4yp

Thomas Aston

May 10, 2024

Contents

1 Introduction

In a world reliant on computer systems, the correctness of those systems are vital. Indeed, simple programming errors can lead to major incidents; examples of these include an automated trader losing $460 million[7] and the inaugural Ariane 5 flight breaking up after launch due to an overflow error[5].

In order to achieve better performance, concurrency can often be introduced to improve the performance programs or systems - especially those with semi-independent tasks or components. Concurrent systems, be this on a single computer or distributed across a network, achieve these performance improvements at the cost of additional complexity as the design now needs to consider the interactions and exchange of data between threads. Each of different possible interactions between threads could potentially lead to a *race condition* in a poorly designed systems; this is where two non-independent actions can occur in an order which produces an incorrect or unwanted outcome. Race conditions can be very damaging in practice - the Therac-25 radiation therapy machine killed three of its patients by radiation overexposure as a result of a race condition[21]. This was caused by the operator quickly changing from the high radiation beam mode to select the lower radiation beam instead; the race condition resulted in the machine erroneously still using the high radiation beam instead. This highlights the importance of thorough system validation; had more through system validation been completed, these deaths would have likely been avoided[2].

There are two main approaches to developing correct software: testing and verification. Though thorough unit testing can be effective in minimising software bugs[12], this form of testing is significantly less effective in concurrent contexts. Testing can only establish the presence of a bug, not the absence of any; this can be somewhat addressed by writing exhaustive tests to cover every possible edge case, however this imposes severe restrictions on the complexity of such systems. Writing exhaustive tests is near impossible for sufficiently complex concurrent systems. This is predominantly due to the sheer number of different interactions between independent threads: considering all possible interactions and testing them effectively are both challenging tasks. *Linearizability* testing is an effective alternative approach, although this this relies on the random testing of edge cases; clearly this is not exhaustive either[20].

By contrast, formal verification can be used to show that systems satisfy some desired properties[3]. This, however, is a complex process and it is often impossible to model check the complete behaviour of a large system simply due to the size of the resulting state space. We instead focus on modelling the concurrent interactions between threads; if the interactions between threads behave as expected then system validation can be reduced to checking the behaviour of the individual, sequential, threads. Focusing on these interactions allows us to decrease the size of a system to the extent where it is now feasible to model check these behaviours.

We choose to use the process algebra Communicating Sequential Processes (CSP) as our tool for modelling these interactions; CSP is a language for describing processes that can interact both with their environment and other parallel processes. As CSP is covered in the Part B Concurrency course[8] we assume familiarity with CSP as described in [23].

1.1 Contributions

The contributions of this project are as follows:

- We produce a CSP model of a 2-thread synchronisation object and how they can be organised into an arbitrary n-thread barrier synchronisation object. We then use the CSP model to prove the correctness of the barrier synchronisation object.
- We model both the standard JVM monitor and the SCL[17] implementation of a monitor, proving that the SCL monitor provides the same mutual exclusion and correctness properties as the JVM monitor but without the same unwanted behaviours.
- We examine a variety of locks from [22] and a number of desirable properties of locks, using CSP refinement checks to determine which locks satisfy. We also discuss the feasibility of modelling infinite properties using CSP.

2 Related Work

Current literature highlights a few main approaches to formal verification of software. Such verification can either be done via verification tests written during development, through automated translation of code after development or alternatively through modelling by hand.

Writing verification proofs throughout development allows developers to ensure that their code meets their specification during development. An example of this approach is AWS, who use formal verification methods in code development in order to ensure that their code always meets its specifications [1, 25]. This process involve writing a specification for each function in the form of a number of pre- and post-conditions, with these properties being validated. Though similar in style to standard unit-tests, these are actual proofs that are continually checked during development, with routine checking of these proofs indicating the correctness (or otherwise) of the code. This approach requires more effort during the development and writing process, the ability to detect system design issues during implementation is significantly more helpful than via system testing at the end of development (or indeed in production). To aid with this, some languages such as Dafny[14] have been developed with built in model checkers as an extension of this approach.

Automated code translation is similar, but tends to focus on more generic properties such as detecting deadlocks. NASA developed the Java PathFinder (JPF) in 1999 as one of the first examples of automated verification of code [4]. This was able to detect and alert on deadlocks, unhandled exceptions and assertions, however it was not able to check for correctness against some specification processes/models [10]. This limits its utility significantly; code not deadlocking does not imply correctness. More modern tools such as Stainless are able to verify some further system properties when given some additional information by the developer[**C4DT**].

By contrast, verification of code by hand using some modelling language only requires verifying a final design or implementation. This can be beneficial as the greater insight gained from

We instead choose to focus on the formal verification of code by hand. This allows us to make use of our insights to optimise the models developed, resulting in a smaller model to nverify and hence allowing us to verify the system for larger numbers of threads or other parameter limitations.

We choose to use CSP for this task. CSP is very suitable for modelling concurrent systems with tightly restricted communications between threads [13] and allows for natural modeling of system behaviour. Through this, we can effectively use CSP and FDR to model and then check a range of primitives and desired properties. This let to significant verification results prviously, most prominently being Lowe's detection of a man-in-the-middle attack on the Needham-Schroeder protocol[15]. It has also been used previously to model software running on the International Space Station, proving that such systems were free of deadlocks[6].

Indeed, CSP has also been successfully used to find bugs in concurrency primitives. Lowe used CSP to model an implementation of a concurrent channel, with FDR returning that the implementation was not deadlock free [18]. This bug was a very niche edge case that required a trace of 37 separate events which had yet to be spotted by hand. The produced trace allowed for a straightforward fix to the code to made to remove the deadlock; this makes it well suited to our needs here as we can both accurately model concurrent datatypes and then easily interpret any resulting error traces.

There are three main styles of concurrent programming as highlighted in [22]: lock-based, message passing and dataype-based concurrent programming. There exists literature on formal verification of the latter two; Lowe has previously proved the correctness of a lock-free queue[16] and also an implementation of a generalised alt operator

[19]. There is also more general work on the verification of lock-free algorithms, such as Schellhorn and Bäumler [24]. Their work uses an extended form of linear temporal logic (LTL) and the rely-guarantee paradigm (introduced by [11]) to prove linearizability and lock-freeness properties.

By contrast, there is an lack of research into lock-based concurrency primitives, we therefore focus on this area.

https://link.springer.com/chapter/10.1007/978-3-642-17511-4_20

3 Modelling and analysing implementations of locks

In this section we will analyse a number of different lock implementations. The primary purpose of locks is to provide *mutual exclusion* between threads; that is to avoid two threads from operating concurrently on the same section of code, referred to as the *critical region*. A good lock should also fulfil some *liveness* requirements, essentially that something good will eventually happen. When devising liveness requirements we assume that no thread wil hold the lock indefinitely; otherwise most reasonable liveness requirements can be invalidated by a thread that gains the lock and never releases it. *Deadlock freedom* is a liveness requirement that if some thread is attempting to acquire the lock then some thread will eventually succeed in acquiring the lock, unless a thread holds the lock indefinitely. *Starvation freedom* is a liveness requirement that any thread that tries to gain the lock will eventually succeed; by contrast deadlock freedom allows one thread to never obtain the lock as long as others complete an infinite number of critical sections. Other requirements/useful properties of locks will be explored later.

```
trait Lock{
  /** Acquire the Lock. */
  def lock : Unit
  /** Release the Lock. */
  def unlock : Unit
  ...
}
```

Figure 1: A Scala interface for a simple lock

3.1 External interfaces

The most straightforward interfaces of a lock can be seen in Figure 1. This provides a lock function for a thread to attempt to gain the lock (blocking if some other thread currently hold the lock) and an unlock function for a thread to release the lock.

When a thread t uses a lock l with there are four main events of importance to model in CSP:

- callLock.l.t : The thread calls the lock function;
- lockAcquired.l.t : The thread exits the lock function, now holding the lock;
- lockReleased.l.t : The thread has called the unlock function and the unlock function has been executed to the point where a thread can now reobtain the lock
- end.t : The thread will make no further calls to the lock $\boxed{\text{Necessary?}}$

  $\boxed{\text{Worth talking about linearization?}}$

We can now specify ideal properties of locks using these channels:

- Mutual Exclusion: This specifies that at most one thread may be in its critical section at any one time; i.e. that once thread A obtains the lock, no other thread can obtain the lock until thread A unlocks. We can therefore deduce that a lock l with model X satisfies the trace refinement:

```
Mutex = lockAcquired.l?t → lockReleased.l.t → Mutex
Mutex ⊑_T X \ (Σ − [|lockAcquired.l, lockReleased.l|])
```

- Deadlock Freedom: This specifies that if some thread attempts to acquire the lock then some thread will succeed in acquiring the lock[9]. This does allow a CSP deadlock $\boxed{\text{Need to explain earlier}}$ if no thread is attempting to acquire the lock, but only if the following holds: $\boxed{\text{format this}}$

  $\forall (s, ref) \in failures(P) \,.\, \#(s \downarrow callLock) = \#(s \downarrow lockAcquired) \Rightarrow ref \subset \Sigma$

  This can be captured by the following failures refinement on lock l with the set of all threads called ThreadID. This process can non-deterministically deadlock when no threads are attempting to obtain the lock and otherwise ensures that if a thread attempts to acquire the lock then some thread obtains the lock

```
AcquireLock(l, ts, TS) =
  end?t:(diff (TS, ts)) → AcquireLock(l, ts, diff (TS, {t}))
  □ callLock.l?t:(diff (TS, ts)) → AcquireLock(l, union(ts, {t}), TS)
  □ lockAcquired.l?t:ts → AcquireLock(l, diff (ts, {t}), TS)

AcquireLock(l, {}, ThreadID) ⊑_F
    X \ (Σ − {|callLock.l.t, lockAcquired.l, end|})

```

  AcquireLock takes three parameters: l is the identity of the lock, ts is the set of threads currently which have communicated a callLock but haven't yet communicated a following lockAcquired event. Finally, TS is the set of all live threads; the individual threads can communicate end throughout whenever they aren't attempting to acquire the lock. In our refinement check, we will assume that no lock events have occurred prior; ie. no threads have attempted to acquire the lock or terminated yet.

