

4yp

Thomas Aston

May 19, 2024

Contents

1	Introduction	2
1.1	Contributions	4
2	Related Work	5
3	Modelling and analysing implementations of locks	8
3.1	External interfaces	8
3.1.1	Mutual Exclusion	9
3.1.2	Deadlock Freedom	9
3.1.3	Livelock Freedom	10
3.1.4	Starvation Freedom	11
3.2	First-come-first-served	12
4	Monitors	13
4.1	The JVM monitor	14
4.1.1	Modelling the JVM monitor	14
4.2	The SCL Monitor	16

4.2.1	The SCL monitor implementation	18
4.2.2	The SCL monitor model	20
4.2.3	The functions and interface of the monitor	26
4.2.4	Correctness	29
4.2.5	Limitations of natural model of the queue	33
5	Barrier synchronisation	36
5.1	The signalling object	36
5.1.1	Modelling the Signal objects	38
5.2	The Barrier object	40
5.2.1	Modelling the Barrier object	41
5.3	Correctness of the model	42
5.4	Specification processes for the Signal objects	44
6	Conclusions and future work	47
	• Change figure linkings	
	• Lineariser spelling??	

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 [8] 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 [24]. 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 thorough system verification 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 [13], 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 relies on the random testing of edge cases; clearly this is not exhaustive either [23].

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 via a variety of synchronisation objects.

There are two main benefits of this. Firstly, we prove that the synchronisation objects function correctly; debugging concurrent programs is a complex task and bugs due to issues with synchronisation objects can often be particularly challenging to resolve. The second benefit is that we can significantly optimise the model checking of systems that use these primitives. We prove that the synchronisation objects fulfil a specification with the same properties but without all of the internal workings. We can therefore optimise model checking of whole systems by replacing the original synchronisation objects with their respective specification process without changing system behaviour. This allows us to significantly reduce the possible state space of the system, making formal verification of complex systems significantly more feasible.

We choose to use the process algebra Communicating Sequential Processes (CSP) [27] as our tool for modelling these interactions and the accompanying FDR as a model checker [FDR]. CSP is a language for describing processes that can interact both with their environment and other parallel processes and its accompanying checker FDR provides powerful model checking. Since CSP is covered in the Part B Concurrency course [9], we assume familiarity with it (as described in [27]) throughout.

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 Scala Concurrency Libraray

(SCL) [18] 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 [25] and a number of desirable properties of locks, using CSP refinement checks to determine which properties each lock satisfies. We also discuss the feasibility of modelling infinite liveness properties using CSP, specifically starvation-freedom.

2 Related Work

Current literature highlights three 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. AWS provide an example of this approach, using formal verification methods in code development in order to ensure that their code always meets its specifications [1, 29]. This process involve writing a specification for each function in the form of a number of pre- and post-conditions, with these properties validated automatically. Though similar in style to standard unit-tests, these actual proofs that are continually checked during development, with routine checking of these proofs indicating the correctness (or otherwise) of the code. Though 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. To aid with this, some languages such as Dafny [15] have been developed with built in model checkers as an

extension of this approach; these are however yet to reach mainstream adoption.

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 [11]. 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 [7]. These approaches tend to be quite inefficient; automated translation has no inherent knowledge about restrictions on the usage of parameters or datatypes, resulting in a potentially very inefficient model.

The final approach to formal verification is to write a model of the code by hand and then verify that model. This approach is more involved and limited in scope than the two alternative methods, but can lead to a larger range of results and additional proofs that the code fulfils certain properties beyond just correctness. It also has the benefit that it is less affected by the state space explosion problem than automated tools.

The state space explosion problem is that adding a single additional process often leads to an exponential increase in the number of states generated by FDR [27]. Indeed, a single process with a parameter and k states $t : S$ can have k^t states; a network of N of these processes can have N^{k^t} total states. This quickly becomes infeasible to check for even relatively small values of N , k and t . Additionally, a queue can have a potentially infinite number of possible permutations, leading to FDR being unable to generate the complete state space to model check. Insights and smart design can be used to create a smaller, but still correct, model than automated translation is capable of. This allows for model checking of system with more threads or a wider range of parameter values,

leading to greater confidence in their correctness. We therefore focus on this approach.

We choose to use CSP for this task. CSP is very suitable for modelling concurrent systems with tightly restricted communications between threads [14] 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 led to significant verification results previously, most prominently being Lowe’s detection of a man-in-the-middle attack on the Needham-Schroeder protocol [16]. 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 [19]. 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 [25]: lock-based, message passing and datatype-based concurrent programming. There exists literature on formal verification of the latter two; Lowe has previously proved the correctness of a lock-free queue[17] and also an implementation of a generalised alt operator [20]. There is also more general work on the verification of lock-free algorithms, such as Schellhorn and Bäumler [28]. Their work uses an extended form of linear temporal logic (LTL) and the rely-guarantee paradigm (introduced by [12]) 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.

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. We will present a few models of locks and examine how we can model certain liveness and safety properties using CSP.

3.1 External interfaces

The most straightforward interface 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.

```
1  trait Lock{
2    /** Acquire the Lock. */
3    def lock : Unit
4    /** Release the Lock. */
5    def unlock : Unit
6    ...
7  }
```

Figure 1: A Scala interface for a simple lock

MOVE 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; this can be used to indicate that the thread has terminated, been permanently descheduled

Throughout the paper, we will use **callX** to represent a thread calling function **X** and **end.t** to represent a thread terminating. The set of all threads is **ThreadID** :: **T**.{0 .. **NTHREADS** - 1}. We will now specify some ideal properties of locks using these channels:

3.1.1 Mutual Exclusion

Mutual exclusion is a safety property which states that at most one thread; 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:

$$\begin{array}{l} 1 \quad \text{Mutex} = \text{lockAcquired.l.t} \rightarrow \text{lockReleased.l.t} \rightarrow \text{Mutex} \\ 2 \quad \text{assert } \text{Mutex} \sqsubseteq_T X \setminus (\Sigma - [\text{lockAcquired.l}, \text{lockReleased.l}]) \\ 3 \end{array}$$

3.1.2 Deadlock Freedom

This specifies that if some thread attempts to acquire the lock then a thread will succeed in acquiring the lock[10].

We can express this in the stable-failures model by ensuring that **lockAcquired.l** is always available to be communicated when some thread has called the lock but not yet obtained it.

$$\begin{array}{l} 1 \quad \text{AcquireLock}(l, \text{ts}) = \\ 2 \quad \quad \text{callLock.l.t:}(\text{diff}(\text{ThreadID}, \text{ts})) \rightarrow \text{AcquireLock}(l, \text{union}(\text{ts}, \{t\})) \\ 3 \quad \quad \square \text{lockAcquired.l.t:ts} \rightarrow \text{AcquireLock}(l, \text{diff}(\text{ts}, \{t\})) \\ 4 \end{array}$$

```

5   assert AcquireLock(l, ts)  $\sqsubseteq_F$ 
6       X \ (\Sigma - \{\text{callLock}.l, \text{lockAcquired}.l\})
7

```

AcquireLock takes two parameters: l is the identity of the lock and ts is the set of threads currently which have communicated a **callLock** but haven't yet acquired the lock. In our refinement check, we will assume that no lock events have occurred prior; ie. no threads have already attempted to acquire the lock.

