Analysing Lock-Free Linearizable Datatypes using CSP

Gavin Lowe

Department of Computer Science, University of Oxford, UK.
`gavin.lowe@cs.ox.ac.uk`

Abstract. We consider how we can use the process algebra CSP and the model checker FDR in order to obtain assurance about the correctness of concurrent datatypes. In particular, we perform a formal analysis of a concurrent queue based on a linked list of nodes. We model the queue in CSP and analyse it using FDR. We capture two important properties using CSP, namely linearizability and lock-freedom.

1 Introduction

Many concurrent programs are designed so that threads interact only via a small number of concurrent datatypes. Code outside of these concurrent datatypes can be written in pretty-much the same way as the corresponding sequential code; only the code of the datatypes themselves needs to be written in a way that takes concurrency into account.

Modern concurrent datatypes are often designed to be *lock-free*. No locks (or lock-like mechanisms) are used. This means that threads should not be indefinitely blocked by other threads, even if a thread is permanently de-scheduled. Many clever lock-free datatypes have been designed, e.g. [16, 24, 25, 9]. However, these datatypes tend to be complex, and less obviously correct than traditional lock-based datatypes. Clearly we need techniques for gaining greater assurance in their correctness.

In this paper we use CSP [20] and the model checker FDR [7] to analyse the lock-free queue of [16]. The queue is based on a linked list of nodes. More precisely, we analyse the version of the queue given in [9], which simplifies the presentation by assuming the presence of a garbage collector (although our CSP model will include that garbage collection); by contrast, the version in [16] performs explicit de-allocation of nodes by threads.

CSP has been used to analyse concurrent programs on a number of previous occasions, e.g. [23, 28, 13]. However, to the best of our knowledge, this is the first model of a dynamic data structure. Further, we include a mechanism —akin to garbage collection— that reclaims nodes when they become free, allowing them to be re-used. This extends the range of behaviours that our model captures to include behaviours with an unbounded number of `enqueue` and `dequeue` operations (on a bounded-length queue).

A concurrent datatype is said to be *linearizable* [10] if:

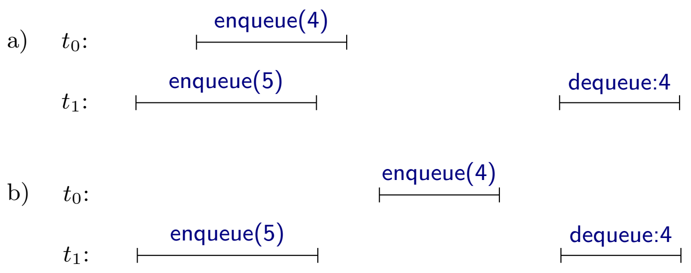

Fig. 1. Two timelines of executions: a) a linearizable execution; b) an unlinearizable execution. Time runs from left to right; each horizontal line indicates the duration of a method call, labelled with the name of the method and (for a dequeue) the value returned; the identities of threads are at the left.

- Each method call appears to take place atomically: different method calls do not interfere with one another. The order in which method calls appear to take place is consistent with a sequential execution.
- Each call appears to take place at some point between the call's invocation and response; this point is referred to as the *linearization point*. Put another way, if one call ends before another begins, then the former should appear to take place before the latter.

For example, consider Figure 1, which displays timelines of two possible executions of a concurrent queue. In execution (a), threads t_0 and t_1 perform concurrent enqueues of 4 and 5, and then t_1 performs a dequeue, obtaining 4. This execution is linearizable. The two enqueues appear to have taken place atomically, with the 4 being enqueued before the 5. Alternatively, if the dequeue had obtained 5, the execution would still have been linearizable. The second clause, above, allows the two enqueues to appear to take place in either order.

By contrast, execution (b) is not linearizable. Here the enqueue of 5 finishes before the enqueue of 4. Hence we would expect that the subsequent dequeue would obtain 5; the second clause above enforces this.

In this paper we use FDR to show that our model of the lock-free queue is indeed a linearizable queue. We believe that CSP allows such linearizable specifications to be captured in a very elegant way. Our method of capturing the specification is modular: it can easily be adapted to other linearizable specifications. Our method does not require the user to identify the linearization points.

Linearizability is a safety property: it specifies that no method call returns a "bad" result; however, it does not guarantee that the system makes progress. The property of *lock freedom* guarantees progress. Formally, a concurrent datatype is lock-free if, in every state, it guarantees that some method call finishes in a finite number of steps [9]. Unsurprisingly, a datatype that uses locks in a meaningful way is not lock-free: if one thread acquires a lock, but is permanently de-scheduled, other threads may perform an unbounded number of steps trying to obtain the lock. But further, a datatype may fail to be lock-free even if it doesn't use locks, for example if threads repeatedly interfere with one another and so have to re-try.

We show that our model of the lock-free queue is indeed lock-free. We capture the property in CSP using a combination of deadlock freedom and divergence freedom.

Finally, we also verify that the model is free from null-pointer references, and dangling references (where deallocated objects are accessed).