- Starvation Freedom: Every thread that attempts to acquire the lock eventually succeeds $\boxed{\text{Definition of starvation freedom}}$

3.2 A simple lock specification

$\boxed{\text{utility of this? Should cut down}}$

We define a specification for a lock

Figure 2 shows a simple trace specification for a lock, where l is the identity of the lock, TS is the set of all threads and ts is the set of all threads that have communicated a callLock.l.t, but haven't yet followed that by a lockAcquired.l.t. At any point, lock can be called by a thread that does not currently hold the lock and that hasn't called the lock since it last held the lock (ie. a thread cannot call the lock twice without holding it in between). If some thread X holds the lock then it can unlock whenever, with the; likewise if no thread hold the lock, then any thread that has called lock but not obtained the lock yet can obtain the lock.

This specification has the required property of mutual exclusion - once a thread has performed a lockAcquired.l.t, no other threads can perform a lockAcquired.l.t' until after the original thread releases the lock via lockAcquired.l.t. It also specifies deadlock-freedom since it can always communicate a callLock unless either ts == TS (in which causes some thread can communicate a lockAcquired, followed later by a lockReleased) or TS = {} (where all threads have 'terminated' via exit and hence is deadlock-free since no threads will attempt to obtain the lock). Livelock-freeness is also satisfied as all actions performed make progress towards obtaining the lock or releasing the lock once it is held.

This specification process for locks will be very useful later as if we can show trace-equivalence between this specification and some implementation of a lock over {callLock, lockAcquired, lockReleased, end} we can use the specification in more complex systems, reducing the size of the systems produced by FDR and hence allowing us to test larger

```
LockSpec(l, ts, TS) =
  end?t:diff (TS, ts) → LockSpec(l, ts, diff (TS, t))
  □ callLock.l?t:(diff (TS, ts)) → LockSpec(l, union(ts, {t}), TS)
  □ lockAcquired.l?t:ts → LockSpecObtained(t, l, ts, TS)
LockSpecObtained(t, l, ts, TS) =
  end?t2:diff (TS, union(ts, t)) → LockSpecObtained(t, l, ts, diff (TS, t2))
  □ callLock.l?t2:(diff (TS, union(ts, {t}))) →
      LockSpecObtained(t, l, union(ts, {t2}), TS)
  □ lockReleased.l.t → LockSpec(l, diff(ts, {t}), TS)
```

Figure 2: A non-starvation-free trace specification for a lock

cases than would otherwise be possible.

3.3 Test-and-Set Lock

The Test-and-Set (TAS) lock implementation is based on using an AtomicBoolean called state to capture whether the lock is currently held. A value of true meaning that some thread holds the lock and false meaning that the lock is currently free. The AtomicBoolean, has atomic get and set operations to read and write values respectively. In the TAS lock we also use the getAndSet operation which atomically sets the Boolean to a new value and returns the old value. The full Scala code can be seen in Figure 3. When a thread attempts to obtain the lock, it performs a state.getAndSet(true); a getAndSet(true) that returns false can be treated as having gained the lock, whereas a true indicates that some other thread already holds the lock. To release the lock a set(false) is done to mark the lock as available to other threads.

3.3.1 Modelling with CSP

Firstly, in order to model the TAS lock, we need a process that acts as an AtomicBoolean to model the state variable. Figure 4 shows a process Var than takes an initial value, and channels get, set : ThreadID.A and getAndSet : ThreadID.A.A for some arbitrary type

```
import java.util.concurrent.atomic.AtomicBoolean

/** A lock based upon the test−and−set operation
  * Based on Herlihy & Shavit, Chapter 7. */
class TASLock extends Lock{
  /** The state of the lock: true represents locked */
  private val state = new AtomicBoolean(false)

  /** Acquire the Lock */
  def lock = while(state.getAndSet(true)){ }

  /** Release the Lock */
  def unlock = state.set(false)
}

```

Figure 3: Test-and-set lock from [22] Need to figure out figure placement

```
Var(value, get, set, getAndSet) =
  get?_!value → Var(value, get, set, getAndSet)
  □ set?_?value' → Var(value', get, set, getAndSet)
  □ getAndSet?_!value?value' → Var(value', get, set, getAndSet)

```

Figure 4: A process encapsulating an Atomic variable with get, set and getAndSet operations

A.

We can therefore represent the state variable from the Scala implementation by the following CSP:

```
channel get, set: ThreadID . Bool
channel getAndSet: ThreadID . Bool . Bool
State = Var(false, get, set, getAndSet)
InternalChannels = {|get, set, getAndSet|}
```

A communication on any of the channels is equivalent to a thread calling that operation in Scala. We use false to indicate that no thread holds the lock initially.

We can then model the operations of the lock itself. The Unlock procedure is quite trivial, simply setting the state to false and then terminating.

```
Unlock(t) = setState.t! False → SKIP -- def unlock = state.set(false)
```

The Lock procedure is also trivial, with the thread just communicating over getAndSet. The procedure terminates once the getAndSet communicates that the original value of the statevariable was false; a getAndSet.t.False.True event in a trace can be linearized as the point at which the thread t obtains the lock.

```
-- while(state.getAndSet(true)){ }
Lock(t) = getAndSet.t?v!True → if v == False then SKIP
                                else Lock(t)
```

We model the threads that are attempting to obtain the lock by a process Thread(x), where x is the identity of the thead. Each thread can either choose to either terminate or obtain the lock, release the lock and repeat its choice; we use external choice here so that we can regulate the behaviour of the threads when analysing the lock's properties.

```
Thread(t) = callLock.L.0.t → Lock(t); Unlock(t); Thread(t)
            □ end.t → SKIP
```

Finally, we construct the overall system as shown below. We first synchronise all the threads over the get, set and getAndSet channels with the State process. Since getAndSet.t.False.True corresponds to when a thread obtains the lock and setState.t.False corresponds to a thread releasing the lock, we can rename these communications to lockAcquired.L.0.t and lockReleased.L.0.t respectively to produce ActualSystemR. Finally, to obtain a system that only visibly communicates the four previously identified events, we hide the internal channels of the lock to produce ActualSystemRExtDiv.

```
-- All initially  do not hold the lock
AllThreads = ||| t : ThreadID • Thread(t)
-- Allow all threads to peform actions on the state variable
ActualSystem = (AllThreads [|InternalChannels|] State)
-- Rename lock acquisition and releasing and hide internal events
ActualSystemR = (ActualSystem
                    [[getAndSet.t.False.True \ lockAcquired.L.0.t,
                    set.t.False \ lockReleased.L.0.t | t <- ThreadID]])
  ActualSystemRExtDiv = ActualSystemR \ InternalChannels
```

3.3.2 Analysis

We will firstly examine whether this model fulfils the properties defined previously. The mutual exclusion and deadlock freedom tests from section 3.1 pass and the model does not diverge before it is first held; these were all expected results$\boxed{\text{Livelock?}}$. The TAS lock is also equivalent under traces with the simple lock specification earlier. However, once the lock is held, a thread attempting to obtain the lock can perform an infinite number of getAndSet operations; an example trace of this behaviour where T.0 obtains the lock follows

$\langle callLock.L.0.T.0, callLock.L.0.T.1, getAndSet.T.0.False.True\rangle\hat{}$
$\langle getAndSet.T.1.True.True\rangle^{\omega}$

This behaviour is expected; thread T.1 is trying to obtain the lock and is being blocked by T.0 which holds the lock. We will now consider the low-level implementation of the getAndSet operation and why an unbounded number of consecutive failed getAndSet operations is problematic for performance. Any getAndSet operation causes a broadcast on the shared memory bus between the processors, delaying all processors whilst also forcing each thread to invalidate the value of state from the caches, regardless of whether the value is actually changed. Since the above trace never results in thread T.1 successfully setting state, it is preferrable to use at least limit the number of getAndSet operations without unneccessarily delaying a thread from obtaining the lock. As a result, it is preferrable to use less costly get operations in order to limit the usage of getAndSet operations to situations where they are likely to change the value of the underlying lock. Since get does not change the underlying value of a variable, the read will result in at most one cache-miss per set/getAndSet on state; this is a marked improvement $\boxed{\text{Level of detail regarding memory buses, performance, caching etc}}$

3.4 Test-and-Test-and-Set Lock

The Test-and-Test-and-Set (TTAS) lock makes use of this improvement, whilst otherwise remaining very similar to the TAS lock. The sole change is to the lock function, as can be seen in Figure 5, which we then specify in our CSP model as the following:

```
Lock(t) = getState.t?s → if s = True then Lock(t)
                          else gASState.t?v!True → if v = False then SKIP
                          else delay ! t → Lock(t)
```

