Testing Synchronisation Objects: User Manual

Gavin Lowe

March 31, 2025

1 Introduction

This manual describes a package for testing synchronisation objects. The package uses the Scala programming language [OSV08]. We briefly describe the purpose of such synchronisation objects, and appropriate correctness conditions. For full details, see [LL25].

A *synchronisation object* allows synchronisations between two or more threads. The most common form of synchronisation object is a synchronous channel. Here, threads call operations to send or receive data. Each invocation should overlap in time with an invocation of the opposite sort, and the receive operation should return the value sent by the send operation.

More generally, a synchronisation involves some number of invocations that overlap in time and that (normally) exchange data. The specification for the object describes what synchronisations are allowed (i.e. which operations synchronise with which others), and the values each should return.

We use the term *binary synchronisation* for a synchronisation between two invocations; and we use the term *binary synchronisation object* for a synchronisation object all of whose synchronisations are binary.

Synchronisations in a synchronous channel are between two *different* operations, sending and receiving; we call such synchronisations *heterogeneous*. By contrast, some synchronisations involve two invocations of the *same* operation; we call such synchronisations *homogeneous*. For example, an exchanger object has a single operation, `exchange`, taking a single parameter; two invocations of this operation should synchronise, and each should return the other's parameter, so the two threads exchange data values.

Some synchronisations are between more than two threads. We use the term *arity* for the number of threads involved in a synchronisation. For example, a barrier synchronisation object allows synchronisations between `n` threads, where `n` is a parameter of the object's constructor; so these synchronisations have arity `n`.

Further, we can consider an invocation that synchronises with no other invocation as a *unary* synchronisation (arity 1). Some operations might have mixed modes of synchronisations, maybe with different arities; for example,

a channel with a timeout might perform a successful binary synchronisation, or might fail to synchronise with another thread and timeout, and so have a unary synchronisation.

Synchronisation objects may maintain some state between one synchronisation and another. For example, a closeable channel is like a synchronous channel, except it can be closed, after which any attempt to send or receive will fail; hence it has two states, open and closed. As another example, a resignable barrier is like a normal barrier, except threads may enrol and resign from the barrier; each synchronisation is between all the currently enrolled threads; hence the state of the object is (abstractly) the set of currently enrolled threads. We use the term *stateful* for such synchronisation objects, and use the term *stateless* otherwise.

The standard correctness condition for synchronisation objects is *synchronisation linearisation* [LL25]. This states that the synchronisations appear to take place in a one-at-a-time order, with each synchronisation happening between the calls and returns of the relevant invocations; further, the values returned by each call should match an appropriate specification for the object (which might depend on the previous synchronisations in the case of a stateful synchronisation object). This is a safety property.

The liveness property *synchronisation progressibility* states, in addition, that invocations don't get stuck unnecessarily: if there is a possible synchronisation between pending invocations, then some such synchronisation should happen; and once a synchronisation has occurred, the relevant invocations should return.

1.1 Testing synchronisation objects

Our approach to testing synchronisation objects is as follows. We run a number of *worker threads*, that invoke operations upon the object, logging the calls and returns. We then run an algorithm over the log history to test whether it is synchronisation linearisable and (optionally) synchronisation progressible. Appropriate algorithms are described in [LL25]: the testing framework encapsulates these algorithms, and it is not necessary for the reader to understand them in order to use the framework. However, different algorithms are appropriate in different cases, depending on: whether synchronisations are binary or not; in the binary case, whether synchronisations are heterogeneous or homogeneous; and whether the synchronisation object is stateful or stateless.

In the remainder of this manual, we describe how to use the testing framework for carrying out such tests. In Section 2, we present a simple input script, describing the technique for binary heterogeneous stateless objects: we describe how to test for synchronisation linearisation; describe various pragmatics; and then describe how to test for synchronisation progressibility. In Section 3 we describe how to use the other testing algorithms,

for homegeneous, non-binary or stateful synchronisation objects. In Section 4 we describe how to interpret the output produced by a tester when it finds a history that is not synchronisation linearisable, or not synchronisation progressibility: the tester prints the history together with some explanation.

The implementation is available via `https://www.cs.ox.ac.uk/people/gavin.lowe/UnderstandingSynchronisation/`.

We assume familiarity with Scala [OSV08] throughout this manual.

2 An example script

We outline how to write a testing script via a simple example. Consider a class SyncChan that implements a synchronous channel, extending a trait Chan, defined as follows.

```
trait Chan[A]{
  def send(x: A): Unit
  def receive(): A
}

class SyncChan[A] extends Chan[A]{ ... }
```

The intention is that an invocation send(x) sends the value x, and syncronises with an invocation receive(), which should return x.

Figure 1 gives a stripped-down script for testing objects of this class for synchronisation linearisation. (The full script allows several different classes that implement Chan to be tested, and replaces the numeric constants in the script with variables that can be set on the command line; the same is true of later example scripts.)

The trait Op gives a representation of operation invocations. The subclasses have obvious meanings. Objects of these subclasses are stored in the log, and so correctness is defined in terms of them.

Each worker thread that performs invocations on a channel c is defined by the function worker(c). This function takes as parameters the identity me of the worker, and a log object log. Here, in order to avoid deadlocks, we run an even number of workers, where workers with an even identity perform receives, and workers with an odd identity perform sends. Each worker performs 4 invocations.