3.1.3 Livelock Freedom

This requirement specifies that the number of internal actions on a lock must be bounded while no thread holds the lock; i.e. threads can't indefinitely repeat actions whilst the lock is unheld. This can be captured in the failures-divergences model, using a specification process parametrised over lock l and a failures-divergences refinement against a system that only has **lockAcquired** and **lockReleased** as visible communications.

```

1 LiveUnlocked(l) = lockAcquired.l ? t  $\rightarrow$  LiveLocked(l)
2                  $\sqcap$  STOP
3 LiveLocked(l) = lockReleased.l ? t  $\rightarrow$  LiveUnlocked
4                  $\sqcap$  DIV
5
6 assert LiveUnlocked(l)  $\sqsubseteq_{FD}$  X \ (\Sigma - [\text{lockAcquired}.l, \text{lockReleased}.l])

```

This specification allows the lock to diverge only when it is held by some thread and to be divergence free otherwise. This forces the number of internal actions when the lock is not held to be finite (else it could diverge and the refinement would fail), indicating that no livelock has occurred. We allow the specification to non-deterministically **STOP** when the lock is unheld; this models the effective behaviour of the lock after all threads terminate.

These are three almost essential properties of useful locks; we will consider starvation-freedom later.

```

1 LockSpec(l, ts) = SpecLock(l) [|{lockAcquired.l, lockReleased.l}|]
2                 (|| t ← ThreadID • SpecThread(l, t))
3 SpecThread(l, t) =
4     callLock .l .t → lockAcquired.l.t → lockReleased.l.t → SpecThread(l, t)
5     □ end.t → STOP
6 SpecLock(l) = lockAcquired.l?t → lockReleased.l.t → SpecLock(l)

```

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

3.2 A simple lock specification

Figure ?? shows a simple trace specification for a lock, where l is the identity of the lock and ts is the set of threads that can interact with the lock.

This specification has the properties of mutual exclusion, livelock-freedom and deadlock-freedom; we have verified this by running the three assertions from 3.1.1, 3.1.2 and 3.1.3. As a result, any process which failures refines `check` this specification also has these three properties. `explanation of why?`

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, with `true` indicating that some thread holds the lock and `false` otherwise. The `AtomicBoolean`, has atomic `get` and `set` operations to read and write values respectively. It also has a `getAndSet` operation which atomically sets the value of the Boolean and returns the old value. The Scala code can be seen in Listing ?. `state` being false is equivalent to the lock being unlocked; a communication of `getAndSet(true)` with previous value `false` indicates that that thread has now obtained the lock.

In order to model the TAS lock, we first need a process that acts as an `AtomicBoolean` to model the `state` variable. Figure ?? introduces a process `AtomicVar` than takes an

```

1  import java.util.concurrent.atomic.AtomicBoolean
2
3  /** A lock based upon the test-and-set operation
4   * Based on Herlihy & Shavit, Chapter 7. */
5  class TASLock extends Lock{
6      /** The state of the lock: true represents locked */
7      private val state = new AtomicBoolean(false)
8
9      /** Acquire the Lock */
10     def lock = while(state.getAndSet(true)){ }
11
12     /** Release the Lock */
13     def unlock = state.set(false)
14 }
15

```

Figure 3: Test-and-set lock from [25]

```

1  AtomicVar(value, get, set, gAS) =
2      get?_!value → AtomicVar(value, get, set, gAS)
3      □ set?_?value' → AtomicVar(value', get, set, gAS)
4      □ gAS?_!value?value' → AtomicVar(value', get, set, gAS)
5

```

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

initial value, and channels $\text{get}, \text{set} : \text{ThreadID}.T$ and $\text{getAndSet} : \text{ThreadID}.T.T$ for some arbitrary type T .

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

```

1  channel get, set: ThreadID . Bool
2  channel getAndSet: ThreadID . Bool . Bool
3  State = Var(false, get, set, getAndSet)
4  InternalChannels = {get, set, getAndSet}

```

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

initially.

We next model the operations of the lock itself. Both operations are trivial to convert, and we can linearize `Lock(t)` when the communication `getAndSet.t.False.True` occurs, indicating that `t` has obtained the lock.

```

1  Unlock(t) = setState.t! False → SKIP — def unlock = state.set(false)
2
3  Lock(t) = getAndSet.t?v!True → if v = False then SKIP
4                                     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 thread. Each thread can non-deterministically choose to either terminate or obtain the lock, release the lock and repeat. Here we use `L.0` as the identity of the lock.

```

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

```

Finally, we construct the lock from its components. We first synchronise all the threads over the `get`, `set` and `getAndSet` channels with the `State` process. Since `getAndSet.t.False.True` and `setState.t.False` are the linearisation points of thread `t` obtaining and 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`.

Diagram?

```

1  — All initially do not hold the lock
2  AllThreads = ||| t : ThreadID • Thread(t)
3  — Allow all threads to perform actions on the state variable
4  ActualSystem = (AllThreads [|InternalChannels|] State)
5  — Rename lock acquisition and releasing and hide internal events
6  ActualSystemR = ActualSystem
7                      [getAndSet.t.False.True \ lockAcquired.L.0.t,
8                      set.t.False \ lockReleased.L.0.t | t ← ThreadID]

```

3.3.1 Analysis

We firstly examine whether this model fulfils the aforementioned properties. The mutual exclusion, deadlock freedom and livelock-freedom tests from sections 3.1.1, 3.1.2 and 3.1.3 respectively pass. The model also does not diverge before it is first held; these are all expected results Why?. The TAS lock is also equivalent under traces with the earlier lock specification. However, the model can diverge whenever the lock is held. This occurs when a thread (or threads) attempting to obtain the lock a thread attempting to obtain the lock perform an infinite number of `getAndSet` operations; an example trace of this behaviour where `T.0` obtains the lock `l` follows

```

1  <callLock .l .T.0, callLock .l .T.1, getAndSet.T.0.False.True>^
2  <getAndSet.T.1.True.True>^ω

```

This behaviour is expected; thread `T.1` is trying to obtain the lock and is being blocked by `T.0` which holds the lock. This behaviour is, however, problematic for low-level performance. Any `getAndSet` operation causes a broadcast on the shared memory bus between the processors, delaying all processors Ref. This also forces each thread to invalidate the value of `state` from the caches, regardless of whether the value has actually been changed. Since the above trace never results in thread `T.1` successfully setting `state`, it is preferable to limit the number of `getAndSet` operations without unnecessarily delaying a thread from obtaining the lock. As a result, we use less costly `get` operations in order to limit the usage of `getAndSet` operations; these `getAndSet` operations are instead limited to situations where they are likely to obtain the 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

```

1  class TTASLock extends Lock{
2      ...
3      /** Acquire the lock */
4      def lock =
5          do{
6              while(state.get()){ } // spin until state = false
7          } while(state.getAndSet(true)) // if state = true, retry
8      ...
9  }
10

```

Figure 5: Test-and-test-and-set lock from [25]

improvement

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 similar to the TAS lock. The sole change is to the `lock` function, as can be seen in Figure ??, which we then specify in our CSP model as the following:

```

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

```

The TTAS lock can still produce traces with threads performing an unbound number of consecutive operations, however these are now `get` operations instead of `getAndSet` operations. The TTAS lock has performs at most one `getAndSet` operation per thread when the lock becomes available. This is the case when each thread's last communication was `get.t.False`, indicating that the lock is available and hence leading to a `getAndSet` communication to attempt to gain the lock.

We now have a linear bound on the number of unsuccessful `getAndSet` operations, resulting in much more efficient usage of caching and shared memory. This has been verified by synchronising with a regulator process which outputs on an error channel

if n `getAndSets` occur in one locking cycle; this regulator acts as a watchdog. Since the trace refinement is satisfied, we have that no `error` event has been communicated and hence at most n `getAndSets` can occur every time the lock is released.

