisation, the tester needs to define the threads so that all invocations will eventually return, i.e. that each will be able to synchronise. For a binary synchronisation with no precondition, we can achieve this by half the threads calling one operation, and the other half calling 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. Note there might be delays involved in writing to the log, so that the log history does not correspond precisely to the history of operation calls and returns. We show that the approximation serves our purposes.

Consider a history h of the synchronisation object, and suppose, for the moment, there are no delays in logging, i.e.: (1) the $\mathsf{call}.\mathsf{op}_1$ and $\mathsf{call}.\mathsf{op}_2$ events happen immediately before the actual calls; (2) the $\mathsf{return}.\mathsf{op}_2$ events happen immediately after the return of op_2; and (3) the $\mathsf{call}.\overline{\mathsf{op}}_1$ and $\mathsf{return}.\overline{\mathsf{op}}_1$ events happen immediately after the return of op_1 — in each case with no intervening events. Then the log history is linearisable with respect to

TwoStepLinSpec if and only if the history h is two-step linearisable with respect to TwoStepLinSpec. But Proposition 8 then shows that this holds if and only if h is synchronisation linearisable. In particular, this shows that errors are detected providing the logging is fast enough.

We now show that delays in logging do not introduce false errors. Consider a history h that is synchronisation linearisable, and consider the corresponding log history h_l. Now build a history h' so that each operation is extended so that: (1) each call of op_1 or op_2 is immediately before the $\mathsf{call}.\mathsf{op}_1$ or $\mathsf{call}.\mathsf{op}_2$ event in h_l; (2) each return of op_1 is immediately after the $\mathsf{return}.\overline{\mathsf{op}}_1$ event in h_l; and (3) each return of op_2 is immediately after the $\mathsf{return}.\mathsf{op}_2$ event in h_l. This construction is illustrated below for the two types of operation.



Now h is synchronisation linearisable; and hence h' is also (since each operation in h' is an extension of the corresponding operation in h). But then Proposition 8 implies that h' is two-step linearisable with respect to TwoStepLinSpec. Hence h_l is linearisable with respect to TwoStepLinSpec, by construction.

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

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 it's possible that *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. Our informal experiments suggest that a duration of 100ms is appropriate (at least, on the architecture we were using): we have not observed any false positives with this duration, but did with a shorter duration of 80ms. This delay does significantly increase the time that a given number of runs will take.

In the remainder of this section we consider algorithms for determining if the resulting log history is synchronisation linearisable, and whether it is synchronisation progressable. 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 *spec* is updated corresponding to the synchronisation;

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

Return. If the next event in the log is a return event, and the corresponding invocation is in *linearised*, then there is an edge where 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.

It is straightforward to adapt the search algorithm to also test for progress. We change the definition of a target configuration to have i at the end of the log, *linearised* empty, and such that no set of invocations in *pending* can synchronise.

6.1.1 Partial-order reduction

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

Definition 9 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.

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

Lemma 10 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 synchronisations.

Our experience is that this tactic is moderately successful. In some cases, it can reduce the total time to check histories by over 30%; although in some cases the gains are smaller, sometimes negligible. The gains seem highest in examples where there can be a reasonably large number of pending 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 an integer variable with get and set operations. Then the problem of deciding whether a given concurrent history is linearisable with respect to Variable is NP-complete.

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 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 with invocations of op_1. This can be captured formally by the following synchronisation specification object.

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

Let ConcVariable be a concurrent object that represents an integer variable. Given a history h of ConcVariable, we build a history h' of VariableSync as follows. We replace every call or return of get(x) by (respectively) a call or return of 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. Deciding the former is NP-complete; hence the latter is also.

6.3 The binary heterogeneous stateless case

The result of the previous subsection used a *stateful* specification object. We now consider the *stateless* case. We show that for binary heterogeneous synchronisations, the problem of deciding whether a history is synchronisation

linearisable can be decided in quadratic time. We consider the homogeneous case in the next subsection.

So consider a binary heterogeneous synchronisation object, whose specification object is stateless. 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 will also be correct from the point of view of the specification object; and we can order the synchronisations in a way that is compatible with the concurrent history. Informally, the idea is to find matching invocations in the concurrent history that could correspond to a particular synchronisation; we therefore reduce the problem to that of finding a matching in a graph.

