Postscript to Analysing Lock-Free Linearizable Datatypes using CSP

Gavin Lowe

April 13, 2022

This note is a postscript to my earlier paper *Analysing Lock-Free Linearizable Datatypes using CSP*. I describe how to improve the analysis described there. I assume that the reader has read that paper.

Recall that the earlier paper captured the notion of linearization by

- Building a CSP model corresponding to the abstract datatype that the concurrent datatype aims to implement, with events corresponding to the operations on that abstract datatype; we call these events *abstract linearization points*. In the earlier paper, this abstract datatype was a bounded queue.
- For each thread t, a *linearizer process*, that observes the calls and returns of operations by t, and ensures that the corresponding abstract linearization point occurs between each.

Figure 1 gives such a specification for a queue, in a very similar style to the earlier paper. Note that the abstract linearization points are hidden.

A shortcoming of this approach is that the specification has to "guess" which abstract linearization point to perform (internally) at each stage. More precisely, while model checking, the normalised specification will be in a state corresponding to the nondeterministic choice over all possible sequences of abstract linearization points that might have occurred. This gives an increase in the number of states that the model checker has to explore.

However, for many concurrent datatypes, we can identify concrete linearization points within the code for at least some cases. Checking becomes more efficient if we can arrange for the specification to observe these concrete linearization points, i.e. to take the abstract and concrete linearization points to be the same. We illustrate this technique below, using the example of a lock-free queue from the earlier paper, and considering each linearization point in turn.

```
QueueSpec = Queue(<>)
Queue(q) =
  ( if  q=<> then dequeueEmpty?_ → Queue(q)
   else  dequeue?_!head(q) → Queue(tail(q)) )
  □
  (□ t : ThreadID •
      if  #q+1 ≤ card(NodeID) then
        enqueue.t?x → Queue(q^<x>) ⊓ enqueueFull.t → Queue(q)
     else  enqueueFull.t  → Queue(q) )

 Linearizer (t) =
   beginEnqueue.t?value → (
     enqueue.t.value → endEnqueue.t.value → Linearizer(t)
     □ enqueueFull.t → endEnqueueFull.t.value → Linearizer(t) )
   □
   beginDequeue.t → (
     dequeueEmpty.t → endDequeueEmpty.t → Linearizer(t)
     □ dequeue.t?value → endDequeue.t.value → Linearizer(t) )

 AllLinearizers  =  ||| t : ThreadID • Linearizer(t)
specSyncSet = {| enqueue, dequeue, dequeueEmpty, enqueueFull |}
Spec0 = ( AllLinearizers  [| specSyncSet |] QueueSpec)
Spec =  Spec0  \  specSyncSet

System0 = ... -- system with all events visible
hideSet  = ... -- all events except begin and end events
System = System0  \  hideSet
assert  Spec  ⊑F  System
```

Figure 1: Capturing linearizability for a queue.

Unsuccessful enqueues. We start with the easiest case, namely an enqueue operation that fails because there is no free node, corresponding to the abstract linearization points enqueueFull.t, above. In the model of the implementation from the earlier paper, a thread t performs an event noFreeNode.t to discover that there is no free node. We can take this latter event to be the concrete linearization point. In order to equate the abstract and concrete linearization points, we perform a renaming, as follows.

```
System1 = System0[[noFreeNode.t ← enqueueFull.t | t ← ThreadID]] \ hideSet
Spec1 = Spec0 \ {| enqueue, dequeue, dequeueEmpty |}
assert Spec1 ⊑F System1
```

Note that the enqueueFull.t events are visible in both sides of the refinement, so they correspond.

Successful enqueues. We consider now the case of a successful enqueue. In the lock-free queue, this corresponds to a particular successful CAS operation, where a thread performs a CAS operation on the next reference of the last node of the list:

```
  CASnext.me.myTail.Null.node?result → ...
```

We would like to rename this event (if result is true) to the corresponding abstract linearization point enqueue.me.value. However, there are two difficulties to overcome:

- CASnext events are used elsewhere in the model for different purposes; we do not want to equate these with linearization points;
- The above CASnext event does not contain a field corresponding to value, which is used in the abstract linearization point.

We can deal with both of these difficulties by adding an extra field to CASnext events, to record information about linearization:

```
datatype EnqLPInfo = NoLP | EnqLP.T
channel CASnext :
  ThreadID.NodeIDType.NodeIDType.NodeIDType.Bool.EnqLPInfo
```

The instance corresponding to the linearization point of an enqueue receives a EnqLP.value field, but only if the CAS is successful:

```
  CASnext.me.myTail.Null.node?result!
    (if result then EnqLP.value else NoLP) → ...
```

Every other instance of a CASnext performed by a thread receives a NoLP field. The nodes allow any EnqLPInfo field, and ignore it.