```

1 channel error
2 Reg(x) =   gASState?_ → if (x < card(ThreadID)) then Reg(x+1)
3           else error → STOP
4           □ lockReleased?_ → Reg(0)
5
6 assert LockSpec(L.0, {}, ThreadID)
7       ⊆T ActualSystemR [|{gASState, lockReleased}|] Reg(0)

```

3.5 Peterson lock

The Peterson lock is

3.5.1 Starvation Freedom

Starvation freedom is a liveness property that states that, under the assumption that no thread holds the lock indefinitely, every thread that attempts to acquire the lock eventually succeeds [10]. It requires that any thread attempting to gain the lock must can only be bypassed by other threads a finite number of times.

One common approach to checking infinite properties in CSP is to hide some (in this case internal) channels and then check that the model does not diverge. This approach does not work here: consider a starvation-free lock which uses busy waiting (repeatedly testing if the lock is available). We have that hiding the internal communications results in a divergence, however the lock is starvation-free. An example of such a lock is the Peterson lock [link to model](#).

Roscoe and Gibson-Robinson showed that every infinite traces property that can be captured by CSP refinement can also be captured by a finite-traces refinement check when combined with the satisfaction of a deterministic Büchi automaton [26]. A Büchi

automaton, as explained by REF, is an automaton that takes infinite inputs and accepts an input if an accepting state is visited infinitely often Improve. need to go over the above However, we can show that no deterministic Büchi automaton can capture starvation freedom.

We will show by contradiction that no such deterministic automaton B can capture starvation freedom. Let us without loss of generality consider 2-threaded locks.

By the definition of starvation freedom, the automaton should be satisfied if T.0 acquires and never releases the lock and T.1 attempts to acquire the lock. The automaton therefore must have some accepting state which is visited an infinite number of times when T.0 holds the lock and T.1 attempts to acquire it; we will call this state q_a .

Now we consider the TAS lock from before. This is clearly not starvation free as the first thread to communicate on getAndSet once the lock becomes available will acquire it; this allows some thread to infinitely bypass some other waiting thread. Consider an execution where T.0 repeatedly acquires and releases the lock, bypassing T.1 which is repeatedly attempting and failing to obtain the lock. This execution, however, passes through the state q_a of B every time T.0 acquires and later releases the lock. This can occur an infinite number of times, resulting in B accepting this execution. This is a contradiction as this execution is clearly not starvation free as T.1 never obtains the lock, however it still satisfies B .

Improve We hence have that no deterministic Büchi automaton can accurately capture starvation freedom and hence we cannot test directly for starvation-freedom through a standard FDR refinement check.

3.6 First-come-first-served

We can instead consider capturing the stronger property of first-come-first-served. Locks that satisfy this property can be split into a *doorway* section of a bounded number of

steps and a following *waiting* section of a potentially unbounded number of execution steps. This property states that once thread X has completed the doorway section of the lock it cannot be overtaken by a thread Y that has not yet started its doorway section; i.e. X will acquire the lock before Y acquires the lock. This implies starvation-freedom as once X has completed its doorway section, it can only be bypassed by threads who have started their doorway section prior to X .

```

1  FCFSNotStarted(l, t, s1, s3) =
2      end.t → FCFSTerminated(l, t)
3      □ callLock.l.t → FCFSStarted(l, t, s1, s3)
4      □ doorwayComplete.l?t':diff(ThreadID, union({t}, union(s1, s3))) → FCFSNotStarted(l,
5          t, union(s1, {t'}), s3)
6      □ lockAcquired.l?t':diff(union(s1, s3), {t}) → FCFSNotStarted(l, t, diff(s1, {t'}), s3)
7
8  FCFSStarted(l, t, s1, s3) =
9      doorwayComplete.l.t → FCFSCompleted(l, t, s1, s3)
10     □ doorwayComplete.l?t':diff(ThreadID, union({t}, union(s1, s3))) → FCFSStarted(l, t,
11         s1, union(s3, {t'}))
12     □ lockAcquired.l?t':diff(union(s1, s3), {t}) → FCFSStarted(l, t, diff(s1, {t'}),
13         diff(s3, {t'}))
14
15  FCFSCompleted(l, t, s1, s3) =
16     empty(s1) & lockAcquired.l.t → FCFSNotStarted(l, t, s3, {})
17     □ doorwayComplete.l?t':diff(ThreadID, union({t}, union(s1, s3))) → FCFSCompleted(l, t,
18         s1, union(s3, {t'}))
19     □ lockAcquired.l?t':diff(union(s1, s3), {t}) → FCFSCompleted(l, t, diff(s1, {t'}),
20         diff(s3, {t'}))
21
22  FCFSTerminated(l, t) =
23     doorwayComplete.l?t':diff(ThreadID, {t}) → FCFSTerminated(l, t)
24     □ lockAcquired.l?t':diff(ThreadID, {t}) → FCFSTerminated(l, t)

```

```

1  FCFSCheck(l) =
2      || t ← ThreadID • [union({callLock.l.t, end.t}, {doorwayComplete.l, lockAcquired.l})]
3          FCFSNotStarted(l, t, {}, {})
4
5  assert FCFSCheck(L.0) ⊆T FCFSActualSystemRExtDiv \ {lockReleased}

```

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 first consider the implementation of a monitor used by the Java Virtual Machine (JVM), before considering an alternative implementation that addresses some of the limitations of the JVM monitor.

4.1 The JVM monitor

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*.

4.1.1 Modelling the JVM monitor

For our model of a monitor in CSP we have extended the `JVMMonitor` provided by Lowe [21].

Lowe's 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 `MonitorID`, 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 `MonitorID`. The internal channels and processes now also take some `MonitorID` value to identify which object is being referred to at any point.

Internally, the model uses one lock per `MonitorID`. These also have an additional parameter storing the identities of any waiting threads 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 `Reg = CHAOS({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:

Listing 1: The interface of the `JVMMonitor` module; changes are underlined

```

1 module JVMMonitor(MonitorID)
2   ...
3   Reg = CHAOS({spuriousWakeup})
4 exports
5
6   -- All events except spuriousWakeup
7   events = { acquire, release , wait, notify , notifyAll }
8
9   channel spuriousWakeup : MonitorID . ThreadID
10
11  InitialiseAll =
12    ||| mon ← MonitorID • (Unlocked(mon, {}) || {spuriousWakeup} || Reg)
13
14  runWith(obj, P) = P || events || (Unlocked(obj, {}) || {spuriousWakeup} || Reg)
15
16  runWithMultiple(objs, P) =
17  P || events || (|| obj ← objs • (Unlocked(obj, {}) || {spuriousWakeup} || Reg))
18
19  -- Interface to threads.
```

```

20
21  -- Lock the monitor
22  Lock(obj, t) = ...
23
24  -- Unlock the monitor
25  Unlock(obj, t) = ...
26
27  -- Perform P under mutual exclusion
28  Synchronized(obj, t, P) = ...
29
30  -- Perform P under mutual exclusion, and apply cont to the result.
31  -- MutexC :: (ThreadID, ((a) → Proc) → Proc, (a) → Proc) → Proc
32  SynchronizedC(obj, t, P, cont) = ...
33
34  -- Perform a wait(), and then regain the lock.
35  Wait(obj, t) = ...
36
37  -- perform a notify()
38  Notify(obj, t) = ...
39
40  -- perform a notifyAll()
41  NotifyAll (obj, t) = ...
42
43  endmodule

```

Each monitor provides `Wait(obj, t)`, `Notify(obj, 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 function `Func` before completing the rest of the function whilst holding the monitor lock. `runWith(obj, threads)` and `runWithMultiple(objs, threads)` are used to initialise a single monitor with identity `obj` or multiple monitors with identities `objs` respectively to synchronise threads that interact with a single object and multiple objects respectively.

4.2 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 Listing 2. Suppose we have significantly more threads wanting to `get` a value than `put` a value. Each `notifyAll` will awake every 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 awoken by a `notify`; as a result repeated `notifys` could potentially just wake up two threads alternatively, both of which are waiting to perform the same process.

Listing 2: Single placed buffer as an example of the inefficiency of untargeted signals

