4yp

Thomas Aston

May 18, 2024

Contents

- Change figure linkings
- Lineariser spelling??

1 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.

1.1 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*.

1.1.1 Modelling the JVM monitor

For our model of a monitor in CSP we have extended the JVMMonitor provided by Lowe [1].

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 reffered 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:

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

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

    channel spuriousWakeup : MonitorID . ThreadID

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

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

    runWithMultiple(objs, P) =
    P [| events |] (||| obj <- objs • (Unlocked(obj, {}) [| {|spuriousWakeup|} |] Reg))

    -- Interface to threads.

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

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

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

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

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

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

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

endmodule
```

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

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 fucntion Func before completing the rest of the function whilst holding the monitor lock. runWith(obj, threads) and runWithMultiple(objs, threads) are used to initalise 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.

1.2 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 1. 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 awoke by a notify; as a result repeated notifys could potentially just wake up two thread alternatively, both of which are waiting to perform the same process.

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

```
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 threasd 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, available on GitHub[1], 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.

1.3 LockSupport

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

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

Listing 2: The CSP model of the Java LockSupport module

```
1  module LockSupport
2
3    channel park: ThreadID
4    channel unpark: ThreadID.ThreadID
5    channel wakeUp: ThreadID.Bool
6
7    LockSupport :: ({ThreadID}, {ThreadID}) → Proc
8    LockSupport(waiting, permits) =
9      if waiting = {} then LockSupport1(waiting, permits)
10     else (     LockSupport1(waiting, permits)
11           □ wakeUp$t:waiting!false → LockSupport(diff(waiting, {t}), permits))
12
13   LockSupport1(waiting, permits) =
14     park?t→ (
15       if member(t, permits)
```

[1] https://github.com/GavinLowe1967/Scala-Concurrency-Library/blob/main/src/Lock/Lock.scala

```
16            then wakeUp.t.true → LockSupport(waiting, diff(permits, {t}))
17          else LockSupport(union(waiting, {t}), permits) )
18        □
19        unpark?t?t2→ (
20          if member(t2, waiting)
21            then wakeUp.t2.true → LockSupport(diff(waiting, {t2}), permits)
22          else LockSupport(waiting, union(permits, {t2})))
23
24
25      LockSupportDet :: ({ThreadID}, {ThreadID}) → Proc
26      LockSupportDet(waiting, permits) = LockSupportDet1(waiting, permits)
27      LockSupportDet1(waiting, permits) = ... -- Analogous to LockSupport1
28
29    exports
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
```

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

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

1.4 The SCL monitor model

DIAGRAM

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

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

1.4.1 The ThreadInfo Queue

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

Since we are handling n threads and all of them can be waiting at the same time, we hence need n separate nodes, which we will represent as datatype Node = N.{0..n−1}. Each of the nodes can be initialised by a thread, at which point it then acts as that thread's ThreadInfo object. It allows threads (including its allocated thread) to check the value of ready via a communication on the isReady channel, and for other threads to set the value of ready to true via the setReady channel. Once the parent thread has reawoken legitimately (i.e. via an Unpark not a spurious wakeup) it then releases the node; this is valid as the node must have already been dequeued and setReady, hence no further communications should happen until it is reinitialised. The ThreadInfo objects

are designed to diverge whenever a communcation occurs that should not be possible in the original Scala code.

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

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

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

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

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

We have that each Queue' is always ready to accept an enqueue (unless all nodes

are already in the queue), but will only communicate one of dequeue or isEmpty at any point in time. We note that the restriction on the values of n that can be enqueued is such that the queue is of finite length; this is required for efficient model checking in FDR.

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

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

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

```
1  -- The set of events that are hidden when a thread uses the monitor
2  HideSet(m, setC) =
3    {|park, unpark, wakeUp, enqueue.m.c, dequeue.m.c, initialiseNode, nodeThread,
4      setReady, isReady, isEmpty.m.c, releaseNode | c ← setC|}
5
```

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

  exports

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

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

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

  ...
```

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

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

```
export
  ...