The code log(me, c.receive(), Receive) (a call of the apply operation on the log object) performs a receive. The three parameters are: the identity of the thread; the invocation to be performed on the channel; and the representation of the invocation to be used in the log. This logs the call of the invocation, performs the invocation on the channel, and logs the return together with the result of the invocation (i.e. the value received). Similarly,

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

  /** A worker. An even number of these workers should not produce a
    * deadlock. */
  def worker(c: Chan[Int])(me: Int, log: HistoryLog[Op]) = {
    for(i <- 0 until 4)
      if(me%2 == 0) log(me, c.receive(), Receive)
      else{ val x = Random.nextInt(100); log(me, c.send(x), Send(x)) }
  }

  /** The specification class. */
  object Spec{
    def sync(x: Int, u: Unit) = ((), x)
  }

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. */
  def matching: PartialFunction[(Op,Op), (Any,Any)] = {
    case (Send(x), Receive) => Spec.sync(x, ()) // = ((), x)
  }

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

  def runTests(reps: Int) = {
    var i = 0; while(i < reps && doTest) i += 1
  }

  def main(args: Array[String]) = runTests(5000)
}
```

Figure 1: An example testing script.

each worker with an odd identity sends a random value x, with suitable logging, via the code log(me, c.send(x), Send(x)).

The specification of the channel is captured by the combination of the synchronisation specification object Spec and the partial function matching. The latter captures that invocations represented in the log by Send(x) and Receive can synchronise together, and the values each returns are given by Spec.sync(x, ()) = ((), x); i.e. the send should return the unit value (), and the receive should return x. More generally, the domain of the matching function represents all pairs of invocations that can synchronise. (Alternatively, we could dispense with Spec, and inline its sync method; however, it can be more convenient to have an explicit synchronisation specification object, particularly in the case of stateful objects.)

The function doTest performs a single test, and returns a boolean indicating if the resulting history is synchronisation linearisable. It creates a particular SyncChan object c to be tested. The class BinaryStatelessTester encapsulates an algorithm for testing a binary heterogeneous stateless synchronisation object, such as a synchronous channel. The constructor takes: a type parameter corresponding to the log representation of invocations (here Op); a function representing a single worker (here worker(c)); the number of workers to run (here 4); and the partial function that specifies the synchronisations (here matching). The doTest function creates a tester object bst. The code bst() (a call to the apply method of bst) then runs the worker threads, and tests whether the resulting log is synchronisation linearisable; the result of that call is also the result of doTest.

Note that each test is rather short: four workers, each performing four invocations. Experiments in [LL25] show that running short tests like this tends to find errors, if they exist, faster.

The code runTests(reps) executes doTest at most reps times, or until it finds an erroneous history. Finally, the main method uses the runTests method to perform at most 5000 tests.

2.1 Using the Tester interface

The trait synchronisationTester.Tester in the distribution, outlined in Figure 2, encapsulates some boiler-plate code often used in testers. Objects that extend Tester must provide the doTest method, but can inherit the runTests method, which executes doTest a set number of times or until an error is found. Thus we could have extended Tester in Figure 1, and avoided defining runTests.

The runTests method also prints dots on the screen to show progress. This can be useful to be sure that the threads working on the synchronisation object have not deadlocked. By default, runTests prints one dot every 100 runs, or, if the progressCheck flag is set, it prints a dot after every test (this flag is intended for use with progress checks, described in Section 2.2, which tend

```
trait Tester{
  /** Number of worker threads to run. */
  var p = 4

  /** Number of iterations per worker thread. */
  var iters = 4

  /** Do we check the progress condition? */
  var progressCheck = false

  /** Number of runs for each dot printed. */
  def runsPerDot = if(progressCheck) 1 else 100

  /** Do a single test. Return true if it passes. Defined in subclasses. */
  def doTest: Boolean

  /** Run 'reps' tests.
    * @param timing are we doing timing experiments? */
  def runTests(reps: Int, timing: Boolean = false) = { ... }
}
```

Figure 2: The Tester interface.

to be slower); however, the frequency of dots can be adjusted by overriding runsPerDot.

In addition, the doTest method prints the time taken, in milliseconds. If the optional timing parameter is set, the time is printed in nanoseconds; this is intended for use in timing experiments.

It is useful to allow various parameters of tests to be set on the command line, such as: the number of worker threads to use in each test; the number of iterations to be performed by each worker in each test; the number of tests to perform; and the maximum value to use for a data value (such as the value sent in Figure 1). The variables p and iters in Tester are intended to be used for the first two of these.

2.2 Progress checks

In order to check for synchronisation progressibility (in addition to synchronisation linearisation), it is necessary to pass a positive integer value to the apply function of bst, representing a timeout time, in milliseconds. For example