```
1  class OneBuff[T] {
2    private var buff = null.asInstanceOf[T]
3    private var available = false
4
5    def put(x: T) = synchronized {
6      while(!available) wait()
7      buff = x
8      available = true
9      notifyAll()
10   }
11
12   def get : T = synchronized {
13     while(! available) wait()
14     available = false
15     notifyAll()
16     return x // not overwritable as we still hold the lock
17   }
18 }
```

The SCL monitor implementation solves both of these issues with a single monitor

offering multiple distinct `Conditions` to allow for more targeted signalling. It is worth noting that these improvements result in the SCL monitor being more computationally expensive per call^[Improve sentence]. Each individual condition offers the following operations:

- `await()` is used to wait for a signal on the condition
- `signal()` is used to signal to a thread waiting on the condition
- `signalAll()` is used to signal to all the threads waiting on the condition

Each of these operations should be performed while holding the lock. We can note that these operations are similar to the JVM monitor `wait()`, `notify()` and `notifyAll()` respectively. This functionality is also similar to the `java.util.concurrent.locks.Condition` class; the primary difference is that the SCL monitor blocks spurious wakeups whereas they are allowing by the JAVA class.

Considering the single-placed buffer, we can use two conditions to separate the threads attempting to `get` and those trying to `put`. We can then modify the program above to only perform a single `signal` 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 [22], makes use of the Java `LockSupport` class; we explore this in the next section^[Sub?]. Here we present a low-level model of an SCL Condition which makes use of a model of the `LockSupport` class.

4.2.1 The SCL monitor implementation

The SCL monitor is implemented in two sections: these are a central `Lock` and conditions associated with the central `Lock`. The `Lock` definition is a re-entrant lock (i.e. one thread

can acquire the lock multiple times whilst it holds it) with a variable `locker`: `Thread` which indicates which thread currently holds the `Lock`. The lock implementation is quite simple and for modelling we will instead use a non-reentrant `Lock`; this will reduce the size of the FDR model significantly for model checking without removing functionality.

We instead focus on the `Condition` class. This internally uses a queue of `ThreadInfo` objects to store the identities of the waiting threads and a Boolean indicating if that thread has been signalled. When a thread calls `await()` it releases the lock and enqueues a `ThreadInfo` object into the queue of the respective condition. A signalling thread dequeues a `ThreadInfo` object from the queue (if non-empty), sets the `ThreadInfo`'s `ready` value to `true` indicating that it has been signalled and then unparks the corresponding thread. `signalAll` acts similarly to `signal`, but repeats the above process for all `ThreadInfo` objects in the queue. We note that we have excluded all code regarding interruptions from the SCL code below: this is to reduce the complexity of the resulting CSP model. The interruption handling is also quite simple - it stops the thread from parking again once it has been interrupted and throws a `InterruptedException`.

Listing 3: A subset of the `Condition` class from [22]

```

1 import java.util.concurrent.locks.LockSupport
2
3 /** A condition associated with 'lock'. */
4 class Condition(lock: Lock){
5     /** Information about waiting threads. */
6     private class ThreadInfo{
7         val thread = Thread.currentThread //the waiting thread
8         @volatile var ready = false // has this thread received a signal?
9     }
10
11     /** Check that the current thread holds the lock. */
12     @inline private def checkThread =
13         assert(Thread.currentThread == lock.locker,
14             s"Action on Condition by thread ${Thread.currentThread}, but the "+
15             s"corresponding lock is held by ${lock.locker}")
16

```



```

17  /** Queue holding ThreadInfos for waiting threads.
18   * This is accessed only by a thread holding lock. */
19  private val waiters = new scala.collection.mutable.Queue[ThreadInfo]()
20
21  /** Wait on this condition. */
22  def await(): Unit = {
23      checkThread
24      // record that I'm waiting
25      val myInfo = new ThreadInfo; waiters.enqueue(myInfo)
26      val numLocked = lock.releaseAll // release the lock
27      while(!myInfo.ready){
28          LockSupport.park() // wait to be woken
29      } // reacquire the lock
30      lock.acquireMultiple(numLocked)
31  }
32
33  /** Signal to the first waiting thread. */
34  def signal(): Unit = {
35      checkThread
36      if (waiters.nonEmpty){
37          val threadInfo = waiters.dequeue()
38          if (!threadInfo.ready){
39              threadInfo.ready = true; LockSupport.unpark(threadInfo.thread)
40          }
41          else signal() // try next one; that thread was interrupted or timed out
42      }
43  }
44
45  /** Signal to all waiting threads. */
46  def signalAll() = {
47      checkThread
48      while(waiters.nonEmpty){
49          val threadInfo = waiters.dequeue()
50          threadInfo.ready = true; LockSupport.unpark(threadInfo.thread)
51      }
52  }
53  }

```

4.2.2 The SCL monitor model

DIAGRAM

We now consider our model of the SCL monitor. This consists of three main com-

ponents:

- The monitor lock: as described above, we model the re-entrant lock of the implementation with a single entry lock here. This is because a thread could obtain the lock an unbounded number of times, resulting in a potentially unbounded state space in FDR.

The lock we use is a simple process $\text{Lock}(m) = \text{acquire.m?t} \rightarrow \text{release.m.t} \rightarrow \text{Lock}(m)$ 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. This provides mutual exclusion between threads as required.

- The `LockSupport` module, described in section 4.2.2.1.
- The queue of `ThreadInfo` values, described in section 4.2.2.2.

4.2.2.1 LockSupport

The Java `LockSupport` [REF](#) class offers two main operations: `park` and `unpark`, which are used to suspend and resume a thread respectively. A thread `t` typically calls `LockSupport.park()` to suspend itself until a permit becomes available by another thread calling `LockSupport.unpark(t)`. As this is a permit-based system, a permit is stored if a thread calls `unpark(t)` and `t` is not yet parked. This allows `t` to immediately resume when it does call `park`. It is also important to note that `LockSupport` is also affected by spurious wakeups where some parked thread can resume without an `unpark`.

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. We also use the `spurious` channel to indicate that the following wake-up is

spurious; this will allow us to check that `LockSupport` can only diverge when an infinite number of spurious wake-ups occur.

Internally, the model stores two sets of threads: `waiting` stores the parked threads and `permits` stores the threads 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 if a permit is available. A thread that is performing an `unpark(t)` will either awake `t` if it is currently waiting or store a permit for `t`. We split our definition into two processes: `LockSupport` can either nondeterministically allow a waiting thread to spuriously wake-up, or it can act as the deterministic `LockSupport1` which initially only communicates `park` or `unpark` events. GAVIN???

Listing 4: The CSP model of the Java `LockSupport` module

```

1 module LockSupport
2
3   channel park: ThreadID
4   channel unpark: ThreadID.ThreadID
5   channel wakeUp: ThreadID.Bool
6
7   LockSupport1(waiting, permits) =
8     park?t→ (
9       if member(t, permits)
10        then wakeUp.t → LockSupport(waiting, diff(permits, {t}))
11      else LockSupport(union(waiting, {t}), permits) )
12   □
13   unpark?t?t2→ (
14     if member(t2, waiting)
15      then wakeUp.t2 → LockSupport(diff(waiting, {t2}), permits)
16    else LockSupport(waiting, union(permits, {t2}))
17   LockSupport(waiting, permits) =
18     if waiting = {} then LockSupport1(waiting, permits)
19   else (LockSupport1(waiting, permits)
20     □ spurious → wakeUp$:waiting → LockSupport(diff(waiting, {t}), permits))
21
22
23   LockSupportDet :: ({ThreadID}, {ThreadID}) → Proc
24   LockSupportDet(waiting, permits) = LockSupportDet1(waiting, permits)
25   LockSupportDet1(waiting, permits) = ... — Analogous to LockSupport1

```