Define two complete 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 appropriate order) whenever there is an edge between $\mathsf{op}_1^{i_1}(x_1){:}y_1$ and $\mathsf{op}_2^{i_2}(x_2){:}y_2$ in the matching; and conversely, each synchronisation-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 [FF56], 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. This problem 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 11 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. Hence finding a synchronisation linearisation is NP-complete.

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

6.6 Implementation

We have implemented a testing framework (in Scala), based on the above algorithms, and used it to implement testers for particular synchronisation objects[4]. We consider the framework to be straightforward to use: most of the boilerplate code is encapsulated within the framework; defining a tester for a new synchronisation object takes just a few minutes.

Figure 5 gives a stripped-down tester for a synchronous channel. (The full version can be used to test a number of implementations with the same interface, and replaces the numeric constants by parameters that can be set on the command line.)

The worker function defines a worker thread that performs operations on the channel c. The function also takes parameters representing the thread's identity and a log object. Here, each worker with an even identity performs 20 receive invocations: the call log(me, c.receive(), Receive) logs the call, performs the receive, and then logs the return. Similarly, each worker with an odd identity performs 20 send invocations of random values. This definition is designed so that an even number of workers with contiguous identities will not deadlock.

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

The function doTest performs a single test. This uses a BinaryStatelessTester object from the testing framework, which encapsulates the search from Section sec:binary-heterogeneous. Here, the tester runs 8 worker threads, and tests the resulting history against matching. If a non-synchronisation-linearisable history is recorded, it displays this for the user. The main function runs doTest either 5000 times or until an error is found. The tester can be adapted to test for synchronisation progressibility by passing a timeout duration to the BinaryStatelessTester.

Other classes of testers are similar. In the case of a stateful specification,

[4]The implementation is available from ???.

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

  def worker(c: SyncChan[Int])(me: Int, log: HistoryLog[Op]) =
    for(i <- 0 until 20)
      if(me%2 == 0) log(me, c.receive(), Receive)
      else{ val x = Random.nextInt(100); log(me, c.send(x), Send(x)) }

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

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

  /** Do a single test.  Return true if it passes. */
  def doTest(): Boolean = {
    val c = new SyncChan[Int]
    new BinaryStatelessTester[Op](worker(c), 8, matching)()
  }

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

Figure 5: A simple tester for a synchronous channel.

the matching function takes the specification object as a parameter, and also returns the new value of the specification object.

7 Experiments

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

We consider synchronisation objects implementing a number of interfaces, summarised in Figure 6. Most of the interfaces were described in earlier sections (namely synchronous channel, filter channel, exchanger, barrier, timeout channel, closable channel, enrollable barrier, and terminating queue).

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

Figure 6: Example interfaces of synchronisation objects.

The *men and women* problem involves two families of threads, known as men and women: each thread wants to pair off with a thread of the other type; each passes in its own identity, and expects to receive back the identity of the thread with which it has paired. In the *two families* problem, there are two families of threads, with n threads of each family; each thread calls an operation n times, and each invocation should synchronise with a thread of the opposite family, a different thread each time. In the *one family* problem, there are n threads, each of which calls an operation $n - 1$ times, and each time should synchronise with a different thread. The *ABC* problem can be thought of as a ternary version of the men and women problem: there are three types of threads, A, B and C; each synchronisation involves one thread of each type. Finally, the *timeout exchanger* is a timed version of the exchanger: if a thread fails to exchange data with another thread, it can timeout and return an appropriate result.

For each interface, we have implemented a tester, using the appropriate algorithm from earlier sections, and have produced a correct implementation. For most interfaces, we also implemented one or more faulty versions that fail to achieve either synchronisation linearisation or progressibility. The faulty versions mostly have realistic mistakes: mistakes that we believe programmers could make.

We describe various experiments below: some concern the time required to find bugs; and some concern the throughput for correct versions. The experiments were performed on an eight-core machine (two 2.40GHz Intel(R)

Xeon(R) E5620 CPUs, with 12GB of RAM).

In each experiment below, we performed *observations*, by running a tester multiple times, and recording the time taken. Each observation aims to reflect a typical use case. In each experiment we performed multiple observations; we give the average running time and a 95%-confidence interval. Each observation was run as a separate operating system process, to ensure independence. The number of observations is chosen so as to obtain a reasonably small confidential interval, but avoiding excessively long experiments.