The TTAS lock can still produce traces with an unbound number of consecutive operations, but these are now get operations instead of getAndSet operations. Whereas the TAS lock can have an unbounded number of getAndSet operations for each time the lock is obtained, the TTAS lock has performs at most one getAndSet operation per thread when the lock becomes available. This is the case when all threads trying to obtain the lock read get.t.False before the first getAndSet is performed; all threads know the lock was not held so try to obtain it via a getAndSet, with only one of the threads succeeding. We now have a linear bound on the number of unsuccessful getAndSet operations, resulting in much more efficient usage of caching and shared memory. $\boxed{\text{Show bound using a CSP regulator function?}}$

3.5 Tree lock

$\boxed{\text{Might be worth introducing Peterson lock for starvation freedom both earlier and here?}}$

Suppose we have an implementation of a lock that works for upto n threads and now wanted to extend this to work with more threads trying to obtain a single lock. One approach to solving this problem is to arrange a number of the n thread locks into

```
class TTASLock extends Lock{
  ...
  /** Acquire the lock */
  def lock =
    do{
      while(state.get()){ } // spin until state = false
    } while(state.getAndSet(true)) // if state = true, retry
  ...
}
```

Figure 5: Test-and-test-and-set lock from [22]

```
                 L.0
               /     \
              /       \
             /         \
            /           \
          L.1           L.2
         /  \           /  \
        /    \         /    \
       /      \       /      \
      L.3     L.4    L.5     L.6
     /  |     / |    |  \    |  \
    /   |    /  |    |   \   |   \
T.3.1   | T.4.1 |    | T.5.2 |   T.6.2
     T.3.2   T.4.2    T.5.1   T.6.1
```

Figure 6: An example of the tree lock with 2 threads per lock and 8 threads

a tree structure. The threads are assigned a leaf node and, once they have obtained that lock, they progress up the tree obtaining the next lock and so on. Once the thread reaches the root of the tree and has obtained the 'root lock' and hence holds the lock; to unlock, the thread unlocks the root lock and progressively unlocks all the locks it held until it is back at the leaf lock. To consider a simple case where n = 2 and NThreads = 8 we have the following structure: Draw properly

Here, for T.5.1 to obtain the root lock L.0, it must first obtain L.5 then L.2 then it can attempt to lock L.0. If it holds L.0 then all of the other 7 threads can't enter their critical section. Once T.5.1 wants to release the lock it unlocks L.0, then unlocks L.2 then finally unlocks L.5. T.5.1 can now either terminate or try to reobtain the root lock.

3.5.1 Modelling

We model the tree structure so that the structure of the tree is independent from the implementation of the individual locks; this will allow us to examine how different the different properties of the locks affect the overall structure.

For convenience, we shall call the threads T.{y}.{z} where y is the leaf lock that the thread will first try and obtain and z is just an index over the threads starting at that leaf lock. Since the thread should be agnostic to the implementation of the lock, the thread initially either exits or calls LockTree to try and obtain the lock. Once the thread has obtained the lock, it then releases it by a call to UnlockTree. Both LockTree and UnlockTree are recursively defined, traversing their way up and down the tree respectively before terminating once they have locked the root lock or unlocked a leaf lock respectively. nextUnlock(x, y) is used as a helper function to generate the next lock to release

```
nextUnlock(x, y) = if (y / 2 > x) then nextUnlock(x, y/2) else L.y

LockTree(t, L.0) = SKIP
LockTree(T.y.z, L.x) = Lock(T.y.z, L.x); LockTree(T.x.y, L.((x−1)/2))

UnlockTree(T.y.z, L.x) = if (x = y) then Unlock(T.y.z, L.x); SKIP
                         else Unlock(T.y.z, L.y);
                              UnlockTree(T.y.z, nextUnlock(x, y))

Thread(T.y.z) = callLock.L.y.T.y.z → LockTree(T.y.z, L.y);
                                     UnlockTree(T.y.z, L.0);
                                     Thread(T.y.z)
             □ end.T.y.z → SKIP
```

We can use the lockAcquired and lockReleased events of the root L.0 lock as the points where a given thread obtains and releases the lock and the callLock events of the leaf nodes to represent when a thread tries to access the lock. To check that this lock fulfils the required properties we will hide all internal locking events; we do not care how the lock functions internally as long as it follows the required specifications. We will also rename the the three above events to appear to occur on the lock L.0. The system is therefore constructed as follows:

```
-- Initialise threads to not hold their locks
AllThreads = ||| (T.y.z) : ThreadID • Thread(T.y.z)
-- Initialise all the locks
AllLocks = ||| (L.x) : LockID • LockSystem(L.x)
-- Sychronise the threads with all the locks
TreeInternal  = (AllThreads [|LockEvents|] AllLocks)
-- Hide unwanted lock events from internal locks
TreeHidden = TreeInternal  \  Union(diff({|lockAcquired|}, {|lockAcquired.L.0|}),
                                    diff ({|unlockedLock|}, {|unlockedLock.L.0|}),
                                    diff ({|callLock|},
                                          {callLock.L.y.T.y.z | T.y.z <- ThreadID}))
-- Rename linearization points so all refer to L.0
TreeInternalRenamed = TreeHidden [[callLock.L.x.t \ callLock.L.0.t]]
```

Add analysis of beheviours with different node locks etc

4 Monitors

This is intended to be used in an earlier background section

A monitor can be used to ensure that certain operations on an object can only be performed under mutual exclusion. Here we consider the implementation of a monitor used by the Java Virtual Machine (JVM).

Mutual exclusion between function calls is provided inside the JVM via synchronized blocks. Only one thread is allowed to be active inside the synchronized blocks of an object at any point; a separate thread trying to execute a synchronized expression will have to wait for the former to release the lock before proceeding. Inside a synchronized block, a thread can also call wait() to suspend and give up the lock. This waits until a separate thread (which can now proceed) executes a notify(), which will wake the waiting thread and allow it to proceed once the notifying thread has released the lock.

It is important to note that the implementation of wait() is buggy. Sometimes a thread that has called wait() will wake up even without a notify(); this is called a *spurious wakeup*.

4.1 Modelling monitors with CSP

For our model of a monitor in CSP we have extended the JVMMonitor provided by Lowe $\boxed{\text{cite}}$. This allows us to more flexibly model using a variable number of monitors, one per object across across n objects.

The module previously provided a single monitor; this is problematic in case we have multiple objects which each require their own monitor. We instead introduce a datatype Monitor, with this type containing all possible 'objects' that could require their own monitors. The JVMMonitor module is then changed to be parameterised over some subset of Monitor. The internal channels and processes now also take some Monitor value to identify which object is being reffered to at any point.

Internally, the model of the monitor uses a one lock per MonitorID. These also have an additional parameter storing the identities of any waiting threads $\boxed{\text{full code in an appendix?}}$. So that it is a faithful model of an actual JVMMonitor we also model spurious wakeups via the spuriousWakeup channel. We therefore run the the lock process in parallel with a regulator process $\mathsf{Reg} = \mathsf{CHAOS}(\{\text{spuriousWakeup}\})$; this is used to non-deterministically allow or block spurious wakeups where appropriate. This is important as when we hide spuriousWakeup events we have that every state in FDR that allows a spuriousWakeup has a pair which blocks the spuriousWakeup. This allows us to perform refinement checks in the stable-failures model instead of just the failures-divergences model, allowing us to write more natural specification processes.

Our model of the monitor provides the following exports:

```
module JVMMonitor_(MonitorID)_
      . . .
      Reg = CHAOS({|spuriousWakeup|})
exports

      -- All events except spuriousWakeup
      events = {| acquire, release , wait, notify , notifyAll   |}

      channel spuriousWakeup : _MonitorID_ . ThreadID

      _InitialiseAll    =
         ||| mon <- MonitorID • (Unlocked(mon, {}) [| {|spuriousWakeup|} |] Reg)_

      runWith(_obj_, P) = P [| events |] (Unlocked(_obj_, {}) [| {|spuriousWakeup|} |] Reg)

      -- Interface to threads.

      -- Lock the monitor
      Lock(_obj_, t) = . . .

      -- Unlock the monitor
      Unlock(_obj_, t) = . . .

      -- Perform P under mutual exclusion
      Synchronized(_obj_, t, P) = . . .

      -- Perform P under mutual exclusion, and apply cont to the result.
      -- MutexC :: (ThreadID, ((a) -> Proc) -> Proc, (a) -> Proc) -> Proc
      SynchronizedC(_obj_, t, P, cont) = . . .

      -- Perform a wait(), and then regain the lock.
      Wait(_obj_, t) = . . .

      -- perform a notify()
      Notify(_obj_, t) = . . .

      -- perform a notifyAll()
      NotifyAll (_obj_, t) = . . .

endmodule
```

Figure 7: The interface of the JVMMonitor module; changes are highlighted in red (and underlined for B&W)

Each monitor provides Wait(o, t), Notify(o, t) and Synchronized(o, t, Proc) methods to model the equivalent functions/blocks in SCL. The Proc parameter is used to specify the process that will be run inside the Synchronized block; the intended usage of this is of the form callFunc.o.t —> Synchronized(o, t, syncFunc) where the process communicates that it is calling the model of fucntion Func before completing the rest of the function whilst holding the monitor lock.