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

```

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

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

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

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

endmodule
```

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

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

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

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

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

1.6 Correctness

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

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

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

```
1 initSet  = mapFromList(<(c, {}) | c ← seq(ConditionID)>)
2 values(map) = Union({mapLookup(map, cnd) | cnd ← ConditionID})
3 channel callSignalSpec : MonitorID.ConditionID.ThreadID.ThreadID
4
5 SpecUnlocked(m, waiting, poss) =
6     SCL::callAcquire.m?t':diff(ThreadID, union(values(waiting), poss)) →
7       SpecUnlocked(m, waiting, union(poss,{t'}))
8   □ SCL::acquire.m?t:poss → SpecLocked(m, t, waiting, diff(poss,{t}))
```

```
9
10
11  SpecLocked(m, t, waiting, poss) =
12    □ c': ConditionID •
13        (
14            (mapLookup(waiting, c') = {}) & callSignalSpec.m.c'.t.t →
15              SpecLocked(m, t, waiting, poss)
16          □ (mapLookup(waiting, c') ≠ {}) &
17              callSignalSpec.m.c'.t?t':mapLookup(waiting, c') →
18                SpecLocked(m, t,
19                           mapUpdate(waiting, c', diff(mapLookup(waiting, c'), {t'})),
20                           union(poss, {t'}))
21        )
22    □ SCL::callSignalAll.m?c:ConditionID!t →
23        (if mapLookup(waiting, c) = {} then
24              SpecLocked(m, t, waiting, poss)
25         else SpecLocked(m, t, mapUpdate(waiting, c, {}),
26                         union(poss, mapLookup(waiting, c))))
27    □ SCL::callRelease.m.t →
28          SpecLockedReleasing(m, t, waiting, poss)
29    □ SCL::callAcquire.m?t':diff(ThreadID,
30                                 Union({values(waiting), poss, {t}})) →
31          SpecLocked(m, t, waiting, union(poss, {t'}))
32    □ SCL::callAwait.m?c:ConditionID!t →
33          SpecLockedWaiting(m, c, t, waiting, poss)
34
```

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

```
1   -- t doing a wait; needs to release the lock
2   SpecLockedWaiting(m, c, t, waiting, poss) =
3         SCL::release.m.t →
4           SpecUnlocked(m, mapUpdate(waiting, c, union(mapLookup(waiting, c), {t})),
5                        poss)
6     □ SCL::callAcquire.m?t':diff(ThreadID,
7                                  Union({values(waiting), poss, {t}})) →
8           SpecLockedWaiting(m, c, t, waiting, union(poss, {t'}))
9
10  -- t releasing the lock
11  SpecLockedReleasing(m, t, waiting, poss) =
```

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

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

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

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

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

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

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

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

We have that both the assertions pass: SCLSystem is not divergence free, but SCLSystemDet is. Since the only difference between SCLSystem and SCLSystemDet is

that we block spurious wakeups in the latter, we can therefore conclude that divergence is only possible as a result of repeated spurious wakeups of waiting threads. Similarly to ref, we have that this potential divergence is not a major concern since it relies on infrequent spurious wakeups occuring. We also note that, similarly to before in 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)
2  assert SpecSCL ⊑FD (SCLSystemDet)
```

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

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

```
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)))))
```

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

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

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

1.7 Limitations of natural model of the queue

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

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

allocated by NodeAllocator i.e. as follows:

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

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

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

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

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

All the definitions remain the same apart from removing any nodeThread and releaseNode communications and the required type changes 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.

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				
		Natural	Simple	Improvement	optimised	Improvement
2	1	2288	1088	2.10	904	2.53
3	1	239428	36262	6.60	26494	9.04
4	1	3.14×10^7	1180416	26.7	792240	39.7
5	1	5.39×10^9	4.06×10^7	133	2.59×10^7	208
2	2	4932	2382	2.07	2382	2.40
3	2	686896	106672	6.44	82973	8.27
4	2	1.22×10^8	4655652	26.2	3363492	36.3
2	3	8436	4106	2.05	3634	2.32
3	3	1445008	227512	6.35	184276	7.84
4	3	3.15×10^8	1.22×10^7	25.9	9212868	34.2

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

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

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

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

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

References

[1] Gavin Lowe. *JVM Monitor Module*. GitHub. 2023. URL: https://github.com/GavinLowe1967/SCL-CSP-analysis/blob/main/JVMMonitor.csp.