TO DO Experiments comparing two-step and direct algorithms.

```
scala -cp .:/home/gavin/Scala/Util experiments.TwoStepExperiment --samples 10
```

Most of the testers have a number of parameters. The parameters that are likely to have the biggest effect on the likelihood of finding bugs are

- the number of threads to run;
- the number of invocations to be performed by each thread in a run.

We ran experiments on two testers, using incorrect implementations of a synchronous channel and the ABC problem, to investigate how the time taken to find an error is affected by these two parameters. Each observation used particular values for these parameters; the observation performed repeated runs until an error was found, and recorded the time taken. (Note that the two testers assume that the number of threads is divisible by two or three, respectively; and we believe that the error in the ABC case requires more than three threads.) For each choice of parameters, 200 observations were performed. Figure 7 gives results, displaying average times and 95%-confidence intervals.

In both cases, bugs are found fastest if threads perform a fairly small number of invocations in each run, around four. Also, beyond a certain limit, running more threads means that it takes longer to detect bugs (although this is clearer for the synchronous channel than the ABC problem). However, we consider it appropriate to run more threads than are required for a single synchronisation, in case bugs depend upon two different synchronisations interfering (as is the case with the faulty ABC implementation).

We now describe experiments concerning how quickly the framework discovers a range of bugs that represent a failure of synchronisation linearisation. In most cases, we ran four threads, except for the ABC problem we ran six, and for the exchanger we ran 16. In most cases, threads performed four invocations per run, except for the exchanger where each thread performed one invocation (to avoid the possibility of deadlocks), and in the one- and two-family problems, where the number of invocations is defined by the problem.

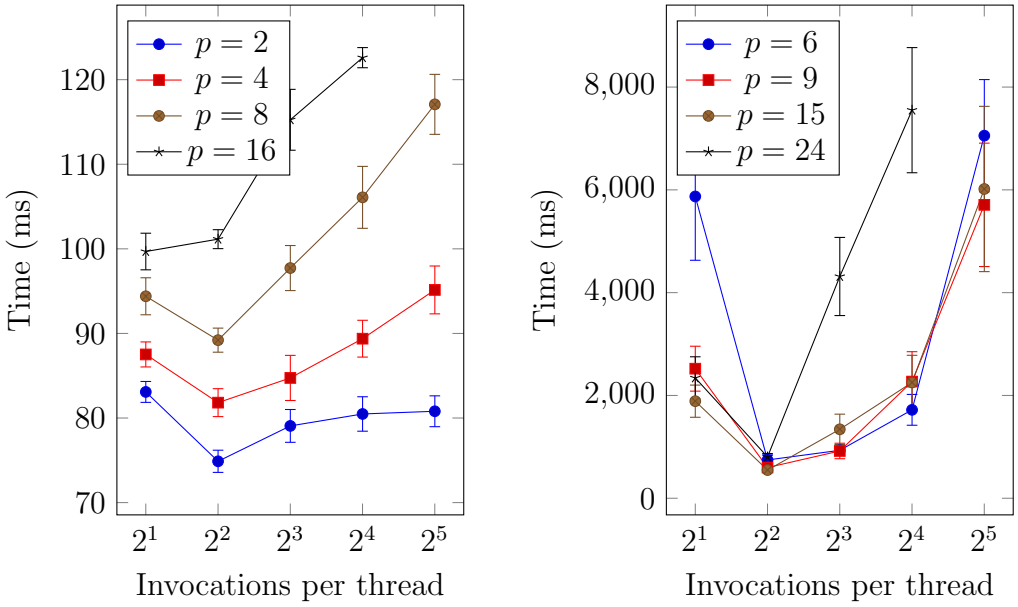


Figure 7: Results of tuning experiments for finding errors in implementations of a synchronous channel (left) and the ABC problem (right). p denotes the number of threads.

Each observation performed repeated runs until an error was found. For each experiment, we performed 200 observations.

Figure 8 (left) gives results. In each, the bug was found quickly, in less than one second on average. We believe that the variation in times mostly reflects differences between the bugs, rather than differences between the testing algorithms.