We can then perform a renaming to equate the abstract and concrete linearization points.

```
System2 =
  System0⟦noFreeNode.t ← enqueueFull.t | t ← ThreadID⟧
          ⟦CASnext.t.tail.Null.n.true.EnqLP.value ← enqueue.t.value |
             t ← ThreadID, tail ← NodeID, n ← NodeID, value ← T⟧
         \ hideSet
Spec2 = Spec0 \ {| dequeue, dequeueEmpty |}
assert Spec2 ⊑_F System2
```

Successful dequeues. We consider next the case of a successful dequeue. In the lock-free queue, this corresponds to a successful CAS operation upon the shared Head variable:

```
  CAS.me.Head.myHead.next?result → ...
```

We want to equate this event (if result is true) with the corresponding abstract linearization event, dequeue.me.value. However, a difficulty here is that the dequeueing thread does not yet know the value being dequeued: it learns this value in the next event. We deal with this by performing a pair of renamings:

- In the implementation, we rename the CAS event to a new event deqLP.t, indicating that this is the linearization of a successful dequeue by thread t of *some* value;
- In the specification, we rename the abstract linearization point dequeue.t.value to the same event deqLP.t.

```
channel deqLP: ThreadID
System3 =
  System0⟦noFreeNode.t ← enqueueFull.t | t ← ThreadID⟧
          ⟦CASnext.t.tail.Null.n.true.EnqLP.value ← enqueue.t.value |
             t ← ThreadID, tail ← NodeID, n ← NodeID, value ← T⟧
          ⟦CAS.t.Head.head.n.true ← deqLP.t |
             t ← ThreadID, head ← NodeIDType, n ← NodeID⟧
          \ hideSet
Spec3 = Spec0⟦dequeue.t.value ← deqLP.t | t ← ThreadID, value ← T⟧
            \ {| dequeueEmpty |}
assert Spec3 ⊑_F System3
```

The thread will indicate the value it believes it has dequeued in a subsequent event endDequeue.t.value. Likewise, the specification will indicate the value that should be dequeued in its own endDequeue.t.value. The refinement assertion requires these values to agree.

Unsuccessful dequeues It appears not to be possible to use a similar technique for unsuccessful dequeues. In this case, a thread performs the following sequence of actions: (1) read head; (2) read tail; (3) read the next field of the node read from head; (4) reread head; (5) if the two reads of head and the read of tail all gave the same node, and the read of the next node gave null, then return that the dequeue is unsuccessful. A bit of thought reveals that step (3) is the appropriate linearization point: the queue is certainly empty at this point; but it might not be empty subsequently. However, it is the linearization point *only if* the subsequent reread of head in step (4) does indeed give the same value as earlier. But there is no way of telling at step (3) whether this will be true, so we cannot straightforwardly use step (3) as the concrete linearization point in the refinement check.

A pragmatic approach at this point is to use the technique from the earlier paper (and as in the assertion Spec3 $\sqsubseteq_F$ System3). However, it is possible to treat step (3) as a *provisional* linearization point, which might be overruled by a later linearization point by the same thread. We illustrate the technique, even though it actually makes checking slightly slower in this case.

As with successful enqueues, we need to deal with the fact that not every read of a next field corresponds to a linearization point. We add a boolean field to the corresponding channel, with a value of true indicating a provisional linearization point of an unsuccessful dequeue.

```
channel getNext : ThreadID . NodeIDType . NodeIDType . Bool
```

Then the dequeueing thread can set the field to be true if null is read:

```
  getNext.me.myHead?next!(next==Null) → ...
```

And for other getNext events, the thread can set the field to be false.

We can then rewrite the linearizers so as to make dequeueEmpty events into provisional abstract linearization points, that can (nondeterministically) subsequently be overruled by another linearization point.

```
Linearizer (me) =
  beginEnqueue.me?value → (
    enqueue.me.value → endEnqueue.me.value → Linearizer(me)
    □ enqueueFull.me → endEnqueueFull.me.value → Linearizer(me) )
  □
  beginDequeue.me → LinDeq(me)
LinDeq(me) =
  dequeueEmpty.me → (endDequeueEmpty.me → Linearizer(me) ⊓ LinDeq(me))
  □ dequeue.me?value → endDequeue.me.value → Linearizer(me)
```

Note that it's important that the dequeueEmpty abstract linearization point has no effect on the state of the abstract datatype: in the case that it is subsequently overruled, it should have no effect.