5 The SCL Monitor

There are two main limitations to the standard JVM monitor: it suffers from spurious wakeups and does not allow targeting of notify calls. Spurious wakeups are obviously bad and are a common source of bugs where not adequately protected against. targeting of signals can also be very beneficial to the performance of a program; take for example the one-place buffer shown below in 1. Suppose we have significantly more threads wanting to get a value than put a value. Each notifyAll will awake each thread waiting on a get, even if gets are blocked by ! available. This results in the majority of the threads immediately sleeping again, adding significant overhead. A notifyAll is also required since the JVM monitor makes no guarantees as to which thread is awoke by a notify; as a result repeated notifys could potentially just wake up two thread alternatively, both of which are waiting to perform the same process.

Listing 1: Single placed buffer as an example of the inefficiency of untargeted signals

```
class OneBuff[T] {
  private var buff = null.asInstanceOf[T]
  private var available = false

  def put(x: T) = synchronized {
    while(available) wait()
    buff = x
    available = true
    notifyAll()
  }

  def get : T = synchronized {
    while(! available) wait()
    available = false
    notifyAll()
    return x // not overwritable as we still hold the lock
  }
}
```

The SCL monitor implementation solves both of these issues with a single monitor offering multiple distinct Conditions to allow for more targeted signalling. If we have two conditions, we can separate the threads attempting to get and those trying to put and have those wait on separate conditions. We can then modify the program above to only perform a single signal (the SCL monitor equivalent of notify for the JVM monitor) towards the threads that are attempting to perform the opposing function, resulting in significant efficiency gains as no threads need to immediately sleep after being woken up.

The implementation of the SCL Monitor, available on GitHub, makes use of the Java LockSupport package. Here we present a low-level model of an SCL Condition which makes use of a model of the LockSupport class.

5.1 LockSupport

Threads interact with the LockSupport module via three main events: a thread can park itself, a thread can unpark another thread and a parked thread can wake up. We therefore introduce channels park, unpark and wakeUp to represent these three synchronisations.

It is also important to note that LockSupport is also affected by spurious wakeups. We therefore add a Boolean parameter to the wakeUp channel, using false to indicate a spurious wakeup and true otherwise.

Listing 2: The CSP model of the Java LockSupport module

```
module LockSupport

  channel park: ThreadID
  channel unpark: ThreadID.ThreadID
  channel wakeUp: ThreadID.Bool

  LockSupport :: ({ThreadID}, {ThreadID}) → Proc
  LockSupport(waiting, permits) =
    if waiting = {} then LockSupport1(waiting, permits)
    else (     LockSupport1(waiting, permits)
          ⊓ wakeUp$t:waiting!false  → LockSupport(diff(waiting, {t}), permits))

  LockSupport1(waiting, permits) =
    park?t→ (
      if member(t, permits)
        then wakeUp.t.true → LockSupport(waiting, diff(permits, {t}))
      else LockSupport(union(waiting, {t}), permits) )
    □
    unpark?t?t2→ (
      if member(t2, waiting)
        then wakeUp.t2.true → LockSupport(diff(waiting, {t2}), permits)
      else LockSupport(waiting, union(permits, {t2})))


  LockSupportDet :: ({ThreadID}, {ThreadID}) → Proc
  LockSupportDet(waiting, permits) = LockSupportDet1(waiting, permits)
  LockSupportDet1(waiting, permits) = ... -- Analogous to LockSupport1

exports

  InitLockSupport = LockSupport({}, {})

  InitLockSupportDet = LockSupportDet({}, {})

  Park(t) = park.t → wakeUp.t?_ → SKIP

  Unpark(t, t') = unpark.t.t' → SKIP


endmodule
```

Internally, the module stores two sets of threads: those parked and those with permits available. A thread that is parking is either added to the waiting set if no permit is available or it is immediately re-awoken. When there is at least one thread waiting and no ongoing wakeup, the LockSupport module can nondeterministically choose to either operate as normal or to allow one of the waiting threads to spuriously wakeup.

InitLockSupportDet is defined similarly to InitLockSupport; the only change is made by removing the nondeterministic choice for a thread to spuriously wakeup. This will allows us to show some divergence results later.

5.2 The SCL monitor model

DIAGRAM

We now consider our model of the SCL monitor. This consists of three main components:

- The monitor lock; this is a simple process Lock(m) = acquire.m?t −> release.m.t −> Lock(m) which specifies that only one thread can hold the lock and that same thread must release the lock before some other thread can obtain it.
- The LockSupport module; this is as described above.
- The queue of ThreadInfo values.

5.2.1 The ThreadInfo Queue

In the Scala code, each Condition maintains a queue of ThreadInfo values which have a thread id and a variable ready indicating whether the corresponding thread has been unparked. The natural method of modelling this queue in CSP is with a series of nodes, a series of processes each corresponding to a node and some co-ordinator processes.

Since we are handling n threads and all of them can be waiting at the same time, we hence need n separate nodes, which we will represent as datatype Node = N.{0..n−1}.

Each of the nodes can be initialised by a thread, at which point it then acts as that thread's ThreadInfo object. It allows threads (including its allocated thread) to check the value of ready via a communication on the isReady channel, and for other threads to set the value of ready to true via the setReady channel. Once the parent thread has reawoken legitimately (i.e. via an Unpark not a spurious wakeup) it then releases the node; this is valid as the node must have already been dequeued and setReady, hence no further communications should happen until it is reinitialised. The ThreadInfo objects are designed to diverge whenever a communcation occurs that should not be possible in the original Scala code.

```
ThreadInfo ::  (Node) → Proc
ThreadInfo(n) =
      initialiseNode  .n?t → ThreadInfoF(n, t)
   □ isReady.n?t?t2.true → DIV
   □ setReady.n?t?t2 → DIV
ThreadInfoF(n, t) =
      isReady.n! t?t2.false  → ThreadInfoF(n, t)
   □ setReady.n!t?t2:diff (ThreadID, t)  → ThreadInfoT(n, t)
   □ initialiseNode  .n! t  → DIV
ThreadInfoT(n, t) =
      isReady.n! t?t2.true → ThreadInfoT(n, t)
   □ setReady.n!t?t2 → DIV
   □ initialiseNode  .n! t  → DIV
   □ releaseNode.n.t → ThreadInfo(n)
```

We then use a process called NodeAllocator to allocate the Nodes to threads and also to collect them when they are no longer required. We note that any thread can use any node in this model; this is analogous to the nodes being memory chunks allocated to each thread by NodeAllocator and then garbage collected once they are no longer needed.

```
NodeAllocator(ns) =
  (not(empty(ns))) &
    (initialiseNode  $n:ns?t → NodeAllocator(diff(ns, {n})))
□ releaseNode?n:ns?t → DIV
□ releaseNode?n:diff(Node, ns)?t → NodeAllocator(union(ns, {n}))
```

Finally we have the Queue processes, which model the queues maintained inside each Condition. Each Queue keeps a sequence of the nodes waiting on its condition, with each Node corresponding to its current holding thread.

```
Queue'(m, c, qs) =
     (not(null (qs))) & dequeue.m.c?t!head(qs) → Queue'(m, c, tail(qs))
  □ (null (qs)) & isEmpty.m.c?t → Queue'(m, c, qs)
  □ enqueue.m.c?t?n:diff(Node, QS) → Queue'(m, c, qs^<n>)
```

We have that each Queue' is always ready to accept an enqueue (unless all nodes are already in the queue), but will only communicate one of dequeue or isEmpty at any point in time. We note that the restriction on the values of n that can be enqueued is such that the queue is of finite length; this is required for efficient model checking in FDR.

5.3 The functions and interface of the monitor

Now we have the components of the model of the monitor, the last step is to place these processes in parallel. The majority of these processes are independent of each other; only the NodeAllocator and the ThreadInfo processes need to synchronise with each other, which occurs when a node is either initialised or released.

```
InitialiseMon (m, setC) =
  (Lock'(m) |||
   (||| c ← setC • Queue(m, c, <>)) |||
   (NodeAllocator(Node) [|{|initialiseNode, releaseNode|}|]
      (||| n ← Node • ThreadInfo(n))) |||
   InitLockSupport)
```

The version of the monitor without spurious wakeups, InitialiseMonDet(m, setC), is defined similarly but interleaved with InitLockSupportDet(m, setC) instead.

The first two processes we export as part of the interface of our monitor are runWith(P, mon, setC) and runWithDet(P, mon, setC), which each take a number of threads in P, an identity for the monitor and a set of conditions on that monitor. These processes synchronise the processes in P with the InitialiseMon(m, setC) and InitialiseMonDet(m, setC) respectively. This is to allow the threads to 'call' the various functions that act on the monitor correctly and so that mutual exclusion can be enforced as intended.

```
-- The set of events that are hidden when a thread uses the monitor
HideSet(m, setC) =
  {|park, unpark, wakeUp, enqueue.m.c, dequeue.m.c, initialiseNode, nodeThread,
    setReady, isReady, isEmpty.m.c, releaseNode | c <- setC|}

-- The set of events to synchronise on between a thread and the monitor
  SyncSet2(m, setC) = Union({{|acquire.m|}, {|release.m|}, HideSet(m, setC)})

exports

channel acquire, release : MonitorID.ThreadID
channel callAcquire , callRelease : MonitorID.ThreadID
channel callAwait , callSignalAll : MonitorID.ConditionID.ThreadID
channel callSignal : MonitorID.ConditionID.ThreadID

-- Runs the monitor with internal spurious wakeups
runWith(P, mon, setC) =
  ((P [|SyncSet(mon, setC)|]
        InitialiseMon (mon, setC)) \ HideSet(mon, setC))

-- Runs the monitor without internal spurious wakeups
runWithDet(P, mon, setC) =
  ((P [|SyncSet(mon, setC)|]
        InitialiseMonDet (mon, setC)) \ HideSet(mon, setC))

...
```

In the definitions above, SyncSet contains every event that we need to synchronise on between a series of threads and the monitor. We then hide all events except for those representing a thread acquiring and releasing the lock; this is the contents of HideSet.