```
bst(100)
```

This will run threads, but interrupt them after the specified time, in milliseconds, if they have not all terminated. It then tests the resulting log

for synchronisation progressibility, i.e. checking that no pending invocations failed to return when they could have done.

Each of the other testing algorithms, described later, also has an apply function that takes an optional integer argument, with the same meaning.

It is necessary to choose a timeout time that is large enough to ensure that any threads that can still run have time to do so, i.e. to avoid interrupting threads that were about to return, which would lead to a false failure of progressibility being reported. Conversely, too large a timeout time will make the testing take longer. Our experience is that a time of 100ms is suitable on most synchronisation objects.

When checking for progressibility, it is no longer necessary to design the threads to avoid deadlocks. Indeed, it is sensible to allow the possibility of deadlocks, in order to provide greater test coverage. A sensible approach is to arrange for workers to pick invocations at random, for example:

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

The interruption is done by calling the interrupt method of the Thread class on each worker, expecting each to throw an InterruptedException. Most, but not all, concurrency primitives will react to the interrupt method as required. In some cases, it will be necessary to build in an additional mechanism to deal with the interruption.

3 Other algorithms

In the previous section, we considered the class BinaryStatelessTester for testing binary heterogeneous stateless synchronisation objects. In this section, we describe classes that encapsulate other testing algorithms.

All example scripts referred to in this section can be found in Appendix A. Fuller versions of each script are included in the distribution.

Each script tests only for synchronisation linearisation, but can be adapted to test for progressibility by passing a positive integer to the apply method of the object encapsulating the algorithm, as for the synchronous channel tester in the previous section.

3.1 Stateless testers

Figure 3 gives the signatures for each of the classes that encapsulates an algorithm for testing a stateless synchronisation object (including BinaryStatelessTester, described in the previous section).

The class HomogeneousBinaryStatelessTester is for testing binary homogeneous stateless synchronisation objects. It takes the same parameters as

```
/** Testing algorithm for binary heterogeneous stateless synchronisation
  * objects. */
class BinaryStatelessTester[Op](
    worker: (Int, HistoryLog[Op]) => Unit,
    p: Int,
    matching: PartialFunction[(Op,Op), (Any,Any)]
){
  def apply(delay: Int = -1): Boolean
}

/** Testing algorithm for binary homogeneous stateless synchronisation
  * objects. */
class HomogeneousBinaryStatelessTester[Op](
    worker: (Int, HistoryLog[Op]) => Unit,
    p: Int,
    matching: PartialFunction[(Op,Op), (Any,Any)]
){
  def apply(delay: Int = -1): Boolean
}

/** Testing algorithm for general binary stateless synchronisation objects. */
class StatelessTester[Op](
    worker: (Int, HistoryLog[Op]) => Unit,
    p: Int,
    arities: List[Int],
    matching: PartialFunction[List[Op], List[Any]],
    suffixMatching: List[Op] => Boolean = (es: List[Op]) => true
){
  def apply(delay: Int = -1): Boolean
}
```

Figure 3: Signatures for the stateless testers.

in the heterogeneous case; and likewise the apply function takes the same optional parameter.

Figure 7 gives an example for this class, giving a testing script for an exchanger. Threads call the method exchange on the exchanger, passing in a value; this call should synchronise with another call, and both threads should receive the other's value.

Most parts of the script are straightforward, and similar to the script for the synchronous channel.

One point to note about this script is that each worker thread performs a *single* invocation: otherwise it is possible for the system to deadlock, for example if one thread has two remaining invocations but all the others have terminated. This tactic might be of use elsewhere.

The class StatelessTester can test arbitrary stateless synchronisation ob-

jects. Its signature is again in Figure 3. The parameters Op, worker and p are as for the binary testers. The parameter arities is a list of all possible arities of synchronisations. The parameter, matching is much like in the binary case, except its domain is all *lists* of operations that might synchronise, and it returns a corresponding *list* of expected results. Finally, the optional parameter suffixMatching is a function that should return true when its argument is a suffix of a possible synchronisation (with default value that always returns true); we explain below how this can be useful in optimisations.

Figure 8 gives an example, giving a testing script for the ABC problem. Here, the synchronisation object provides three operations, syncA(a: A), syncB(b: B), and syncC(c: C). Each synchronisation should be between three invocations, one of each operation, with each invocation returning the parameters of the other two invocations.

Most aspects of the tester are straightforward. Here the tester runs 6 threads (the number must be a multiple of 3 to avoid deadlocks), with two for each operation. Each synchronisation involves 3 invocations.

Figure 9 gives another example, for a timeout channel, where an invocation can fail to synchronise and timeout. The sendWithin operation returns a boolean, indicating whether is correctly sent its value. The receiveWithin operation returns an Option value, with Some(x) indicating that it received x, and None indicating that it timed out.

The definition of matching illustrates how to specify mixed modes of synchronisation: a send(x) may fail to synchronise, so timeout and return false; a receive may fail to synchronise, so timeout and return None; or a send(x) and receive may synchronise and return true and Some(x), respectively. Thus synchronisations may have arities 1 or 2, as captured by the parameter List(1,2) of the StatelessTester constructor.

Figure 10 gives an example of a tester for a barrier synchronisation object. Each such object is used by some number n of threads, each of which calls an operation sync: no call to sync should return until all n threads have called it, so this synchronises all n threads.

Most aspects of the script are straightforward. A synchronisation will be represented by a list of n Sync objects, one for each worker. However, with a naive approach, each such synchronisation could be represented in n! different ways, giving an increase in the complexity of checking. We therefore make the decision that we will require each such list to be in sorted order of the workers' identities (as tested by the recursive isSorted function), so as to reduce the number of cases considered.

We also supply an argument for the optional parameter suffixMatching of the StatelessTester constructor. Recall that this parameter tests whether its argument is a suffix of a possible synchronisation, so this function tests if the identities are of the form $[k \,..\, n)$ for some k. Internally to the StatelessTester, this reduces the number of lists of operations built as possible synchronisations. When run with six worker threads, the use of this parameter reduces