The provisional abstract and concrete linearization points can then be equated via a renaming, as previously.

```
System4 =
  System0⟦noFreeNode.t ← enqueueFull.t | t ← ThreadID⟧
         ⟦CASnext.t.tail.Null.n.true.EnqLP.value ← enqueue.t.value |
            t ← ThreadID, tail ← NodeID, n ← NodeID, value ← T⟧
         ⟦CAS.t.Head.head.n.true ← deqLP.t |
            t ← ThreadID, head ← NodeIDType, n ← NodeID ⟧
         ⟦getNext.t.head.Null.true ← dequeueEmpty.t |
            t ← ThreadID, head ← NodeIDType⟧
         \ syncSet
Spec4 = Spec0⟦dequeue.t.value ← deqLP.t | t ← ThreadID, value ← T⟧
assert Spec4 ⊑F System4
```

Progress properties So far we have been concentrating on linearizability. This is a safety property, so we could have performed the tests in the traces model. By using the stable failures model (and assuming the system is divergence-free), we also capture progress properties. We discuss those progress properties in more detail here.

Consider, again, the original specification and the check Spec $\sqsubseteq_F$ System (Figure 1). Suppose that after some trace there is a pending call (i.e. a begin event has happened without the corresponding end event). Because the Queue process always offers at least one of the corresponding abstract linearization events, and because these events are hidden, in the corresponding stable state of the specification, an end event is available (in the case of an enqueue, the specification allows either an enqueue or an enqueueFull event). Thus the refinement assertion requires that the implementation cannot stably refuse these events. The assertion requires that each pending call should be able to return.

This property is weaker than the standard property of lock freedom. That property allows threads to be permanently descheduled. But in the CSP model, most states where a thread is permanently descheduled are unstable, so have no stable failures. Modern schedulers do not permanently deschedule threads, so we should not be concerned about such states.

The stable states in the lock-free queue model correspond to where every thread is about to perform a begin or end event. However, in a different datatype, a stable state might correspond to a state where a thread is blocked on some other event, e.g. waiting to obtain a lock. An assertion like the one above would signal this as an error, since that thread is refusing the expected end event. And we should probably consider this as an error: it represents a partial deadlock; since the state is stable, no other thread is about to release the lock to unblock the blocked thread.

Now consider a check using explicit linearization points, such as our final check Spec4 $\sqsubseteq_F$ System4.

Consider a state where a pending operation has performed its linearization point. Then the refinement check requires that in each subsequent stable state, the end event must not be refused. This seems like a reasonable progress property: after linearization, an operation should be able to return regardless of what other threads do.

Now consider a state where a pending operation has *not* performed its linearization point. The refinement check then requires that a corresponding linearization point is available. This is true for the lock-free queue, and will be true of other lock-free datatypes: lock freedom implies that each thread should be able to proceed to its linearization point.

However, this will normally not be true of a datatype that uses locking: in most such datatypes, each linearization point holds while the thread is holding the lock. Suppose two threads t_1 and t_2 have pending linearization points, and that t_1 holds the lock; then t_2's linearization point will be refused until after t_1's linearization point.

A suitable progress property in this case would be that if there are pending linearization points, then they cannot all be refused. We capture this property via an extra component LPReg of the specification, which tracks the set of threads with a pending linearization point. We first illustrate the technique by adapting Spec3 (where linearization points of empty dequeues are hidden). For convenience, we define LPReg by a renaming of a simpler process LPReg0, which uses an event begin.t for an arbitrary begin event of thread t, and event LP.t for an arbitrary linearization point of t.

```
channel begin, LP: ThreadID
LPReg0(pending) =
  (begin?t → LPReg0(pending∪{t}) ⊓ STOP)
  □ pending ≠ {} & LP$t:pending → LPReg0(pending−{t}))
LPReg =
  LPReg0({})
    ⟦ begin.t ← beginEnqueue.t.value, begin.t ← beginDequeue.t |
         t ← ThreadID, value ← T ⟧
    ⟦ LP.t ← enqueue.t.value, LP.t ← enqueueFull.t, LP.t ← dequeue.t.value |
         t ← ThreadID, value ← T ⟧
Spec3' =
  let sync = {| beginEnqueue, beginDequeue, enqueue, enqueueFull, dequeue |}
  within (Spec0 [| sync |] LPReg)
              ⟦ dequeue.t.value ← deqLP.t | t ← ThreadID, value ← T ⟧
              \ {| dequeueEmpty |}
assert Spec3' ⊑F System3
```