We now describe experiments to measure the throughput of the testers. We have tried to make the results comparable, although that is not completely possible, given the different nature of the synchronisation objects: each observation is based on approximately 240,000 invocations. For most objects, we ran four threads, each performing six operation invocations per run; the relevant algorithm was then used to test whether the history was synchronisation linearisable. Each observation performed 10,000 runs, which might represent a typical use case. The ABC tester assumes that the number of threads is divisible by three (so there are equal numbers of A-, B- and C-threads); we therefore ran six threads, each performing four invocations per run, to give a total of 24 invocations per run, the same as the previous cases. For the exchanger, we ran 24 threads, each performing one invocation. For the one family tester, we ran five threads (giving a total of $5 \times 4 = 20$ invocations per run in total), and adjusted the number of runs per observation to

Synchronous channel	82 ± 2
Men and women	77 ± 1
Exchanger	81 ± 1
Two families	260 ± 18
One family	349 ± 20
ABC	762 ± 81
Timeout channel	127 ± 4
Timeout exchanger	225 ± 21
Closeable channel	182 ± 8

Synchronous channel	$6{,}670 \pm 107$
Filter channel	$6{,}987 \pm 58$
Men and women	$6{,}711 \pm 56$
Exchanger	$28{,}930 \pm 159$
Two families	$10{,}431 \pm 67$
One family	$13{,}403 \pm 350$
ABC	$11{,}462 \pm 78$
Barrier	$10{,}575 \pm 90$
Timeout channel	$57{,}860 \pm 125$
Timeout exchanger	$105{,}784 \pm 118$
Closeable channel	$6{,}718 \pm 73$
Terminating queue	$6{,}881 \pm 49$

Figure 8: Times (in ms) to find bugs (left), and to run and analyse 240,000 invocations (right).

give the same number of invocations per observation as previous cases. For the two family tester, we ran three threads in one family and four in the other (again giving a total of 24 invocations per run in total). For the terminating queue, we ran four threads, with two threads always dequeueing, and two enqueueing with probability 0.5 and dequeueing with probability 0.5; we ran the threads until the termination condition was reached; this gives slightly more invocations per run on average than previous cases; this process was repeated until the total number of invocations reached that in previous cases.

Figure 8 (right) gives times (based on ten observations per experiment). Most of the testers give a throughput of tens of thousands of invocations per second. Where testers are slower, this is mostly in accordance with the earlier theoretical results. The exchanger, timeout channel and timeout exchanger are slower simply because the synchronisation objects themselves are slower, because threads spend a considerable proportion of the time waiting; informal profiling shows that about 87%, 97% and 94%, respectively, of the time is spent running the objects, as opposed to examining the logs.

Redo Exchanger? High variance

7.1 Progress

We now describe experiments concerning progressibility.

We carried out informal experiments to find an appropriate timeout time. With a delay of 80ms, we encountered false positive errors on some implementations: the system timed out just before threads could have returned. However, with a delay of 100ms, we encountered no false positives on a range

Synchronous channel	359 ± 29
Filter channel	382 ± 36
Men and women	237 ± 15
One family	1168 ± 253
ABC	996 ± 114
Barrier	168 ± 3

Figure 9: Times (in ms) taken to find errors of progressibility.

of implementations. We therefore use a 100ms delay on subsequent experiments. However, we suspect a different length of delay might be required on different architectures.

Figure 9 gives times to find failures of progressibility on various implementations of concurrent objects (these are different implementaitons from those considered earlier). The experimental set up was as earlier. The errors are again found quickly. We believe that the reason these errors take slightly longer to find than previously is because of the 100ms delays before timeouts: these simply slow down the throughput of the testing system.

We also carried out some experiments to assess the throughput on correct implementations when testing for progressibility. However, the times were dominated by the times waiting for timeouts. Where there were differences between examples, these simply reflect the probability of the system completing on its own, and not having to wait for the timeout. We omit the results, because they are uninteresting.

8 Conclusions

Model checking.

References

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

[FF56] Lestor R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8:399–404, 1956.

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

[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.

[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.

[WBM+10] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Alting barriers: synchronisation with choice in Java using JCSP. *Concurrency and Computation: Practice and Experience*, 22(8), 2010.