```
class BinaryStatefulTester[Op,S](
  worker: (Int, HistoryLog[Op]) => Unit,
  p: Int,
  specMatching: S => PartialFunction[(Op,Op), (S,(Any,Any))],
  spec0: S,
  doASAP: Boolean = false
){
  def apply(delay: Int = -1): Boolean
}

class StatefulTester[Op,S](
  worker: (Int, HistoryLog[Op]) => Unit,
  p: Int,
  arities: List[Int],
  matching: S => PartialFunction[List[Op], (S,List[Any])],
  suffixMatching: List[Op] => Boolean = (es: List[Op]) => true,
  spec0: S,
  doASAP: Boolean = false
){
  def apply(delay: Int = -1): Boolean
}
```

Figure 4: Signatures for the stateful testers.

the running time of the tester by a factor if over 20, although the speed-up is less with fewer workers.

3.2 Stateful testers

We now consider stateful testers. The classes that encapsulate testing algorithms are described in Figure 4.

The algorithm for testing binary stateful synchronisation objects is encapsulated in the class BinaryStatefulTester. The parameters are as follows.

- The type parameter Op is the representation of operation invocations in the log, as before.
- The type parameter S is the type of synchronisation specification objects, giving an abtract representation of the state of the synchronisation object. Such specification objects should be *immutable*; they should have a suitable definition for equality (equals) and a compatible hashCode.
- The parameter worker defines a worker that operates on the synchronisation object, as before.
- The parameter p gives the number of threads to run, as before.

- The parameter specMatching captures the specification. This function takes a parameter s corresponding to the state of the specification object. Then specMatching(s) is a partial function defining what synchronisations are allowed given the state s. Its domain, as before, is the pairs of invocations that may synchronise; the function returns the resulting state of the specification object, and the results to be returned by the two invocations.

- The parameter spec0 is the initial state of the specification object.

- The optional parameter doASAP specifies whether the ASAP partial order reduction [LL25] should be employed. Experience suggests that this is normally beneficial.

Figures 11 and 12 give an example, for the one-family problem. Here, n threads, with identities [0 .. n), each call a method sync at most n − 1 times, passing in its own identity. Each such invocation should synchronise with another invocation, and return the identity of the other thread; however, each pair of threads should synchronise together at most once. Hence the synchronistion object is stateful: abstractly, its state is the set of pairs of threads that have synchronised so far.

The specification object Spec(bitMap) captures this state using the bitmap bitMap; for each pair of threads a and b, bitMap(a)(b) is true if they have already synchronised. The operation sync(a, b) specifies the result of a synchronisation between threads a and b. This is allowed only if a and b have not already synchronised, as captured by the **requires** check; if this check fails, the testing framework catches the resulting IllegalArgumentException, but does not allow the synchronisation to be linearised in this state. If the synchronisation is allowed, it creates a new bitmap recording the synchronisation, and returns a corresponding new specification object, together with the correct results for the two invocations. Recall that the specification object must be immutable: hence we create a new specification object rather than simply updating the current one. Recall also that the specification object must have appropriate definitions of equality and hash code: we define equality as value equality over the bitmaps, and the hash code based directly on the content of the bitmap.

The matching function defines that two invocations may synchronise as captured by the sync method on the current specification object. We could have captured the precondition of the synchronisation within matching, and dispensed with the **requires** check in Spec.sync, as follows:

```
def matching(spec: Spec): PartialFunction[(Sync,Sync), (Spec,(Any,Any))] = {
  case (Sync(a), Sync(b)) if !spec.bitMap(a)(b) && !spec.bitMap(b)(a) =>
    spec.sync(a, b)
}
```

Most of the rest of the definitions are straightforard. In the construction of the BinaryStatefulTester, we start with a specification object whose bitmap records no previous synchronisations. We choose to employ the ASAP partial order reduction.

The class StatefulTester encapsulates the algorithm for testing general stateful synchronisation objects; it is described in Figure 4. The type parameters and most of the parameters are as for BinaryStatefulTester. The type of matching is adapted to capture that a *list* of invocations synchronise, as in StatelessTester; and the parameters arities and suffixMatching are as in StatelessTester.

As an example, a tester for a closeable channel is given in Figures 13 and 14. A closeable channel has an operation close to close the channel. An attempt to send or receive after the channel has been closed should fail, and throw a ClosedException. The closeable channel mixes binary and unary synchronisations, so we cannot use BinaryStatefulTester here.

The testing algorithm cannot deal directly with the exceptions, so instead we build wrappers round the operations, to map the results to proper values: the function trySend maps the result of a send to a boolean, with true representing success, and false representing failure; the function tryReceive maps the result of a receive to an Option value, with Some(x) representing the receipt of x, and None representing a failure.

The channel has two states, open and closed. We represent this state using a boolean, with true representing that the channel has been closed. The function matching then specifies allowed synchronisations: a send and receive can synchronise in the normal way if the channel is not closed (and the channel remains not closed); a send or receive can fail if the channel is closed (and the channel remains closed); and a close operation can always succeed (even if the channel is already closed), and subsequently the channel is closed.

The definition of a worker is straightforward. On each iteration, a worker closes the channel with probability 0.05. Otherwise, workers act much as for a standard channel, but we use the functions trySend and tryReceive as above.

We use the default value for suffixMatching, which treats all lists as being a possible suffix. We choose not to use the ASAP optimisation here, since it seems not to help. The rest of the definitions are then straightforward.

As another example, a testing script for a resignable barrier is in Figures 15–17. A resignable barrier is like a normal barrier, except workers may enrol or resign from the barrier, and each barrier synchronisation is between the workers currently enrolled.

The synchronisation specification object Spec is parameterised by the (immutable) set enrolled of identities of threads currently enrolled. The methods enrol and resign on Spec correspond to the operations with the same names on the barriers; each definition is straightforward; the assertions are just sanity checks, that the workers have satisfied the preconditions of these

operations.

We represent a barrier synchronisation by a list of Sync objects. As a state-space reduction strategy, we require this list to be ordered by the workers' identities. The helper method getSyncs returns the list that would correspond to a correct barrier synchronisation in the current state. The sync method assumes such a correct synchronisation, and gives the expected results.

Note that equality over Spec objects corresponds to value equality over enrolled parameters, as required.

The matching method is then straightforward.

The suffixMatching function tests whether its argument is a possible suffix of a correct synchronisation, i.e. it is ordered by the workers' identities. Using this function allows the underlying algorithm to avoid building unordered lists, and leads to a fairly large speed-up, particularly for larger numbers of threads

The definition of a worker is straightforward: if a worker is enrolled, it attempts to synchronise with probability 0.7, and otherwise resigns; if a worker is not enrolled, it does so.

The rest of the definitions are straightforward. Note that if there are p workers, the arity of a synchronisation can be anything between 1 and p. We do not use the ASAP reduction strategy in this case, as it turns out to make checks slightly slower.

4 Interpreting errors

If a tester finds that its synchronisation object does not meet the specification, it outputs a counterexample history, together with some explanation. We discuss how to interpret that output. We present a number of examples. While none of the error histories immediately reveals the bug in the code, each does give a useful clue to the source of the error.

Below is the output from a tester run on an incorrect implementation of a synchronous channel.