We now consider the functions offered by the monitor. We have the interface given below, with each of the five processes corresponding to the function of the same name. Each process starts with a communication on the correspondingly named callX channel to indicate that the specified thread has just called that function; this makes examining any traces produced significantly simpler. We will refer to these callX communications as 'external' and all other channels as being 'internal'. $\boxed{\text{Better convention for placeholder values?}}$

```
export
  ...

  -- Operations on the monitor
  Await(mon, cnd, t) = callAwait.mon.cnd.t → ...

  Signal(mon, cnd, t) = callSignal.mon.cnd.t → ...

  SignalAll(mon, cnd, t) = callSignalAll.mon.cnd.t → ...

  Lock(mon, t) = callAcquire.mon.t → acquire.mon.t → SKIP

  Unlock(mon, t) = callRelease.mon.t → release.mon.t → SKIP

 endmodule
```

Both Lock and Unlock both only require a single internal communication (either acquiring or releasing the lock) after the thread's external communication. By contrast Await, Signal and SignalAll are more complex, so the initial external communication is followed by another process, in each case named X1. Each of these processes are natural translations of the Scala code into CSP; the main exception is using Await2 to represent the while loop in the Scala await() function.

```
  Signal1(mon, cnd, t) =
       isEmpty.mon.cnd.t → SKIP
    □ dequeue.mon.cnd.t?n → nodeThread.n?t2!t → isReady.n.t2.t?b →
         (if b then Signal1(mon, cnd, t)
          else setReady.n.t2.t → Unpark(t, t2); SKIP)

  SignalAll1(mon, cnd, t) =
       isEmpty.mon.cnd.t → SKIP
    □ dequeue.mon.cnd.t?n → nodeThread.n?t2!t → setReady.n.t2.t →
            Unpark(t, t2); SignalAll1(mon, cnd, t)

  Await1(mon, cnd, t) =
    initialiseNode ?n!t → enqueue.mon.cnd.t.n → release.mon.t → Await2(mon, cnd, t,
    n)

  Await2(mon, cnd, t, n) =
    isReady.n.t.t?b → (if b then releaseNode.n.t → acquire.mon.t → SKIP
                       else Park(t); Await2(mon, cnd, t, n))
```

5.4 Correctness

We now consider the correctness of our model. We present a specification process for a idealised monitor with conditions and perform refinement checks against it. We show that the ordering of awaits is also upheld by a separate refinement check.

We will first consider the specification process of a monitor with multiple conditions. Each of the monitor processes are parametised over the identity of the monitor, a map of ConditionID => {ThreadID} representing the set of threads waiting on each Condition, and a set of ThreadIDs that are waiting to obtain the lock. We choose to use sets of waiting threads instead of queues of waiting threads to make this a more general specification of a monitor; we consider orderings in a later test. initSet is the initial mapping of the waiting threads, with each condition mapping to an empty set. We define valuesSet as a helper function which returns a set of all the threads that are currently waiting on any condition; this allows us to restrict the specification to only allow threads that aren't waiting to obtain the lock.

We also define a new channel callSignalSpec. This is similar to the callSignal channel introduced earlier, but has an additional parameter indicating which thread is being signalled. A thread will 'signal' itself if no threads are waiting on the selected condition, otherwise it will non-deteministically signal one of the waiting threads. This extra parameter is required as there is no set operator to select a single element of a set in CSP; we use this channel instead to indicate the selected ThreadID for signalling. We rename callSignalSpec back to SCL::callSignal for when we perform the refinements.

```
initSet   = mapFromList(<(c, {}) |  c ← seq(ConditionID)>)
values (map) = Union({mapLookup(map, cnd) | cnd ← ConditionID})
channel callSignalSpec :  MonitorID . ConditionID . ThreadID . ThreadID

SpecUnlocked(m, waiting, poss)  =
      SCL::callAcquire .m?t':diff (ThreadID, union(values (waiting ),  poss)) →
        SpecUnlocked(m, waiting, union(poss,{t'}) )
  □ SCL::acquire.m?t:poss → SpecLocked(m, t, waiting, diff(poss,{t}))


SpecLocked(m, t, waiting,  poss)  =
  □ c':  ConditionID •
       (
            (mapLookup(waiting, c') =  {}) & callSignalSpec .m.c'.t.t →
               SpecLocked(m, t, waiting,  poss)
         □ (mapLookup(waiting, c') ≠  {}) &
              callSignalSpec  .m.c'.t?t':mapLookup(waiting, c') →
                SpecLocked(m, t,
                             mapUpdate(waiting, c',  diff (mapLookup(waiting, c'),  {t'}) ),
                             union(poss,  {t'}) )
       )
  □ SCL::callSignalAll  .m?c:ConditionID!t →
        (if  mapLookup(waiting, c) =  {} then
              SpecLocked(m, t, waiting,  poss)
         else  SpecLocked(m, t, mapUpdate(waiting, c, {}),
                               union(poss,  mapLookup(waiting, c))))
  □ SCL::callRelease .m.t →
         SpecLockedReleasing(m, t,  waiting,  poss)
  □ SCL::callAcquire .m?t':diff (ThreadID,
                                  Union({values(waiting ),  poss,  {t}})) →
         SpecLocked(m, t, waiting,  union(poss,  {t'}) )
  □ SCL::callAwait .m?c:ConditionID!t →
         SpecLockedWaiting(m, c, t,  waiting,  poss)

```

Here we have defined the processes where either the monitor lock is not held, or where it is held by thread `t` and is waiting to perform a function. We next define the processes where `t` is in the middle of waiting or releasing the lock.

```
-- t doing a wait; needs to release  the lock
```

```
SpecLockedWaiting(m, c, t, waiting, poss) =
      SCL::release .m.t →
        SpecUnlocked(m, mapUpdate(waiting, c, union(mapLookup(waiting, c), {t})),
                    poss)
  □ SCL::callAcquire .m?t':diff (ThreadID,
                                 Union({values(waiting), poss, {t}})) →
        SpecLockedWaiting(m, c, t, waiting, union(poss, {t'}))

-- t releasing the lock
SpecLockedReleasing(m, t, waiting, poss) =
      SCL::release .m.t → SpecUnlocked(m, waiting, poss)
  □ SCL::callAcquire .m?t':diff (ThreadID,
                                 Union({values(waiting), poss, {t}})) →
        SpecLockedReleasing(m, t, waiting, union(poss, {t'}))

SpecSCL = (let m = SigM.S.0 within
            (SpecUnlocked(m, initSet, {})
              [[callSignalSpec .m.c.t.t' \ SCL::callSignal .m.c.t
                  | c ← ConditionID, t ← ThreadID, t' ← ThreadID]]))

```

We first note that the specification process provided is divergence free; we choose this as an idealised monitor should never internally diverge.

To test against this specification, we interleave a number of process of ThreadSCL(t), with each of these representing the potential (correct) usage of the monitor that thread t could perform. These are interleaved to form ThreadsSCL and then this is then synchronised with the SCL monitor, via the use of runWith or runWithDet as outlined above.

```
ThreadSCL(t) = SCL::Lock(SigM.S.0, t); ThreadSCL1(t)
ThreadSCL1(t) =
  □ c : ConditionID •
     (
          (SCL::Await(SigM.S.0, c, t); ThreadSCL1(t))
       □ (SCL::Signal(SigM.S.0, c, t); ThreadSCL1(t))
       □ (SCL::Signal(SigM.S.0, c, t); ThreadSCL1(t))
       □ (SCL::SignalAll (SigM.S.0, c, t); ThreadSCL1(t))
     )
  □ (SCL::Unlock(SigM.S.0, t); ThreadSCL(t))

ThreadsSCL = ||| t←ThreadID • ThreadSCL(t)
```

```

SCLSystem = SCL::runWith(ThreadsSCL, SigM.S.0, ConditionID)
SCLSystemDet = SCL::runWithDet(ThreadsSCL, SigM.S.0, ConditionID)

assert not SCLSystem :[divergence free]
assert SCLSystemDet :[divergence free]
```

We have that both the assertions pass: SCLSystem is not divergence free, but SCLSystemDet is. Since the only difference between SCLSystem and SCLSystemDet is that we block spurious wakeups in the latter, we can therefore conclude that divergence is only possible as a result of repeated spurious wakeups of waiting threads. Similarly to $\boxed{\text{ref}}$, we have that this potential divergence is not a major concern since it relies on infrequent spurious wakeups occuring. We also note that, similarly to before in $\boxed{\text{ref to previous chapter}}$, we have that each of these states where a divergence can occur has a corresponding stable state, hence it is valid for us to check refinement under stable-failures in this case.

```
  assert SpecSCL ⊑_F (SCLSystem)
  assert SpecSCL ⊑_FD (SCLSystemDet)
```

We have that both the assertions hold, indicating that the SCL monitor fulfils the specification of a monitor as required.