Capturing this progress condition when using an explicit linearization point for empty dequeues is a little harder. Recall that in Spec4 an event dequeueEmpty.t might or might not represent a true linearization point. We arrange for LPReg to guess which is the case. However, recall that the linearizer used in Spec4 made a similar guess. If these two components make their guesses in an incompatible way, this would allow such a thread to deadlock, which certainly isn't what we want. Instead we:

- Replace the nondeterministic choice in the linearizer by an external choice, so the linearizer follows the guess made by LPReg;
- Have LPReg insist that endDequeueEmpty events are available when it has guessed that a corresponding dequeueEmpty event was a true linearization point.

The corresponding definitions are in Figure 2.

Results The table below gives experimental results for a system containing six nodes, three threads, and two data vales. For simplicity, the files did not include aspects of node recycling. The experiments were performed on a 32-core server (two 2.1GHz Intel(R) Xeon(R) Gold 6130 CPUs with hyperthreading enabled, with 384GB of RAM).

```
LinDeq(me) =
  dequeueEmpty.me → (endDequeueEmpty.me → Linearizer(me) □ LinDeq(me))
  □ dequeue.me?value → endDequeue.me.value → Linearizer(me)
LPReg0(pending, pendingRet) =
  (begin?t → LPReg0(pending∪{t}, pendingRet) ⊓ STOP)
  □
  pending ≠ {} & (
    LP$t:pending → LPReg0(pending−{t}, pendingRet)
    ⊓ dequeueEmpty$t:pending →
          (LPReg0(pending, pendingRet) ⊓ LPReg0(pending−{t}, pendingRet∪{t}))
  )
  □
  endDequeueEmpty?t:pendingRet → LPReg0(pending, pendingRet − {t})
LPReg =
  LPReg0({}, {})
    ⟦ begin.t ← beginEnqueue.t.value, begin.t ← beginDequeue.t |
        t ← ThreadID, value ← T ⟧
    ⟦ LP.t ← enqueue.t.value, LP.t ← enqueueFull.t, LP.t ← dequeue.t.value |
        t ← ThreadID, value ← T ⟧
Spec4' =
  let sync = {| beginEnqueue, beginDequeue, endDequeueEmpty,
                enqueue, enqueueFull, dequeueEmpty, dequeue |}
  within (Spec0 [| sync |] LPReg)
             ⟦ dequeue.t.value ← deqLP.t | t ← ThreadID, value ← T ⟧
assert Spec4' ⊑_F System4
```

Figure 2: Capturing progress with explicit linearization points.

Refinement tests	States	Transitions	Compilation	Checking
Spec $\sqsubseteq_F$ System	22.6B	71.7B	433s	4101s
Spec1 $\sqsubseteq_F$ System1	20.2B	63.9B	415s	3947s
Spec2 $\sqsubseteq_F$ System2	10.5B	33.0B	8s	1524s
Spec3 $\sqsubseteq_F$ System3	7.1B	22.5B	5s	1037s
Spec4 $\sqsubseteq_F$ System4	7.4B	23.6B	6s	1054s

Note that the cases that make most difference are for successful enqueues and dequeues, i.e. the linearization points that change the state of the abstract datatype. This should not be surprising: when such abstract linearization points do not correspond to concrete datatypes, the specification can be in a state corresponding to a nondeterministic choice over which operations have been linearized, which increases the state space, and also the time to normalise the specification.

Note that making explicit the provisional linearization points of unsuccessful dequeues made the check slightly larger in this case: the linearizers have more states, and this outweighs other advantages. However, this technique might prove more useful in other cases.

On failed refinement checks Suppose we perform a refinement check as above, but that it fails. This doesn't immediately tell us that the concurrent datatype is not linearizable: it only tells us that it is not linearizable *with respect to the claimed linearization points.*

However, it will often be the case that the counterexample returned by the model checker, restricted to the begin and end events, gives a concurrent history that can be seen not to be linearizable: i.e. there is no way of correctly inserting abstract linearization points into the history. In this case (assuming the model is correct) we have found an error on the concrete datatype.

Where this is not the case, examining the counterexample may reveal better concrete linearization points, or, at least, that the claimed linearization points were incorrect. But, if all else fails, it might be necessary to resort to the technique of the earlier paper.