```
0:   Call of Send(3)
0:   Return of () from Send(3)
1:   Call of Receive
1:   Return of 3 from Receive
Invocation 0 does not synchronise with any other completed operation.
```

The left-hand column gives an invocation number to each invocation, and identifies corresponding calls and returns. The problem displayed by this history is that invocation 0 returns immediately, and so does not synchronise with a corresponding receive.

The history below is for a faulty implementation of an exchanger.

```
0:   Call of Exchange(93)
1:   Call of Exchange(88)
2:   Call of Exchange(54)
3:   Call of Exchange(83)
2:   Return of 88 from Exchange(54)
0:   Return of 54 from Exchange(93)
1:   Return of 54 from Exchange(88)
3:   Return of 93 from Exchange(83)
Invocation 0 does not synchronise with any other operation.
```

Here, invocations 1 and 2 have correctly exchanged. However, invocation 0 has not correctly exchanged with any other invocation: it returns 54, which is the parameter of invocation 2; but invocation 2 does not return invocation 0's parameter 93. This error reveals that there is interference between the different exchanges.

A longer example, for the ABC problem, is in Figure 5. The bottom part of the figure shows possible synchronisations for the different invocations considered in isolation, i.e. where the three invocations overlap and return consistent results. For example, invocation 0 could only have synchronised with invocations 1 and 2; but invocation 5 might have synchronised with invocations 6 and 8, or invocations 9 and 8.

The error is that there is no possible synchronisation for invocations 3, 4 and 7. Invocation 3 (syncA(0), returning (4,5)) did not synchronise with any other thread: no syncB(4) returned (0,5), and no syncC(5) returned (0,4). Likewise invocations 4 and 7 (syncB(1) returning (0,2), and syncC(2) returning (0,1)) did not synchronise with any A-thread (although those two invocations were consistent with each other): no overlapping syncA(0) returned (1,2).

The erroneous history below is for an incorrect closeable channel.

```
0:   Call of Send(4)
1:   Call of Close
2:   Call of Receive
3:   Call of Receive
1:   Return of () from Close       :  unary; linearisation index 0
2:   Return of None from Receive   :  unary; linearisation index 1
3:   Return of Some(4) from Receive:  unmatched
0:   Return of false from Send(4)  :  unary; linearisation index 2
```

This is a stateful object, and so the order of synchronisations is important. The linearisation indexes on the right give a possible order for the successful synchronisations (in this case, each happens to be a unary synchronisation). These linearisation indexes give a maximal prefix of a linearisation.

The problem displayed by this history is that invocation 3 (a receive successfully returning 4) cannot be matched with a corresponding send. In particular, invocation 0 (a send of 4) found the channel closed and so failed. The problem is that the receive returned and then the close took effect, and the send reacted to this closure rather than detecting that its value had been received before the closure.

```
0:   Call of SyncA(0)
1:   Call of SyncB(1)
2:   Call of SyncC(2)
0:   Return of (1,2) from SyncA(0)
3:   Call of SyncA(0)
1:   Return of (0,2) from SyncB(1)
4:   Call of SyncB(1)
5:   Call of SyncB(4)
6:   Call of SyncA(3)
2:   Return of (0,1) from SyncC(2)
7:   Call of SyncC(2)
4:   Return of (0,2) from SyncB(1)
7:   Return of (0,1) from SyncC(2)
8:   Call of SyncC(5)
6:   Return of (4,5) from SyncA(3)
9:   Call of SyncA(3)
5:   Return of (3,5) from SyncB(4)
10:  Call of SyncB(4)
8:   Return of (3,4) from SyncC(5)
11:  Call of SyncC(5)
3:   Return of (4,5) from SyncA(0)
9:   Return of (4,5) from SyncA(3)
10:  Return of (3,5) from SyncB(4)
11:  Return of (3,4) from SyncC(5)
No candidate synchronisation for invocations 3, 4, 7.
Possible synchronisations:
0: (0, 1, 2)
1: (0, 1, 2)
2: (0, 1, 2)
3:
4:
5: (6, 5, 8), (9, 5, 8)
6: (6, 5, 8)
7:
8: (9, 10, 8), (6, 5, 8), (9, 5, 8)
9: (9, 10, 8), (9, 5, 8), (9, 10, 11)
10: (9, 10, 8), (9, 10, 11)
11: (9, 10, 11)
```

Figure 5: An erroneous history for a faulty implementation of the ABC problem.