We next consider the fairness of the monitor with regards to individual signal calls. In the SCL monitor, queues are used so that each signal wakes the thread that has been waiting for the longest time on the condition (if one exists). We test that this property holds using AwaitOrder, a process which maintains a list of the threads waiting on each condition in the order that they started waiting.

```
valuesSeq(map) = Union({set(mapLookup(map, cnd)) | cnd <- ConditionID})
channel error : MonitorID
OrderCheck(m, waiting) =
      SCL::acquire.m?t:ThreadID ->
      (if member(t, valuesSeq(waiting)) then error.m -> STOP−−DIV
          else  OrderCheck(m, waiting))
  □ SCL::callAwait.m?c?t →
      (if  member(t, valuesSeq(waiting)) then error .m → STOP
          else  OrderCheck(m, mapUpdate(waiting, c, mapLookup(waiting, c)^<t>)))
  □ SCL::callSignalAll .m?c?_ →
          OrderCheck(m, mapUpdate(waiting, c, <>))
  □ SCL::callSignal .m?c?_ →
      (if  null (mapLookup(waiting, c)) then OrderCheck(m, waiting)
          else  OrderCheck(m, mapUpdate(waiting, c,
                                              tail (mapLookup(waiting, c)))))
```

We introduce a new channel error here. Any communication on this channel indicates that the ordering of the threads has not been maintained correctly, hence we can use the specification process and refinement checks to establish this. This new process only synchronises on the events that indicate a thread waking, waiting or acquiring the lock; this is sufficient to detect any threads which have non-spuriously woken up before they should.

To run the refinement checks, we place OrderCheck in parallel with SCLSystem and synchronise on all events that OrderCheck offers except for error.m. We then check that this still refines SpecSCL under stable failures, which it does. We can therefore conclude that no error events occur and no new stable failures are introduced, hence the ordering within the model of the SCL monitor are maintained correctly.

```
assert  SpecSCL  ⊑F  (OrderCheck(SigM.S.0, initSeq)
                     [|{|SCL::callAwait.SigM.S.0,
                         SCL::acquire .SigM.S.0,
                         SCL::callSignal .SigM.S.0,
                         SCL::callSignalAll .SigM.S.0|}|] SCLSystem)
```

5.5 Limitations of natural model of the queue

Though the model given above is a natural model of the SCL monitor, this is quite ill suited to refinement checking in FDR. The current implementation of the queue allows any thread to obtain and use any of the Nodes as its own; this leads to exponential blow up in the number of states as the number of threads increases. Considering a case where we have n threads and m are currently waiting with their nodes queued, this has $\binom{n}{m}$, or $O(n^m)$ permutations.

We can instead use the same nodes, but restrict them so that each node N.x can only be used by the respective thread T.x, removing this source of blow up. This is most trivially done by changing Await1 to specify the node to initialise and not a random one allocated by NodeAllocator i.e. as follows:

```
Await1(mon, cnd, t) = initialiseNode .N.t → ...

NodeAllocator(ns) =
    (not(empty(ns))) & (initialiseNode ?n:ns?t → NodeAllocator(diff(ns, {n})))
  □ ...
```

We will refer to this version of the queue as the ‘Simple’ model.

For further performance improvements, we can also remove the node allocator process as each node is pre-allocated. Additionally we can change the type signature of Node to N.ThreadID and simplify many of the channels (removing nodeThread and releaseNode entirely) as node indicates which thread it corresponds to as follows:

```
datatype Node = N.ThreadID
channel enqueue: MonitorID.ConditionID.Node
channel dequeue: MonitorID.ConditionID.ThreadID.Node
channel setReady: Node.ThreadID
channel isReady: Node.ThreadID.Bool
channel initialiseNode : Node
channel isEmpty: MonitorID.ConditionID.ThreadID
channel await, signalAll : MonitorID.ConditionID.ThreadID
```

All the definitions remain the same apart from removing any nodeThread and releaseNode communications and the required type changes $\boxed{\text{put raw code in an appendix?}}$. We keep initialiseNode to so that a thread can use it to indicate it is initialising a ‘new’

Table 1: The number of states generated by FDR for the different queue implementations. The improvement value is given as the $\frac{\text{Original number of states}}{\text{Reduced number of states}}$

No.	No.	Number of states				
threads	conditions	Natural	Simple	Improvement	optimised	Improvement
2	1	2288	1088	2.10	904	2.53
3	1	239428	36262	6.60	26494	9.04
4	1	3.14×10^7	1180416	26.7	792240	39.7
5	1	5.39×10^9	4.06×10^7	133	2.59×10^7	208
2	2	4932	2382	2.07	2382	2.40
3	2	686896	106672	6.44	82973	8.27
4	2	1.22×10^8	4655652	26.2	3363492	36.3
2	3	8436	4106	2.05	3634	2.32
3	3	1445008	227512	6.35	184276	7.84
4	3	3.15×10^8	1.22×10^7	25.9	9212868	34.2

ThreadInfo object and hence to reset the ready value to false. We also change InitialiseMon and InitialiseMonDet to remove the NodeAllocator; each of the individual ThreadInfo processes are still interleaved as before. We will refer to this as the 'optimised' version.

We first need to check that this simplified model remains correct. To complete this, we repeat the same refinement checks as before. These still all pass, indicating that the monitor model with a modified queue fulfills the specification similarly $\boxed{\text{wording}}$ to the natural queue model.

We next verify that the efficiency improvements occurs in practice too. We do this by running the FDR verification of assert SpecSCL [F= SCLSystem for a range of numbers of threads and conditions. We then compare the number of states generated by the natural queue model against the more efficient queues, with the results visible in table 1.

Here we see that the restricted model with each thread allocated a single node to use results in a state space reduced by a factor of at least $n!$ where n is the number of threads.

$\boxed{\text{Check exact explanation, also check wrt normalisation and symmetry}}$ This is as expected: the simplified queue has one possible bijective mapping of threads to nodes. By contrast, the natural queue has $n!$ bijective mappings of threads to nodes. As a result, for every single state that the model with the simplified queue can be in, there are upto $n!$ states of the natural model that are identical in all manners other than the node allocations.

Though the state space clearly still grows exponentially with the simplified queues, it is significantly more efficient and makes refinement checks for larger numbers of threads and conditions significantly more feasible.

$\boxed{\text{Introduce efficient spec version of SCL monitor and compare performance?}}$.

6 Barrier synchronisation

A barrier synchronisation object is used to synchronise some number of threads. $\boxed{\text{THIS}}$ This allows for a program with threads working on some shared memory which all threads can update to use a number of rounds of synchronisation in order to ensure thread-safety. Programs which use global synchronisation typically operate by instantiating some barrier object with each thread calling `sync` on the barrier once they have completed their current round. Each call to `sync` only returns after all threads have called `sync`, synchronising all the threads at that point in time and allowing the threads to proceed[17].

Here we will model and analyse an implementation of a barrier synchronisation for `n` threads internally using a binary heap of `n` two-thread signalling objects.

6.1 The Signal object

We first consider the signalling object Signal. This is used to synchronise between a 'parent' and 'child' thread, providing three external methods:

- signalUpAndWait is used by the child to signal to the parent that the child is ready to synchronise and waits until the parent signals back;
- waitForSignalUp is used by the parent to wait for the child to be ready;
- signalDown is used to indicate to the child that the synchronisation has completed.

Internally, the Signal object makes use of a private Boolean variable state with true indicating that a child is waiting and false otherwise. The use of this variable is protected by a monitor. The Scala code for the Signal object can be found in figure 8.

```
/** An object for signalling between a child and its parent in the heap. */
private class Signal{
  /** The state of this object.  true represents that the child has signalled,
    * but not yet received a signal back. */
  private var state = false

  /** Signal to the parent, and wait for a signal back. */
  def signalUpAndWait = synchronized{
    require(!state,
      "Illegal  state of Barrier:  this  might mean that it is\n"+
      "being used by two threads with the same identity.");
    state = true; notify()
    while(state) wait()
  }

  /** Wait for a signal from the child. */
  def waitForSignalUp = synchronized{ while(!state) wait() }

  /** Signal to the child. */
  def signalDown = synchronized{ state = false; notify() }
}

```

Figure 8: The Scala code for the Signal object

The signalUpAndWait function first asserts that there currently is no other child waiting for the parent to complete a signalDown; this ensures that we do not have more than one child using the same Signal object. It then sets state to true, indicating that the child is now waiting and it notifies any potentially waiting parents that the child is now syncing. It then forces the child thread to wait until the parent sets state to false and notifies the child; the use of the while loop here is to guard against spurious wakeups.

waitForSignalUp is used by a parent to wait for the child node to perform a signalUpAndWait and notify the parent; the while loop is again used to guard against spurious wakeups.

The signalDown function is used to signal to the child that the synchronisation has been completed and that the child can return from its signalUpAndWait call.