```

26
27 exports
28
29 channel spurious
30
31 InitLockSupport = LockSupport({}, {})
32
33 InitLockSupportDet = LockSupportDet({}, {})
34
35 Park(t) = park.t → wakeUp.t?_ → SKIP
36
37 Unpark(t, t') = unpark.t.t' → SKIP
38
39
40 endmodule

```

4.2.2.2 The ThreadInfo Queue

As seen in listing 3, each `Condition` maintains a queue of `ThreadInfo` objects 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 sequence of nodes, a number of processes each corresponding to a node Wording etc and some co-ordinator processes.

We first consider the `ThreadInfo` process. Each `ThreadInfo` process has a corresponding `Node` which remains unchanged throughout; this allows us to use the `Node` datatype to specify which `ThreadInfo` process a thread is interacting with. The `ThreadInfo(n)` process communicates over five channels:

- `channel` `initialiseNode` : `Node` . `ThreadID`. A communication of `initialiseNode.n.t` is used to indicate that `ThreadInfo(n)` will act as thread `t`'s `threadInfo` object until it is released.
- `channel` `releaseNode` : `Node` . `ThreadID`. A communication on this will stop `ThreadInfo(n)` from acting as `t`'s `threadInfo` object, it can now be reinitialised by some thread.

- `channel setReady : Node . ThreadID . ThreadID` is used by some thread to indicate that thread `t` has been signalled; this is only allowed to occur once per initialisation.
- `channel isReady : Node . ThreadID . ThreadID . Bool` is used to indicate whether or not thread `t` has been signalled yet. It is used by `t` to protect against spurious wake-ups, reparking itself if it has awoken without being signalled.
- `channel nodeThread : Node . ThreadID . ThreadID`. A communication on this channel of the form `nodeThread.n.t.t2` indicates to thread `t2` that `ThreadInfo(n)` is currently initialised by thread `t`.

The lifecycle of the `ThreadInfo` process is that some thread `t` will initialise the `ThreadInfo` process with its identity after it is performing an `await`. The `ThreadInfo` process will then act as that thread's `threadInfo` object from the Scala implementation. Its corresponding `Node` value will then be enqueued in the condition's queue. A thread may then dequeue that `Node` value via either a `signal` or `signalAll` call; in both cases it will then call `setReady.n.t?_` to indicate that the corresponding thread has been signalled. Once thread `t` has been unparked and awoken, it checks that it has been signalled via a communication on `isReady`. When this indicates that `t` has been signalled, `t` then communicates a `releaseNode` event to indicate that it has finished using the corresponding `ThreadInfo` process. It can then be re-initialised by some other thread and the above lifecycle repeated. We always allow `ThreadInfo(n)` to communicate on any of its channels, however we diverge whenever some communication occurs that should not be possible in the original Scala code. This allows us to easily check that if the `ThreadInfo` processes are used correctly; any incorrect usage will lead to a divergence which we can find by checking for divergence-freedom later.

Listing 5: The definition of a `ThreadInfo` process

```

1 ThreadInfo :: (Node) → Proc
2 ThreadInfo(n) =
3     initialiseNode . n ? t → ThreadInfoF(n, t)
4     □ isReady . n ? t ? t2 . true → DIV
5     □ setReady . n ? t ? t2 → DIV
6     □ nodeThread . n ? t ? t' → DIV
7     □ releaseNode . n ? t → DIV
8 ThreadInfoF(n, t) =
9     isReady . n ! t ? t2 . false → ThreadInfoF(n, t)
10    □ setReady . n ! t ? t2 . diff (ThreadID, t) → ThreadInfoT(n, t)
11    □ initialiseNode . n ! t → DIV
12    □ nodeThread . n . t ? t' → ThreadInfoF(n, t)
13    □ releaseNode . n ? t → DIV
14 ThreadInfoT(n, t) =
15     isReady . n ! t ? t2 . true → ThreadInfoT(n, t)
16     □ setReady . n ! t ? t2 → DIV
17     □ initialiseNode . n ! t → DIV
18     □ nodeThread . n . t ? t' → ThreadInfoT(n, t)
19     □ releaseNode . n . t → ThreadInfo(n)

```

Since we are handling n threads and all of them can be waiting at the same time, we hence need n `ThreadInfo` processes and hence we define the `Node` datatype as `datatype Node = N.{0..n-1}`.

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.

We note that neither the `NodeAllocator` nor the `Nodes` themselves take a `MonitorID` as a parameter; this is because each thread can only perform an `await` on one condition or monitor at any point in time. This choice allows us to use the same `Nodes` across multiple monitors, leading to significantly smaller FDR state spaces when modelling using multiple monitors.

Listing 6: The `NodeAllocator` process

```

1 NodeAllocator(ns) =

```

```

2      (not(empty(ns))) &
3      (initialiseNode $n:ns?t → NodeAllocator(diff(ns, {n})))
4      □ releaseNode?n:ns?t → DIV
5      □ 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 **Node** identities waiting on its condition. The **Queue** processes have three parameters: **m** is the identity of the SCL monitor and **c** is the identity of the condition that the queue belongs to; **qs** is the current state of the queue.

```

1 Queue(m, c, qs) =
2   (not(null (qs))) & dequeue.m.c?t!head(qs) → Queue(m, c, tail(qs))
3   □ (null (qs)) & isEmpty.m.c?t → Queue(m, c, qs)
4   □ 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.

4.2.3 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.

```

1 Queues(m, setC) = ||| c ← setC • Queue(m, c, <>)
2 NodeSystem =
3   NodeAllocator(Node)
4   [|{initialiseNode , releaseNode}|] (|| n ← Node • ThreadInfo(n))
5
6 InitialiseMon (m, setC) =
7   (Lock(m) ||| Queues(m, setC) ||| NodeSystem ||| InitLockSupport)

```

The first function we export as part of the interface of our module is `runWith(P, mon, setC)` which each takes a process `P` representing program threads, an identity for the monitor and a set of conditions on that monitor. This processes synchronises `P` with the `InitialiseMon(m, setC)`. This is to allow the threads to ‘call’ the various functions that act on the monitor correctly and so that mutual exclusion between threads can be obtained using the functions of the monitor. Include spurious in next?

```

1   ...
2   -- The set of events that are hidden when a thread uses the monitor
3   HideSet(m, setC) =
4       {park, unpark, wakeUp, enqueue.m.c, dequeue.m.c, initialiseNode, nodeThread,
5         setReady, isReady, isEmpty.m.c, releaseNode | c ← setC}
6
7   -- The set of events to synchronise on between a thread and the monitor
8   SyncSet2(m, setC) = Union({{acquire.m}, {release.m}, HideSet(m, setC)})
9
10  exports
11
12  channel spurious
13  channel acquire, release : MonitorID.ThreadID
14  channel callAcquire, callRelease : MonitorID.ThreadID
15  channel callAwait, callSignalAll : MonitorID.ConditionID.ThreadID
16  channel callSignal : MonitorID.ConditionID.ThreadID
17
18  -- Runs the monitor with internal spurious wakeups
19  runWith(P, mon, setC) =
20      ((P [|SyncSet(mon, setC)|]
21         InitialiseMon (mon, setC)) \ HideSet(mon, setC))
22
23  -- Runs the monitor without internal spurious wakeups
24  runWithDet(P, mon, setC) =
25      ((P [|SyncSet(mon, setC)|]
26         InitialiseMonDet (mon, setC)) \ HideSet(mon, setC))
27
28  ...

```

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’. Better convention for placeholder values?