```
0:   Call of Sync(0)
1:   Call of Sync(1)
0:   Return of 1 from Sync(0):  matched with 1; linearisation index 0
2:   Call of Sync(0)
1:   Return of 0 from Sync(1):  matched with 0; linearisation index 0
3:   Call of Sync(1)
4:   Call of Sync(2)
4:   Return of 0 from Sync(2):  matched with 2; linearisation index 1
5:   Call of Sync(2)
6:   Call of Sync(3)
2:   Return of 2 from Sync(0):  matched with 4; linearisation index 1
7:   Call of Sync(0)
5:   Return of 1 from Sync(2):  matched with 3; linearisation index 2
8:   Call of Sync(2)
3:   Return of 2 from Sync(1):  matched with 5; linearisation index 2
9:   Call of Sync(1)
6:   Return of 1 from Sync(3):  matched with 9; linearisation index 3
10:  Call of Sync(3)
9:   Return of 3 from Sync(1):  matched with 6; linearisation index 3
10:  Return of 2 from Sync(3):  unmatched
11:  Call of Sync(3)
7:   Return of 3 from Sync(0):  unmatched
11:  Return of 0 from Sync(3):  unmatched
8:   Return of 0 from Sync(2):  unmatched
```

Figure 6: An erroneous history for a faulty implementation of the one-family problem.

Figure 6 gives an example error for the one-family problem. Here, in order: invocations 0 and 1 synchronise; invocations 2 and 4 synchronise; invocations 3 and 5 synchronise; then invocations 9 and 6 synchronise. But then the return of 2 from Sync(3) is erroneous, because the concurrent call of Sync(2) (invocation 8) does not return 3.

Similarly, if a tester finds a failure of synchronisation progressibility, it displays the history. Below is such a history for a faulty synchronous channel.

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

Here executions 0 and 1 have successfully synchronised. However, executions 2 and 3 have failed to synchronise when they should have done.

References

[LL25] Jonathan Lawrence and Gavin Lowe. Synchronisation: Specification and testing. Submitted for publication, 2025.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. Artima, 2008.

A Example input scripts

In this appendix we give example input scripts referred to in the body of the manual.

```
object ExchangerTester extends Tester{
  /** Representation of operations within the log. */
  case class Exchange(x: Int)

  /** Specification object. */
  object Spec{
    def sync(x: Int, y: Int) = (y, x)
  }

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. */
  def matching: PartialFunction[(Exchange,Exchange), (Any,Any)] = {
    case (Exchange(x), Exchange(y)) => Spec.sync(x, y)
  }

  /** A worker. Each worker performs a single invocation, to avoid
    * deadlocks. */
  def worker(exchanger: ExchangerT[Int])(me: Int, log: HistoryLog[Exchange]) = {
    val x = Random.nextInt(100)
    log(me, exchanger.exchange(x), Exchange(x))
  }

  /** Do a single test. */
  def doTest = {
    val exchanger: ExchangerT[Int] = new Exchanger[Int]
    val tester = new HomogeneousBinaryStatelessTester[Exchange](
      worker(exchanger), 8, matching)
    tester()
  }

  def main(args: Array[String]): Unit = runTests(5000)
}
```

Figure 7: A testing script for an exchanger, illustrating the HomogeneousBinaryStatelessTester.

```
object ABCTester extends Tester{
  // Representation of operations within the log
  trait Op
  case class SyncA(id: Int) extends Op
  case class SyncB(id: Int) extends Op
  case class SyncC(id: Int) extends Op

  // The result type of an invocation.
  type IntPair = (Int,Int)

  /** The specification class. */
  object Spec{
    // Each of a, b, c get the identities of the other two
    def sync(a: Int, b: Int, c: Int) = List((b,c), (a,c), (a,b))
  }

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. */
  def matching: PartialFunction[List[Op], List[IntPair]] = {
    case List(SyncA(a), SyncB(b), SyncC(c)) => Spec.sync(a, b, c)
  }

  /** A worker with identity me. */
  def worker(abc: ABCT[Int,Int,Int])(me: Int, log: HistoryLog[Op]) = {
    for(i <- 0 until 4){
      if(me%3 == 0) log(me, abc.syncA(me), SyncA(me))
      else if(me%3 == 1) log(me, abc.syncB(me), SyncB(me))
      else log(me, abc.syncC(me), SyncC(me))
    }
  }

  def doTest = {
    val abc: ABCT[Int,Int,Int] = new ABC[Int,Int,Int]
    val tester = new StatelessTester[Op](worker(abc), 6, List(3), matching)
    tester()
  }

  def main(args: Array[String]) = runTests(1000)
}
```

Figure 8: A testing script for a the ABC problem, illustrating the StatelessTester.