6.1.1 Modelling the Signal objects

Now we have a model for monitors we can progress onto modelling the Signal object. In each case the calling Proc will first run the procedure Func which communicates the function and the SignalID that has been called by a given ThreadID. This will then run Func2 within a Synchronized block as outlined above.

```
SignalUpAndWait :: (SignalID, ThreadID) → Proc
SignalUpAndWait(s, t) =
  callSignalUpAndWait.s.t → Synchronized(SigM.s, t, syncSignalUpAndWait(s, t))
syncSignalUpAndWait(s, t) =
  getState.s.t?val → if val == true then DIV -- Required to be false
                     else setState.s.t.true →
                          Notify(SigM.s, t); SignalWaitingForFalse(s, t)


SignalWaitingForFalse :: (SignalID, ThreadID) → Proc
SignalWaitingForFalse(s, t) =
  getState.s.t?val → if val == false then SKIP
                     else Wait(SigM.s, t); SignalWaitingForFalse(s, t)
```

This models entering a synchronized block and checks that state is not true, diverging if so. This divergence is used to model a failed assertion; the rest of the code is a more direct translation.

```
WaitForSignalUp :: (SignalID, ThreadID) → Proc
WaitForSignalUp(s, t) =
  callWaitForSignalUp.s.t → Synchronized(SigM.s, t, syncWaitForSignalUp(s, t))
syncWaitForSignalUp(s, t) =
  getState.s.t?val → if val == true then SKIP
                     else Wait(SigM.s, t); syncWaitForSignalUp(s, t)
```

Again this is a fairly natural model of the Scala code presented earlier; we communicate that thread t has called waitForSignalUp on Signal object s, enter a synchronized block and then simulate the while loop used to guard against spurious wakeups.

```
SignalDown :: (SignalID, ThreadID) → Proc
SignalDown(s, t) = callSignalDown.s.t → Synchronized(SigM.s, t, syncSignalDown(s, t))
syncSignalDown(s, t) = setState.s.t.false → Notify(SigM.s, t); SKIP
```

SignalDown is the most simple function of the three to model; it obtains the monitor's lock, sets the state variable to false and then notifies the child that the synchronisation has completed.

```
signalChannels = {|callSignalUpAndWait, callWaitForSignalUp, callSignalDown|}
```

```

InitialiseSignals    (threads) =
  (threads [|union(Monitors:: events, stateChannels)|]
    (State ||| InitialiseAlls    )) \ union(stateChannels, Monitors:: events)
```

We finally define a process InitialiseSignals. This is used to synchronise the threads with the interleaving of the state variables and the monitors and then hiding the internal behaviour of the Signal objects.

6.2 The Barrier object

When initialised, the Barrier(n: Int) object creates an array of n Signal objects, with these organised in the structure of a heap. As per the trait of a barrier synchronisation, Barrier only provides a single function sync(me) which takes the thread's identity as an input:

```scala
/** Perform a barrier synchronisation.
  * @param me the unique identity of this thread. */
def sync(me: Int) = {
  require(0 <= me && me < n,
    s"Illegal  parameter $me for sync: should be in the range [0..$n).")
  val child1 = 2*me+1; val child2 = 2*me+2
  // Wait for children
  if (child1 < n) signals(child1).waitForSignalUp
  if (child2 < n) signals(child2).waitForSignalUp
  // Signal to parent and wait for signal back
  if (me != 0) signals(me).signalUpAndWait
  // Signal to children
  if (child1 < n) signals(child1).signalDown
  if (child2 < n) signals(child2).signalDown
}
```

This checks that the thread's identity is such that signals(me) does not cause an ArrayIndexOutOfBoundsException. It then waits for the thread's children (if they exist) to signal that they are ready to synchronise, before signalling to its parent that all of its descendants are ready to synchronise. Once the parent signals back that the synchronisation has occurred the thread notfies its children that the synchronisation has completed befoe returning. The exception to this is thread 0, which has no parent to signal to and all of its descendants are ready to synchronise, hence thread 0 reaching line 11 of its execution can be taken as the linearization point of when the barrier synchronisation occurs.

6.2.1 Modelling the Barrier object

```
1  Thread(T.me) = beginSync.T.me → Sync(T.me) ⊓ end.T.me → SKIP
2  Sync(T.me) =
3    let  child1  = 2∗me+1
4         child2  = 2∗me+2
5    within
6         (if  (child1  <  n) then WaitForSignalUp(S.(child1),  T.me) else SKIP);
7         (if  (child2  <  n) then WaitForSignalUp(S.(child2),  T.me) else SKIP);
8         (if  me ≠ 0 then SignalUpAndWait(S.me, T.me) else SKIP);
9         (if  (child1  <  n) then SignalDown(S.(child1),  T.me) else SKIP);
10        (if  (child2  <  n) then SignalDown(S.(child2),  T.me) else SKIP);
11        endSync.T.me → Thread(T.me)
12
13
14 Threads = ||| t :  ThreadID • Thread(t)
15
16 BarrierSystem  = InitialiseSignals   (Threads)
```

$\boxed{\text{Prior explanation for T.me and S.me}}$ The process Thread(T.me) models the individual behaviour of a specific thread with identity T.me :: ThreadID, with each thread nondeterministically choosing to either terminate after communicating an end.T.me or to call the CSP model of the sync(me). In the latter case, a communication of beginSync.T.me is used to indicate the start of the synchronisation. The Sync(T.me) definition is very straightforward, with most of it directly following from the Scala definition; the only further change is that Sync(T.me) communicates a endSync.T.me event just before it terminates.

The thread processes are then interleaved together to yield Threads. We then initialise the system with Signal objects that can nondeterministically allow or block spurious wakeups to give BarrierSystem. We hide all events of the signal object, so the only visible channels of BarrierSystem are {beginSync, endSync, spuriousWakeup, end}.

6.3 Correctness of the model

We first will show that the barrier synchronisation is correct. This means that it can be correctly linearised and if a synchronisation is possible then it will always occur. Correctly linearised means that the barrier synchronisation can be considered to occur at some point between when all n threads have communicated beginSync and when the first thread communicates an endSync event. The requirement that a linearisation must occur means that if all n threads communicate a beginSync then none of the threads can be blocked from communicating their respective endSync.

```
Lineariser (t) = beginSync.t → sync → endSync.t → Lineariser(t)
                 ⊓ end.t → STOP
Spec = ( ‖ t ← ThreadID • [{beginSync.t, sync, endSync.t, end.t}]
                Lineariser (t)) \ {sync}
```

Lineariser(t) allows any thread to beginSync.t followed by an endSync.t, representing the call and return of barrier.sync(). The sync event can be considered to be the point at which the barrier synchronisation occurs since all threads must synchronise on this, fulfilling the requirement above. Additionally, each thread can terminate via end.t, indicating that it will perform no further synchronisations; this also restricts all other threads from completing a barrier synchronisation as t is no longer able to communicate a sync.

We first note that BarrierSystem is divergence-free, but BarrierSystem \ {|spuriousWakeup|} is not. BarrierSystem being divergence-free is relevant as this means that we never breach the assertion in the SignalUpAndWait function (recall that the model would diverge if the value of state was true) and that the system is divergence-free with visible spurious wakeups. This therefore means hiding the spuriousWakeup events must be the cause of the divergences. This is expected behaviour as one thread could spuriously wakeup, check the test condition and wait again before spuriously waking up like this indefinitely. This is not a particular cause for concern as although the JVM could allow an unbounded number of spurious wakeups, in practice spurious wakups occur infrequently.

We now consider the traces model, where we have that the following holds:

```
assert Spec $\sqsubseteq_T$ BarrierSystem \ {|spuriousWakeup|}
```

This means that BarrierSystem fulfils the requirements fulfilled by Spec i.e. that it can be linearised and that the synchronisation between all n threads occurs correctly (if indeed it does occur).

Since Spec also cannot diverge we will also consider refinement under the stable failures model for both systems. This ensures that if a synchronisation can occur then it must occur and all threads communicating callSync can return if all n threads communicate a callSync. We note that using the stable failures model is normally inappropriate for a system that can diverge. However, this is valid here as for any state that could be unstable due to a hidden spuriousWakeup there exists a corresponding state where the regulator process Reg blocks the spurious wakeup and is therefore stable. FDR yields that both the following hold for systems of upto 6 threads in $\boxed{\text{Time value}}$:

```
assert Spec $\sqsubseteq_F$ BarrierSystem \ {|spuriousWakeup|}
```

As a result, we have that the Barrier object presented earlier is linerais a correct barrier synchronisation object for all n upto and including 6.

6.4 Specification processes for the Signal objects

Our current model of Signal models the internal workings of the object, modelling the synchronized blocks and the internal state variable. Though this is a faithful recreation, this is a rather complex model and leads to the number of states that FDR must generate being exponential in the number of threads, resulting in us only being able to test refinement on a system of size 6 in reasonable time. We can instead construct a specification process which models the use of the Signal object. Though this will still result in an exponential model, the model will be of significantly smaller size allowing us to model the Barrier object for larger numbers of threads in the same approximate time.

By inspecting the usage of Signal we observe that there are two synchronisations between threads performed by each Signal object $\boxed{\text{Diagram}}$

- waitForSignalUp and signalUpAndWait synchronise to indicate that that all threads using objects in this subtree are waiting to synchronise. This synchronisation has the parent waiting on the child to signal, with the child being allowed to signal and progress immediately

- signalDown and signalUpAndWait synchronise, with the parent signalling to the child that the barrier synchronisation has occurred and that signalUpAndWait can return. This synchronisation has the child thread waiting on the parent signalling down to it; the child thread is always waiting first as the child starts waiting on this synchronisation immediately after the previous synchronisation occurs.