```

1 export
2   ...
3
4   -- Operations on the monitor
5   Await(mon, cnd, t) = callAwait.mon.cnd.t → Await1(mon, cnd, t)
6
7   Signal(mon, cnd, t) = callSignal.mon.cnd.t → Signal1(mon, cnd, t)
8
9   SignalAll(mon, cnd, t) = callSignalAll.mon.cnd.t → SignalAll1(mon, cnd, t)
10
11  Lock(mon, t) = callAcquire.mon.t → acquire.mon.t → SKIP
12
13  Unlock(mon, t) = callRelease.mon.t → release.mon.t → SKIP
14
15 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.

```

1 Signal1(mon, cnd, t) =
2   isEmpty.mon.cnd.t → SKIP
3   □ dequeue.mon.cnd.t?n → nodeThread.n?t2!t → isReady.n.t2.t?b →
4     (if b then Signal1(mon, cnd, t)
5      else setReady.n.t2.t → Unpark(t, t2))

```

```

6
7 SignalAll1 (mon, cnd, t) =
8   isEmpty.mon.cnd.t → SKIP
9   □ dequeue.mon.cnd.t?n → nodeThread.n?t2!t → setReady.n.t2.t →
10      Unpark(t, t2); SignalAll1 (mon, cnd, t)
11
12 Await1(mon, cnd, t) =
13   initialiseNode ?n!t → enqueue.mon.cnd.t.n → release.mon.t → Await2(mon, cnd, t, n)
14
15 Await2(mon, cnd, t, n) =
16   isReady.n.t.t?b → (if b then releaseNode.n.t → acquire.mon.t → SKIP
17                      else Park(t); Await2(mon, cnd, t, n))

```

4.2.4 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 monitor process is parameterised over the identity of the monitor and a map of **ConditionID** => {**ThreadID**} representing the set of threads waiting on each **Condition**. 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. This also is significantly more efficient; it significantly reduces the number of states of the specification process Explanation. **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.

```

1 initSet = mapFromList(<(c, {}) | c ← seq(ConditionID)>)
2 valuesSet (map) = Union({mapLookup(map, cnd) | cnd ← ConditionID})
3 Spec2Unlocked(m, waiting) =

```

```

4   SCL::acquire . m ? t : diff (ThreadID, values (waiting)) → Spec2Locked (m, t, waiting)
5
6   Spec2Locked (m, t, waiting) =
7     □ c' : ConditionID •
8       (
9         (mapLookup (waiting, c') = {}) & SCL::callSignal . m . c' . t →
10          Spec2Locked (m, t, waiting)
11       □ (mapLookup (waiting, c') ≠ {}) &
12         □ t' : mapLookup (waiting, c') • SCL::callSignal . m . c' . t →
13          Spec2Locked (m, t,
14            mapUpdate (waiting, c', diff (mapLookup (waiting, c'), {t'})))
15       )
16     □ SCL::callSignalAll . m ? c : ConditionID ! t →
17       Spec2Locked (m, t, mapUpdate (waiting, c, {}))
18     □ SCL::callRelease . m . t → SCL::release . m . t →
19       Spec2Unlocked (m, waiting)
20     □ SCL::callAwait . m ? c : ConditionID ! t → SCL::release . m . t →
21       Spec2Unlocked (m, mapUpdate (waiting, c, union (mapLookup (waiting, c), {t})))
22
23
24   Spec2Thread (t, m) = SCL::callAcquire . m . t → SCL::acquire . m . t → Spec2Thread2 (t, m)
25   Spec2Thread2 (t, m) =
26     SCL::callAwait . m ? c . t → SCL::release . m . t → SCL::acquire . m . t → Spec2Thread2 (t, m)
27     □ SCL::callRelease . m . t → SCL::release . m . t → Spec2Thread (t, m)
28     □ SCL::callSignalAll . m ? c . t → Spec2Thread2 (t, m)
29     □ SCL::callSignal . m ? c . t → Spec2Thread2 (t, m)
30
31   Spec2SCL = (let m = SigM . S . 0 within
32     (Spec2Unlocked (m, initSet) [SpecChans (m, ConditionID)]
33       (|| t ← ThreadID • (Spec2Thread (t, m)))))
34

```

We first note that the specification process provided is divergence free; we choose this as an idealised monitor should never internally diverge. We use the technique of having multiple threads linear

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.

```

1 ThreadSCL(t) = SCL::Lock(SigM.S.0, t); ThreadSCL1(t)
2 ThreadSCL1(t) =
3   □ c : ConditionID •
4   (
5     (SCL::Await(SigM.S.0, c, t); ThreadSCL1(t))
6     □ (SCL::Signal(SigM.S.0, c, t); ThreadSCL1(t))
7     □ (SCL::Signal(SigM.S.0, c, t); ThreadSCL1(t))
8     □ (SCL::SignalAll(SigM.S.0, c, t); ThreadSCL1(t))
9   )
10  □ (SCL::Unlock(SigM.S.0, t); ThreadSCL(t))
11
12 ThreadsSCL = ||| t ← ThreadID • ThreadSCL(t)
13
14 SCLSystemSpur = SCL::runWith(ThreadsSCL, SigM.S.0, ConditionID)
15 SCLSystem = SCLSystemSpur \ {spurious}
16
17 assert SCLSystemSpur : [divergence free]
18 assert not SCLSystem : [divergence free]

```

We have that both the assertions pass: **SCLSystemSpur** is divergence free, but hiding the **spurious** channel introduces divergences. Since the system is divergence-free with the **spurious** channel visible, we can conclude that spurious wake-ups don't lead to a state where the system can diverge without more spurious wake-ups. We can therefore conclude that the divergences of the system with **spurious** hidden are only due to repeated spurious wakeups

Similarly to [ref](#), we have that this divergence is not a major concern since it relies on infrequent spurious wakeups occurring. We also note that, similarly to before in [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.

```

1 assert SpecSCL ⊑F (SCLSystem)

```

We have that this the assertion holds, 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 **OrderCheck**, a process which maintains a list of the threads waiting on each condition in the order that they started waiting. **OrderCheck** effectively acts as a watchdog, completing a communication over the new channel **error** if an error in ordering is detected; we therefore verify this has not occurred by a trace refinement against the specification.

We choose to use a watchdog as the left hand side of any assertion has to be normalised by FDR. Normalisation is an expensive process for complex specifications with many states; maintaining queues of waiting threads on each condition is therefore more suited to the right-hand side of an assertion as this does not require normalisation. For further details, see [27].

```

1 valuesSeq(map) = Union({set(mapLookup(map, cnd)) | cnd ← ConditionID})
2 channel error : MonitorID
3 OrderCheck(m, waiting) =
4     SCL::acquire .m?t:ThreadID →
5     (if member(t, valuesSeq(waiting)) then error .m → STOP--DIV
6      else OrderCheck(m, waiting))
7   □ SCL::callAwait .m?c?t →
8     (if member(t, valuesSeq(waiting)) then error .m → STOP
9      else OrderCheck(m, mapUpdate(waiting, c, mapLookup(waiting, c)^(t>)))
10  □ SCL::callSignalAll .m?c?_ →
11    OrderCheck(m, mapUpdate(waiting, c, <>))
12  □ SCL::callSignal .m?c?_ →
13    (if null (mapLookup(waiting, c)) then OrderCheck(m, waiting)
14     else OrderCheck(m, mapUpdate(waiting, c,
15                                tail (mapLookup(waiting, c)))))

```

This new process only synchronises on the events ; 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`. We only synchronise on events that indicate a thread waking, waiting or acquiring the lock; these are all the communications offered by `OrderCheck` except for `error.m`.

We then check that this still trace refines `SpecSCL`, 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.

```

1 assert SpecSCL  $\sqsubseteq_T$  (OrderCheck(SigM.S.0, initSeq)
2   [|{SCL::callAwait.SigM.S.0,
3     SCL::acquire.SigM.S.0,
4     SCL::callSignal.SigM.S.0,
5     SCL::callSignalAll.SigM.S.0}|] SCLSystem)

```

4.2.5 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 `ThreadInfo` allows any thread to obtain and use any of the `Nodes`; 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`. This forces each thread to use the same `ThreadInfo` each time, 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:

```

1 Await1(mon, cnd, t) = initialiseNode .N.t  $\rightarrow$  ...
2
3 NodeAllocator(ns) =
4   (not(empty(ns))) & (initialiseNode ?n:ns?t  $\rightarrow$  NodeAllocator(diff(ns, {n})))
5   □ ...

```

We will refer to this version of the queue as the ‘Simple’ model. This simplified model has one possible bijective mapping of threads to nodes. By contrast, the natural model 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.