```
object TimeoutChannelTester extends Tester{
  /** Representation of an operation in the log. */
  trait Op
  case class Send(x: Int) extends Op
  case object Receive extends Op

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. */
  def matching: PartialFunction[List[Op], List[Any]] = {
    case List(Send(x)) => List(false) // Unsuccessful send.
    case List(Receive) => List(None) // Unsuccessful receive.
    case List(Send(x), Receive) => List(true, Some(x)) // Synchronisation.
  }

  /** A worker. */
  def worker(chan: TimeoutChannelT[Int])(me: Int, log: HistoryLog[Op]) = {
    for(i <- 0 until iters){
      // Delay to ensure a mix of successul and unsuccessful invocations.
      Thread.sleep(Random.nextInt(1))
      if(Random.nextInt(2) == 0){
        val x = Random.nextInt(20)
        log(me, chan.sendWithin(x, 1+Random.nextInt(1)), Send(x))
      }
      else // receive
        log(me, chan.receiveWithin(1+Random.nextInt(1), Receive)
    }
  }

  /** Run a single test. */
  def doTest = {
    val chan: TimeoutChannelT[Int] = new TimeoutChannel[Int]
    val tester = new StatelessTester[Op](worker(chan), 4, List(1,2), matching)
    tester()
  }

  def main(args: Array[String]): Unit = runTests(1000)
}
```

Figure 9: A testing script for a timeout channel, illustrating the StatelessTester.

```
object BarrierTester extends Tester{
  /** The number of threads involved in each synchronisation. */
  var n = 4

  /** Representation of an operation in the log. */
  case class Sync(id: Int)

  /** Is syncs sorted by id's? */
  def isSorted(syncs: List[Sync]): Boolean =
    syncs.length <= 1 || syncs(0).id < syncs(1).id && isSorted(syncs.tail)

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. Any n invocations can
    * synchronise, and all should receive the unit value. We require the id's to
    * be in increasing order, to reduce the number of cases by a factor of n!. */
  def matching: PartialFunction[List[Sync], List[Unit]] = {
    case syncs if syncs.length == n && isSorted(syncs) => List.fill(n)(())
  }

  /** A worker, which calls barrier.sync. */
  def worker(barrier: BarrierT)(me: Int, log: HistoryLog[Sync]) = {
    for(i <- 0 until 4) log(me, barrier.sync(me), Sync(me))
  }

  /** Is ops a suffix of a possible synchronisation? I.e. are the ids fields
    * of the form [k..n) for some k? */
  def suffixMatching(ops: List[Sync]) : Boolean = {
    assert(ops.nonEmpty); var k = ops.head.id; var ops1 = ops.tail
    while(ops1.nonEmpty && ops1.head.id == k+1){
      k = ops1.head.id; ops1 = ops1.tail
    }
    ops1.isEmpty && k == n-1
  }

  /** Run a single test. */
  def doTest = {
    val barrier: BarrierT[Int] = new Barrier(n)
    val tester = new StatelessTester[Sync](
      worker(barrier), n, List(n), matching, suffixMatching, false)
    tester()
  }

  def main(args: Array[String]) = runTests(1000)
}
```

Figure 10: A testing script for a barrier synchronisation object, illustrating the StatelessTester.

```
object OneFamilyTester extends Tester{
  /** Number of threads to run. */
  var n = 4

  /** Representation of operations within the log. */
  case class Sync(id: Int)

  type BitMap = Array[Array[Boolean]]

  /** The specification class. bitMap shows which threads have already
    * synchronised. */
  class Spec(val bitMap: BitMap){
    def sync(a: Int, b: Int): (Spec, (Int, Int)) = {
      // These two must not have sync'ed before
      require(!bitMap(a)(b) && !bitMap(b)(a))
      val newBitMap = bitMap.map(_.clone) // Create updated bitmap.
      newBitMap(a)(b) = true; newBitMap(b)(a) = true
      (new Spec(newBitMap), (b,a))
    }

    override def equals(that: Any) = that match{
      case s: Spec =>
        (0 until n).forall(a => bitMap(a).sameElements(s.bitMap(a)))
    }

    override def hashCode = {
      var h = 0
      for(a <- 0 until n; b <- 0 until n){
        h = h << 1; if(bitMap(a)(b)) h += 1
      }
      h
    }
  } // end of Spec

  // ...
}
```

Figure 11: A testing script for a the one family problem, illustrating the BinaryStatefulTester (part 1).

```
/** Mapping showing how synchronisations of concrete operations correspond
  * to operations of the specification object. */
def matching(spec: Spec): PartialFunction[(Sync,Sync), (Spec,(Any,Any))] = {
  case (Sync(a), Sync(b)) => spec.sync(a, b)
}

/** A worker. */
def worker(of: OneFamilyT)(me: Int, log: HistoryLog[Sync]) = {
  for(_ <- 0 until n-1) log(me, of.sync(me), Sync(me))
}

/** Do a single test. */
def doTest = {
  val of: OneFamilyT = new OneFamily(n)
  val spec = new Spec(Array.ofDim[Boolean](n,n))
  val bst = new BinaryStatefulTester[Sync,Spec](
    worker(of), n, matching, spec, true)
  bst()
  }
}

def main(args: Array[String]) = runTests(5000)
```

Figure 12: A testing script for a the one-family problem, illustrating the BinaryStatefulTester (part 2).