We can model this simplified Signal object via the following CSP:

```
channel endWaitForSignalUp, endSignalUpAndWait : SignalID . ThreadID

-- Simplified spec for a correctly  used Signal object
SpecSig(s) =
    callSignalUpAndWait.s?t → callWaitForSignalUp.s?t2 → SpecSig2(s, t, t2)
  □ callWaitForSignalUp.s?t2 → callSignalUpAndWait.s?t → SpecSig2(s, t, t2)
SpecSig2(s, t, t2) =
  endWaitForSignalUp.s.t2 → callSignalDown.s.t2 → endSignalUpAndWait.s.t →
  SpecSig(s)

-- The individual functions for the Signal object
SpecSignalUpAndWait(s, t) = callSignalUpAndWait.s.t → endSignalUpAndWait.s.t →
  SKIP
SpecWaitForSignalUp(s, t) = callWaitForSignalUp.s.t → endWaitForSignalUp.s.t → SKIP
SpecSignalDown(s, t) = callSignalDown.s.t → SKIP

-- Construct the system for each of the SpecSig objects
SpecSignals =
  || s ← SignalID • [{|callSignalUpAndWait.s, callWaitForSignalUp.s,
                        callSignalDown.s, endWaitForSignalUp.s,
  endSignalUpAndWait.s|}]
                     SpecSig(s)

-- Method for barrier-sync to initialise the two objects
InitialiseSpecSignals (threads) =
  (SpecSignals [|union(signalChannels, waitChannels)|] threads) \ waitChannels
```

We introduce channels endWaitForSignalUp and endSignalUpAndWait to represent the synchronisations between the child and parent, with each of the channels indicating that their respective functions are able to return. SpecSig(s) is used to dictate the order that communications are allowed to occur:

1. Initially, it can either communicate a callSignalUpAndWait from the child thread or a callWaitForSignalUp from the parent. It then communicates the other event.
2. It then communicates an endWaitForSignalUp to indicate to the parent that the first synchronisation has occurred.

3. The parent then commmunicates a callSignalDown indicating that the barrier synchronisation has occured.

4. Finally, a endSignalUpAndWait is communicated to indicate to the child that they can now terminate; SpecSig(s) then repeats.

We also define the specification versions of the three external methods offered by a Signal object. SpecSignalUpAndWait and SpecWaitForSignalUp both initially communicate an event indicating that they have been 'called' before communicating a endSignalUpAndWait or endWaitForSignalUp respectively before terminating. By contrast, SpecSignalDown immediately terminates after communicating that it has been 'called' as it does not require a synchronisation with another thread.

Finally for the Signal specifications, we let SpecSignals be the alphabetised parallel composition of each of the SpecSig(s) processes, with the parallel composition forcing each specification object to only synchronise on events with the matching SignalID. The individual threads and the overall system are defined similaly to the above, with the exception that all calls are to the specification processes and not the originals.

$\boxed{\text{Stuff about how the initial implementation of Signal fulfils this specification}}$

```
-- sThread is the same as Thread but uses the spec Signal
sThread(T.me) = beginSync.T.me → sSync(T.me) ⊓ end.T.me → SKIP
sSync(T.me) =
  let child1  = 2*me+1
      child2  = 2*me+2
  within (if  (child1 <  n) then SpecWaitForSignalUp(S.(child1), T.me) else SKIP);
         (if  (child2 <  n) then SpecWaitForSignalUp(S.(child2), T.me) else SKIP);
         (if  me ≠ 0 then SpecSignalUpAndWait(S.me, T.me) else SKIP);
         (if  (child1 <  n) then SpecSignalDown(S.(child1), T.me) else SKIP);
         (if  (child2 <  n) then SpecSignalDown(S.(child2), T.me)else SKIP);
         endSync.T.me → sThread(T.me)

-- Initialise  the simple system
sThreads = ||| t : ThreadID • sThread(t)
sBarrierSystem  = InitialiseSpecSignals  (sThreads)

-- Spec failure—divergences refines it  as expected
assert  Spec  ⊑_FD  sBarrierSystem
```

Since the simplified Signal object does not use monitors we have that the system should be divergence-free. This is verifed by FDR as the above refinement holds against our (divergence-free) linearization checker.

Proper performance comparison Simplified can run 10 threads in about the same time

the normal version can run 6

In this paper, we have examined a range of concurrency primitives offered by the Java Virtual Machine, Scala Concurrency Library module and a range of different lock designs. We have examined and proved the correctness of each of these whilst also proving complexity results and examining other properties. There are, however, some limitations to our work.

Firstly, by the nature of model checking, we are only able to model a limited number of threads with restrictions on other parameters too. Though model checking with larger numbers of threads is technially possible, the exponential blow up in the number of states renders it infeasible. If a model is correct for small numbers of threads, we have significantly more confidence in the model remaining correct for larger numbers of threads. Taking the SCL monitor which has been proven correct for 6 threads and 2 conditions, since our model for the SCL monitor has been proven to be correct for 6 threads, any bug would require some interaction that requires at least 7 threads or 3 conditions.

We have also only considered a number of specific concurrency primitives. Though our approach can be extended to many other primitives, this still requires work during development to verify the correctness of these. An automated translation system from normal code to CSP would aid in this, however any general translator would suffer from additional complexity blow up due to a lack of insight. An example of this can be seen in the SCL Monitor model, where a more natural implementation of the queue lead to state spacea $n!$ times larger than a model with additional insight. Implementing an automated translator, either naïve or optimised, is beyond the scope of the project and therefore left as further work.

References

[1] Chong et al. “Code-Level Model Checking in the Software Development Workflow”. In: 2020.

[2] Sara Baase. *A Gift of Fire, 4th Edition*. 2012. URL: https://www.philadelphia.edu.jo/academics/lalqoran/uploads/A-Gift-of-Fire-4thEd-2012.pdf.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. MIT Press, 2008.

[4] John Bluck and Valerie Williamson. *New NASA Software Detects ’Bugs’ in Java Computer Code*. Tech. rep. NASA, 2005. URL: https://web.archive.org/web/20100317110540/https://www.nasa.gov/centers/ames/news/releases/2005/05_28AR.html.

[5] Independent Inquiry Board. *ARIANE 5: Flight 501 Failure*. Report. 1996. URL: http://sunnyday.mit.edu/nasa-class/Ariane5-report.html.

[6] Bettina Buth et al. “Deadlock analysis for a fault-tolerant system”. In: *Algebraic Methodology and Software Technology*. Ed. by Michael Johnson. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 60–74. ISBN: 978-3-540-69661-2.

[7] Henrico Dolfing. *Case Study 4: The $440 Million Software Error at Knight Capital*. 2019. URL: https://www.henricodolfing.com/2019/06/project-failure-case-study-knight-capital.html.

[8] Bill Roscoe Gavin Lowe. *Concurrency Lecture Slides*. 2020. URL: https://www.cs.ox.ac.uk/teaching/materials19-20/concurrency/.

[9] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.

[10] Andrew Jones. *An Introduction to NASA's Java Pathfinder*. 2010. URL: https://www.doc.ic.ac.uk/~wlj05/ajp/lecture6/slides/slides.pdf.

[11] Cliff Jones. "Tentative steps toward a development method for interfering programs". In: *ACM Transactions on Programming Languages and Systems* 5.4 (1983).

[12] Gunnar Kudrjavets Laurie Williams and Nachiappan Nagappan. *On the Effectiveness of Unit Test Automation at Microsoft*. URL: https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf.

[13] Jonathan Lawrence. "Practical Application of CSP and FDR to Software Design". In: *Communicating Sequential Processes. The First 25 Years: Symposium on the Occasion of 25 Years of CSP, London, UK, July 7-8, 2004. Revised Invited Papers*. Ed. by Ali E. Abdallah, Cliff B. Jones, and Jeff W. Sanders. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 151–174. ISBN: 978-3-540-32265-8. DOI: 10.1007/11423348_9. URL: https://doi.org/10.1007/11423348_9.

[14] K. Rustan M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370. ISBN: 978-3-642-17511-4.

[15] Gavin Lowe. "An attack on the Needham-Schroeder public-key authentication protocol". In: *Information Processing Letters* 56.3 (1995), pp. 131–133. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(95)00144-2. URL: https://www.sciencedirect.com/science/article/pii/0020019095001442.

[16] Gavin Lowe. *Analysing Lock-Free Linearizable Datatypes using CSP*. 2017. URL: https://www.cs.ox.ac.uk/people/gavin.lowe/LockFreeQueue/lockFreeListAnalysis.pdf.

[17] Gavin Lowe. *Concurrent Programming Lecture Slides*. 2022. URL: https://www.cs.ox.ac.uk/teaching/materials22-23/concurrentprogramming/.

[18] Gavin Lowe. “Discovering and correcting a deadlock in a channel implementation”. In: *Formal Aspects of Computing* 31 (4 2019). URL: https://doi.org/10.1007/s00165-019-00487-y.

[19] Gavin Lowe. “Implementing Generalised Alt”. In: *Communicating Process Architectures 2011* (2011).

[20] Gavin Lowe. “Testing for linearizability”. In: *Concurrency and Computation: Practice and Experience* 29.4 (2017), e3928. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.3928.

[21] Jamie Lynch. *The Worst Computer Bugs in History: Race conditions in Therac-25*. Blog. 2017. URL: https://www.bugsnag.com/blog/bug-day-race-condition-therac-25/.

[22] Hanno Nickau and Gavin Lowe. *Concurrent Algorithms and Data Structures Lecture Notes*. 2023. URL: https://www.cs.ox.ac.uk/teaching/courses/2023-2024/cads/.

[23] Andrew (Bill) Roscoe. *Understanding Concurrent Systems*. Springer, 2010.

[24] Gerhard Schellhorn and Simon Bäumler. “Formal Verification of Lock-Free Algorithms”. In: *Ninth International Conference on Application of Concurrency to System Design* (2009).

[25] Daniel Schwartz-Narbonne. *How to integrate formal proofs into software development*. 2020. URL: https://www.amazon.science/blog/how-to-integrate-formal-proofs-into-software-development.