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:

```

1  datatype Node = N.ThreadID
2  channel enqueue: MonitorID.ConditionID.Node
3  channel dequeue: MonitorID.ConditionID.ThreadID.Node
4  channel setReady: Node.ThreadID
5  channel isReady: Node.ThreadID.Bool
6  channel initialiseNode : Node
7  channel isEmpty: MonitorID.ConditionID.ThreadID
8  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 put raw code in an appendix?. We keep `initialiseNode` to so that a thread can use it to indicate it is initialising a ‘new’ `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 wording to the natural queue model.

We next verify that the efficiency improvements occurs in practice too. We do

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. threads	No. conditions	Number of states		Improvement	to optimised	Improvement
		Natural	Simple			
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

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. This is as expected due to the reduction in the mappings from threads to nodes as stated above. 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.

Introduce efficient spec version of SCL monitor and compare performance?.

5 Barrier synchronisation

A *barrier synchronisation* object is used to synchronise some number of threads. 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 synchronisations* (synchronising all threads) typically operate by instantiating some barrier object and then having each thread call `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 afterwards [18].

Here we model and analyse an `n` thread barrier synchronisation object which internally uses a binary heap of `n` two-thread signalling objects.

5.1 The signalling 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 7.

Listing 7: The Scala code for the Signal object

```
1 private class Signal{
2   /** The state of this object. true represents that the child has signalled,
3     * but not yet received a signal back. */
4   private var state = false
5
6   /** Signal to the parent, and wait for a signal back. */
7   def signalUpAndWait = synchronized{
8     require(!state,
9       "Illegal state of Barrier: this might mean that it is\n"+
10      "being used by two threads with the same identity.");
11     state = true; notify()
12     while(state) wait()
13   }
14
15   /** Wait for a signal from the child. */
16   def waitForSignalUp = synchronized{ while(!state) wait() }
17
18   /** Signal to the child. */
19   def signalDown = synchronized{ state = false; notify() }
20 }
```

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 the parent, awaking them if they are waiting. 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 again guards 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.

5.1.1 Modelling the Signal objects

We use our model for a JVM monitor from [REF] in our modelling of the **Signal** object. [Wording] An interacting thread will run the CSP process of the equivalent Scala function, here referred to as **Func**. In each case these first communicates a **callFunc** before running the **syncFunc** process within a **Synchronized** block as outlined previously in section [REF] 1.

From signal-scala.csp

Listing 8: The state variable(s) and the function call channels

```
1 channel getState, setState : SignalID . ThreadID . Bool
2 stateChannels(s) = {getState.s, setState.s}
3
4 State :: (SignalID) → Proc
5 State(s) = Var(false, getState.s, setState.s)
6
7 channel callSignalUpAndWait, callWaitForSignalUp, callSignalDown : SignalID . ThreadID
8 signalChannels = {callSignalUpAndWait, callWaitForSignalUp, callSignalDown}
9
```

We first initialise the **state** variable, with the **SignalID** parameter in the channels indicating the **Signal** object the variable corresponds to. We also introduce the function call channels as indicated above.

Listing 9: The CSP model of the **signalUpAndWait** function of the **Signal** object

```
1 SignalUpAndWait :: (SignalID, ThreadID) → Proc
2 SignalUpAndWait(s, t) =
3   callSignalUpAndWait.s.t → Synchronized(SigM.s, t, syncSignalUpAndWait(s, t))
4 syncSignalUpAndWait(s, t) =
5   getState.s.t?val → if val = true then DIV — Required to be false
6                     else setState.s.t.true →
7                       Notify(SigM.s, t); SignalWaitingForFalse(s, t)
8
9
10 SignalWaitingForFalse :: (SignalID, ThreadID) → Proc
11 SignalWaitingForFalse(s, t) =
12   getState.s.t?val → if val = false then SKIP
13                     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.

Listing 10: The CSP model of the `waitForSignalUp` function of the `Signal` object

```

1 WaitForSignalUp :: (SignalID, ThreadID) → Proc
2 WaitForSignalUp(s, t) =
3   callWaitForSignalUp.s.t → Synchronized(SigM.s, t, syncWaitForSignalUp(s, t))
4 syncWaitForSignalUp(s, t) =
5   getState.s.t?val → if val = true then SKIP
6                       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.

Listing 11: The CSP model of the `signalDown` function of the `Signal` object

```

1 SignalDown :: (SignalID, ThreadID) → Proc
2 SignalDown(s, t) = callSignalDown.s.t → Synchronized(SigM.s, t, syncSignalDown(s, t))
3 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.

Listing 12: The initialisation of the `Signal` objects,

```

1 InitialiseSignal (sig, threads) =
2   runWith(SigM.sig, threads [|stateChannels(sig)|] State(sig))
3   \ union(stateChannels(sig), events)
4
5 allStateChannels (sigs) = {getState.s, setState.s | s ← sigs}
6 States(sigs) = ||| s ← sigs • State(s)
7 monitors(sigs) = {SigM.s | s ← sigs}
8
9 InitialiseSignals (sigs, threads) =
10  runWithMultiple(monitors(sigs), threads [|allStateChannels(sigs)|] States(sigs))

```

We finally define processes `InitialiseSignal` and `InitialiseSignals` to initialise a single signal and multiple signals respectively. 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. `runWith` and `runWithMultiple` are both used to initialise the monitors for each of the `Signal` objects that are being modelled, ensuring mutual exclusion between threads on each of the `Signal` objects.

5.2 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:

Listing 13: The Scala definition of the `Barrier.sync` function

```

1  /** Perform a barrier synchronisation.
2   * @param me the unique identity of this thread. */
3  def sync(me: Int) = {
4    require(0 <= me && me < n,
5      s"Illegal parameter $me for sync: should be in the range [0..$n).")
6    val child1 = 2*me+1; val child2 = 2*me+2
7    // Wait for children
8    if (child1 < n) signals(child1).waitForSignalUp
9    if (child2 < n) signals(child2).waitForSignalUp
10   // Signal to parent and wait for signal back
11   if (me != 0) signals(me).signalUpAndWait
12   // Signal to children
13   if (child1 < n) signals(child1).signalDown
14   if (child2 < n) signals(child2).signalDown
15 }

```

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

children are ready to synchronise. Once the parent signals back that the synchronisation has occurred the thread notifies its children that the synchronisation has completed before returning. The exception to this is thread 0, which has no parent to signal to. Thread 0 reaching line 11 of its `sync(0)` call can therefore be taken as the linearization point of the barrier synchronisation.

5.2.1 Modelling the Barrier object

Listing 14: The CSP model of a thread interacting with the Barrier object

```

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

```

We recall from earlier that `datatype ThreadID = T.{0..n-1}` and `datatype SignalID = S.{0..n-1}`. 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 communicate an `end.T.me` and terminate or to call the CSP model of `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 it mostly following directly 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 $\{\text{beginSync}, \text{endSync}, \text{spuriousWakeup}, \text{end}\}$.

5.3 Correctness of the model

We will show that the barrier synchronisation is correct; here correct requires that the synchronisation can be correctly linearised and if a synchronisation is possible then it will always occur.

Correctly linearised in this context 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**.

