Testing Synchronisation Objects: User Manual

Gavin Lowe

July 30, 2024

1 Introduction

This user manual describes a package for testing synchronisation objects. It should be read in conjunction with the paper *Understanding Synchronisation* [**?**].

worker threads logging.
pragmatics – shortish runs
assume familiarity with Scala
install, using.
Overview, roadmap
Possible order: Example script;
Interpreting errors.

Variations: Progress Different testing algorithms – after progress; Stateful; Progress checks

Interpreting errors; Fine details? ; Tactics (params in Tester). Avoid deadlocks – single invocation.

2 An example script

We introduce the technique of writing testing scripts via a simple example. Consider a class SyncChan that implements a synchronous channel, extending a trait Chan, defined as follows.

```
trait Chan[A]{
  def !(x: A): Unit
  def ?(u:Unit): A
}

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

The intention is that an invocation !x sends the value x, and syncronises with an invocation ?(), 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 operations. 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, with 20 invocations in each case. The code log(me, c?(), 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, each worker with an odd identity sends a random value x, with suitable logging, via the code log(me, c!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 these channels. 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

```
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 20)
      if(me%2 == 0) log(me, c?(), Receive)
      else{ val x = Random.nextInt(100); log(me, c!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, ())
  }

  /** 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.

(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.

The 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 5000 tests.

2.1 Pragmatics

When running a tester, it is useful to see some indication that it is making progress, and in particular that the threads working on the synchronisation object have not deadlocked. One way is to print something on the screen occasionally, say printing a dot every 100 tests.

The trait synchronisationTester.Tester in the distribution, outlined below, encapsulates some boiler-plate code often used in testers.

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

  /** Number of iterations per worker thread. */
  var iters = 20

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

  /** 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) = { ... }
}
```

Objects that extend Tester must provide the doTest method, but can inherit the runTest 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 to show progress, as described above, by default, one dot every 100 runs (if the progressCheck flag is set, it prints a dot after every test; this is intended for use with progress checks, described below, which tend to be slower).

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)
    else{ val x = Random.nextInt(100); log(me, c!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

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

3.1 Stateless testers

The class HomogeneousBinaryStatelessTester is for testing binary homogeneous stateless synchronisation objects. Its signature is in Figure 2: it takes the same parameters as in the heterogeneous case; and likewise the apply function takes the same optional parameter.

Figure 4 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 thread 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 if one thread has two remaining invocations, but all the others have terminated.

The class StatelessTester can test arbitrary stateless synchronisation testers. Its signature is in Figure 2. 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 5 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

```
/** 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 2: Signatures for the stateless testers.

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 6 gives another example, for a timeout channel. This example also 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 7 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 we can use the isSorted function. 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 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 algorithm for testing binary stateful synchronisation objects is encapsulated in the class BinaryStatefulTester whose signature is in Figure 3. 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 synchronisation object. Such 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.

```
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 3: Signatures for the stateful testers.

- 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 for the specification object. 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 should be employed. Experience suggests that this is normally beneficial.

Figures 8 and 9 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 being allowed within matching, and dispensed with the **requires** check, 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.

Testing for general stateful synchronisation objects is encapsulated in the class StatefulTester, described in Figure 3. 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 parameter arities records the list of arities of synchronisations, again as in StatelessTester. The optional parameter suffixMatching should give true if its argument is a non-empty suffix of a list of invocations that could synchronise; this can be useful to optimise testers, as we describe later.

As an example, a tester for a closeable channel is given in Figures 10 and 11. 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 12–14. Recall that this 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 specification.

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

worker

.....

Enrolled workers attempt sync with probability 0.7.

ASAP seems slower

3.3 Interpreting errors

We now discuss how to interpret the output from a tester when an error is found[1]

```
0:   Call of Send(2)
0:   Return of () from Send(2)
1:   Call of Receive
2:   Call of Send(61)
1:   Return of 2 from Receive
2:   Return of () from Send(61)
3:   Call of Receive
3:   Return of 61 from Receive
Invocation 0 does not synchronise with any other
completed operation.
```

A Example input scripts

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

[1] An error like this `scala synchronisationTester.ChanTester --faulty --iters 2 -p 2`

```
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), 20, matching)
    tester()
  }

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

Figure 4: 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 20){
      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 5: 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)
    case List(Receive) => List(None)
    case List(Send(x), Receive) => List(true, Some(x))
  }

  /** 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 6: 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 20) log(me, barrier.sync(me), Sync(me))
  }

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

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

Figure 7: A testing script for a barrier synchronisation object, illustrating the StatelessTester.

```
object OneFamilyTester extends Tester{
  /** Number of threads to run. */
  var n = 5

  /** 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 = Array.ofDim[Boolean](n,n)){
    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 8: 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()
    val bst = new BinaryStatefulTester[Sync,Spec](
      worker(of), n, matching, spec, true)
    bst()
    }
  }

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

Figure 9: 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!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?()) } 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 10: 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 11: 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 12: 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 10){
    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 13: 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 14: A testing script for a resignable barrier, illustrating the StatefulTester (part 3).