```
object CloseableChanTester extends Tester{
  /** Representation of an operation in the log. */
  trait Op
  case class Send(x: Int) extends Op
  case object Receive extends Op
  case object Close extends Op

  /** Try to send x on chan, catching a ClosedException. Return true if
    * successful. */
  @inline private def trySend(chan: CloseableChan[Int], x: Int): Boolean =
    try{ chan.send(x); true } catch { case _: ClosedException => false }

  /** Try to receive on chan, catching a ClosedException. Optionally return the
    * value received. */
  @inline private def tryReceive(chan: CloseableChan[Int]): Option[Int] =
    try{ Some(chan.receive()) } catch { case _: ClosedException => None }

  /** Mapping showing how synchronisations of concrete operations correspond
    * to operations of the specification object. Here the specification object
    * is simply a Boolean, indicating whether the channel is closed. */
  def matching(closed: Boolean): PartialFunction[List[Op], (Boolean, List[Any])] = {
    case List(Send(x), Receive) if !closed => (closed, List(true, Some(x)))
    case List(Send(x)) if closed => (closed, List(false))
    case List(Receive) if closed => (closed, List(None))
    case List(Close) => (true, List(()))
  }

  // ...
}
```

Figure 13: A testing script for a closeable channel, illustrating the StatefulTester (part 1).

```
/** A worker. Workers close the channel with probability 0.05;
  * otherwise, workers with an even identity send; workers with an odd
  * identity receive. */
def worker(chan: CloseableChan[Int])(me: Int, log: HistoryLog[Op]) = {
  for(i <- 0 until iters){
    if(Random.nextFloat() < 0.05) log(me, chan.close, Close)
    else if(me%2 == 0){
      val x = Random.nextInt(MaxVal); log(me, trySend(chan, x), Send(x))
    }
    else log(me, tryReceive(chan), Receive)
  }
}

/** Run a single test. */
def doTest = {
  val chan: CloseableChan[Int] = new CloseableSyncChan[Int]
  val tester = new StatefulTester[Op,Boolean](
    worker(chan), 4, List(1,2), matching, spec0 = true, doASAP = false)
  tester()
}

def main(args: Array[String]) = runTests(5000)
```

Figure 14: A testing script for a closeable channel, illustrating the StatefulTester (part 2).

```
object ResignableBarrierTester{
  /** The type of barriers. */
  type Barrier = ResignableBarrierT[Int]

  /** Operations. */
  abstract class Op
  case class Enrol(id: Int) extends Op
  case class Resign(id: Int) extends Op
  case class Sync(id: Int) extends Op

  /** Type of the set of threads currently enrolled. */
  type Enrolled = HashSet[Int]

  /** The specification class.
    * @param enrolled the set of threads currently enrolled. */
  case class Spec(enrolled: Enrolled){
    /** The effect of an enrol invocation. */
    def enrol(id: Int) = {
      assert(!enrolled.contains(id)); (new Spec(enrolled + id), List(()))
    }

    /** The effect of a resign invocation. */
    def resign(id: Int) = {
      assert(enrolled.contains(id)); (new Spec(enrolled - id), List(()))
    }

    /** The list of Sync objects that would correspond to a barrier
      * synchronisation in the current state. */
    def getSyncs: List[Sync] = enrolled.toList.sorted.map(Sync)

    /** The effect of a barrier synchronisation. Pre: syncs = getSyncs. */
    def sync(syncs: List[Op]) = {
      val n = enrolled.size; assert(syncs.length == n)
      (this, List.fill(n)(()))
    }
  } // end of Spec

  ...
}
```

Figure 15: A testing script for a resignable barrier, illustrating the StatefulTester (part 1).

```
/** Partial function showing how a list of invocations can synchronise, and
  * returning the expected next state and list of return values. */
def matching(spec: Spec): PartialFunction[List[Op], (Spec, List[Unit])] = {
  ops => ops match{
    case List(Enrol(id)) => spec.enrol(id)
    case List(Resign(id)) => spec.resign(id)
    case syncs if syncs == spec.getSyncs => spec.sync(syncs)
      // Note: the above "if" clause tests the precondition for this being a
      // valid barrier synchronisation.
  }
}

/** Does `ops` represent a suffix of a possible synchronisation (including the
  * unary operations)? */
def suffixMatching(ops: List[Op]) = ops.length <= 1 || suffixMatching1(ops, 0)

/** Is ops a list of Sync values, with increasing values of id, all at least
  * n? */
def suffixMatching1(ops: List[Op], n: Int): Boolean =
  if(ops.isEmpty) true
  else ops.head match{
    case Sync(m) if m >= n => suffixMatching1(ops.tail, m+1)
    case _ => false
  }

/** A worker. */
def worker(barrier: Barrier)(me: Int, log: HistoryLog[Op]) = {
  var enrolled = false
  for(i <- 0 until 4){
    if(enrolled){
      if(Random.nextFloat() < 0.7) log(me, barrier.sync(me), Sync(me))
      else{ log(me, barrier.resign(me), Resign(me)); enrolled = false }
    }
    else{ log(me, barrier.enrol(me), Enrol(me)); enrolled = true }
  }
  // Resign at the end, to avoid deadlocks
  if(enrolled) log(me, barrier.resign(me), Resign(me))
}
```

Figure 16: A testing script for a resignable barrier, illustrating the StatefulTester (part 2).

```
  var p = 4 // # workers

  /** Do a single test. */
  def doTest() = {
    val barrier = new ResignableBarrier[Int](faulty)
    val spec = new Spec(new Enrolled)
    val tester = new StatefulTester[Op,Spec](
      worker(barrier), p, (1 to p).toList, matching, suffixMatching, spec, false)
    if(!tester()) sys.exit()
  }

  def main(args: Array[String]) = runTests(10000)
}
```

Figure 17: A testing script for a resignable barrier, illustrating the StatefulTester (part 3).