Listing 15: The lineariser specification for barrier synchronisations

```

1 Lineariser (t) = beginSync.t → sync → endSync.t → Lineariser(t)
2               □ end.t → STOP
3 Spec = ( || t ← ThreadID • [{beginSync.t, sync, endSync.t, end.t}]
4         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 blocks all other threads from completing a barrier synchronisation, which is the intended behaviour.

We first note that **BarrierSystem** is divergence-free, but **BarrierSystem** $\setminus \{\text{spuriousWakeup}\}$ is not. **BarrierSystem** with spurious wakeups visible being divergence-free is relevant

as this means that we never breach the assertion in the `SignalUpAndWait` function; this therefore means hiding the `spuriousWakeup` events must be the cause of the divergences check that isn't an

This is expected behaviour as one thread could spuriously wakeup, check the test condition and wait again before spuriously waking up like this indefinitely. Similarly to before this is not a particular cause for concern; in practice spurious wakeups occur infrequently within the JVM.

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

1 `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` cannot diverge we will also consider refinement under the stable failures model 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 stable state where the regulator process `Reg` blocks the spurious wakeup. FDR yields that both the following hold for systems of upto 6 threads in Time value: 1277 seconds

1 `assert Spec \sqsubseteq_F BarrierSystem \ \{spuriousWakeup\}`

As a result, we have that the `Barrier` object presented earlier is a correct implementation of barrier synchronisation for `n` upto 6. confidence arg here?

5.4 Specification processes for the Signal objects

Check 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 significant state space explosion, resulting in us only being able to test for correctness on models with upto 6 threads. We can instead construct a specification process which models the use of the **Signal** object. Though this still results in a model size exponential in the number of threads, 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 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:

Listing 16: The CSP model of the specification **Signal** object

```
1 channel endWaitForSignalUp, endSignalUpAndWait : SignalID . ThreadID
2 waitChannels = {endWaitForSignalUp, endSignalUpAndWait}
3
4 -- Simplified spec for a correctly used Signal object
```

```

5 SpecSig(s) =
6   callSignalUpAndWait.s?t → callWaitForSignalUp.s?t2 → SpecSig2(s, t, t2)
7   □ callWaitForSignalUp.s?t2 → callSignalUpAndWait.s?t → SpecSig2(s, t, t2)
8 SpecSig2(s, t, t2) =
9   endWaitForSignalUp.s.t2 → callSignalDown.s.t2 → endSignalUpAndWait.s.t → SpecSig(s)
10
11 -- The individual functions for the Signal object
12 SpecSignalUpAndWait(s, t) = callSignalUpAndWait.s.t → endSignalUpAndWait.s.t → SKIP
13 SpecWaitForSignalUp(s, t) = callWaitForSignalUp.s.t → endWaitForSignalUp.s.t → SKIP
14 SpecSignalDown(s, t) = callSignalDown.s.t → SKIP
15
16 -- Construct the system for each of the SpecSig objects
17 SpecSignals =
18   || s ← SignalID • [{callSignalUpAndWait.s, callWaitForSignalUp.s,
19                       callSignalDown.s, endWaitForSignalUp.s, endSignalUpAndWait.s}]
20                       SpecSig(s)
21
22 -- Method for barrier-sync to initialise the two objects
23 InitialiseSpecSignals (threads) =
24   (SpecSignals [|union(signalChannels, waitChannels)|] threads) \ waitChannels
25

```

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 communicates a `callSignalDown` indicating that the barrier synchronisation has occurred.
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 similarly to the above, with the exception that all calls are to the specification processes and not the originals.

Stuff about how the initial implementation of `Signal` fulfils this specification

Listing 17: The implementation of the `Barrier` based on

```

1  -- sThread is the same as Thread but uses the spec Signal
2  sThread(T.me) = beginSync.T.me → sSync(T.me) □ end.T.me → SKIP
3  sSync(T.me) =
4    let child1 = 2*me+1
5      child2 = 2*me+2
6    within (if (child1 < n) then SpecWaitForSignalUp(S.(child1), T.me) else SKIP);
7      (if (child2 < n) then SpecWaitForSignalUp(S.(child2), T.me) else SKIP);
8      (if me ≠ 0 then SpecSignalUpAndWait(S.me, T.me) else SKIP);
9      (if (child1 < n) then SpecSignalDown(S.(child1), T.me) else SKIP);
10     (if (child2 < n) then SpecSignalDown(S.(child2), T.me) else SKIP);
11     endSync.T.me → sThread(T.me)
12
13  -- Initialise the simple system
14  sThreads = ||| t : ThreadID • sThread(t)
15  sBarrierSystem = InitialiseSpecSignals (sThreads)
16
17  -- Spec failure—divergences refines it as expected
18  assert Spec ⊆FD sBarrierSystem

```

Since the simplified `Signal` object does not use monitors we have that the system should be divergence-free. This is verified 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

6 Conclusions and future work

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 technically possible, the exponential blow up in the number of states renders it practically 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; we do however note that this does not necessarily imply correctness.

We take, for example, the SCL monitor which we have previously proved correct for six threads and two conditions in section [ref](#). The components of each individual condition are distinct with the exception of the `ThreadInfo` objects; any flaw with this would be expected to show itself with only two conditions. Likewise any issue due to interfering threads must require at least seven interacting threads; this seems remarkably unlikely due to the simplicity of the system and limited possible interactions between threads. [counter abstraction?](#)

We have also only considered a number of specific concurrency primitives. Though our approach can be extended to many other primitives, this would still require significant work to verify the correctness of these. An automated translation system from normal code to CSP would aid in this task, however any general translator would likely 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 space- $n!$ times larger than a model with additional insight. Though a naïve translation is very feasible, the utility of such an approach is limited, though non-zero. Implementing an automated translator, either optimised or naïve, 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] Centre for Digital Trust. *Interview with Prof. Viktor Kunčak on Formal Software Verification and Stainless*. Newsletter. 2019. URL: <https://mailchi.mp/6d0f3c2d3070/c4dt-newsletter-02?e=ec2376a2c8>.
- [8] 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>.
- [9] Bill Roscoe Gavin Lowe. *Concurrency Lecture Slides*. 2020. URL: <https://www.cs.ox.ac.uk/teaching/materials19-20/concurrency/>.
- [10] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012. ISBN: 9780123973375.
- [11] Andrew Jones. *An Introduction to NASA’s Java Pathfinder*. 2010. URL: <https://www.doc.ic.ac.uk/~wlj05/ajp/lecture6/slides/slides.pdf>.
- [12] Cliff Jones. “Tentative steps toward a development method for interfering programs”. In: *ACM Transactions on Programming Languages and Systems* 5.4 (1983).
- [13] 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.
- [14] 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.
- [15] 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.
 - [16] 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](https://doi.org/10.1016/0020-0190(95)00144-2). URL: <https://www.sciencedirect.com/science/article/pii/0020019095001442>.
 - [17] Gavin Lowe. *Analysing Lock-Free Linearizable Datatypes using CSP*. 2017. URL: <https://www.cs.ox.ac.uk/people/gavin.lowe/LockFreeQueue/lockFreeListAnalysis.pdf>.
 - [18] Gavin Lowe. *Concurrent Programming Lecture Slides*. 2022. URL: <https://www.cs.ox.ac.uk/teaching/materials22-23/concurrentprogramming/>.
 - [19] 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>.
 - [20] Gavin Lowe. “Implementing Generalised Alt”. In: *Communicating Process Architectures 2011* (2011).
 - [21] Gavin Lowe. *JVM Monitor Module*. GitHub. 2023. URL: <https://github.com/GavinLowe1967/SCL-CSP-analysis/blob/main/JVMMonitor.csp>.

- [22] Gavin Lowe. *Scala Concurrency Library - lock.scala*. 2022. URL: <https://github.com/GavinLowe1967/Scala-Concurrency-Library/blob/main/src/Lock/Lock.scala>.
- [23] 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>.
- [24] 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/>.
- [25] 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/>.
- [26] A. W. Roscoe and Thomas Gibson-Robinson. *The relationship between CSP, FDR and Büchi automata*. 2016. URL: <https://www.cs.ox.ac.uk/files/8301/infinitec.pdf>.
- [27] Andrew (Bill) Roscoe. *Understanding Concurrent Systems*. Springer, 2010.
- [28] Gerhard Schellhorn and Simon Bäuml. “Formal Verification of Lock-Free Algorithms”. In: *Ninth International Conference on Application of Concurrency to System Design* (2009).
- [29] 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>.