FDR has recently been extended with a form of symmetry reduction [8]. We use this symmetry reduction in our analysis. The model will contain a large amount of symmetry: it will be symmetric in the type of identities of nodes, the type of identities of threads, and the type of data, in the sense that applying any permutation over one of these types to any state of the model will produce another state of the model. The symmetry reduction factors the model by this symmetry. We show that this gives a significant speed up in checking time, and hence an increase in the size of system that we can analyse.

One additional benefit of our approach is that it is easy to adapt the model in order to consider variants in the design of the datatype. We briefly illustrate how such changes in the model can explain a couple of non-obvious aspects of the datatype.

To summarise, our main contributions are as follows:

– A technique for modelling and analysing concurrent datatypes;
– A generic modular technique for capturing linearizability;
– A straightforward technique for capturing lock freedom;
– A technique for modelling reference-linked data structures, including a mechanism for recycling of nodes;
– An investigation into the performance improvements provided by symmetry reduction;
– An instructive case study in the application of CSP-based verification.

The rest of the paper is structured as follows. We present the lock-free queue datatype in Section 2. We present the CSP model in Section 3, and describe our analysis using FDR in Section 4. We sum up and discuss prospects for this line of work in Section 5.

1.1 Related work

A number of other papers have considered the verification of linearizability, using either model checking or other verification techniques. However, we are not aware of other examples of the verification of lock freedom.

Vechev et al. [27] study linearizabilty using the SPIN model checker, using two different approaches. One approach uses bounded-length runs, at the end of which it is checked whether the run was linearizable, by considering all relevant re-orderings of the operations. The other approach requires linearization points to be identified by the user. Like us, they model a garbage collector. The approach suffers from state-space explosion issues: for a concurrent set based on a linked list, applicability is limited to two threads and two keys, even when linearization points are provided.

Liu et al. [12] also study linearizability in the context of refinement checking, in their case, using the model checker PAT. They capture linearizability but not liveness properties. They describe experiments using symmetry reduction and partial order reduction to improve the efficiency of the search. By way of comparison, we (with Tom Gibson-Robinson) have built a CSP model corresponding to one of their examples, in a similar style to the model of this paper; our experiments suggest that FDR is several hundred times faster on these models (on similar architectures).

Burckhardt et al. [2] analyse for linearizability as follows. They randomly pick a small number of threads to perform a small number of operations, typically three threads each performing three operations. They then use the CHESS model checker to generate all behaviours caused by interleaving these operations, and test whether each corresponds to a sequential execution of the same operations. They uncover a large number of bugs within the .NET Framework 4.0.

Černý et al. [3] show that linearizability is decidable for a class of linked-list programs that invoke a fixed number of operations in parallel. Their restrictions exclude the example of this paper: they assume a *fixed* head node, no tail reference, and that threads traverse the list monotonically; however, it is not clear how essential these restrictions are. Their approach shows that a program is a linearizable version of its own sequential form, rather than a linearizable version of a more abstract specification, such as a queue. In practice, their approach is limited to a pair of operations in parallel, because of the state space explosion.

Vafeiadis [26] uses abstract interpretation to verify linearizability, by considering candidate linearization points. The technique works well on some examples, but does not always succeed, and works less well on examples with more complex abstractions. Colvin et al. [5] and Derrick et al. [6] prove linearizability by verifying a simulation against a suitable specification, supported by a theorem prover. These approaches give stronger guarantees than our own, but require much more effort on the part of the verifier.

1.2 CSP

In this section we give a brief overview of the syntax for the fragment of CSP that we will be using in this paper. We then review the relevant aspects of CSP semantics, and the use of the model checker FDR in verification. For more details, see [20].

CSP is a process algebra for describing programs or *processes* that interact with their environment by communication. Processes communicate via atomic *events*. Events often involve passing values over channels; for example, the event $\mathsf{c.3}$ represents the value $\mathsf{3}$ being passed on channel c. Channels may be declared using the keyword **channel**; for example, **channel** $\mathsf{c} : \mathsf{Int}$ declares c to be a channel that passes an Int. The notation $\{|\mathsf{c}|\}$ represents the set of events over channel c.

The simplest process is STOP, which represents a deadlocked process that cannot communicate with its environment. By contrast, $\mathbf{div}$ is a divergent process that performs an unbounded number of internal τ events.

The process $\mathsf{a} \rightarrow \mathsf{P}$ offers its environment the event a; if the event is performed, the process then acts like P. The process $\mathsf{c?x} \rightarrow \mathsf{P}$ is initially willing to input a value x on channel c, i.e. it is willing to perform any event of the form $\mathsf{c.x}$; it then acts like P (which may use x). Similarly, the process $\mathsf{c?x{:}X} \rightarrow \mathsf{P}$ is willing to input any value x from set X on channel c, and then act like P. Within input constructs, we use "_" as a wildcard: $\mathsf{c?\_}$ indicates an input of an arbitrary value. The process $\mathsf{c!v} \rightarrow \mathsf{P}$ outputs value v on channel c. Inputs and outputs may be mixed within the same communication, for example $\mathsf{c?x!v} \rightarrow \mathsf{P}$.

The process $\mathsf{P} \mathbin{\Box} \mathsf{Q}$ can act like either P or Q, the choice being made by the environment: the environment is offered the choice between the initial events of P and Q. By contrast, $\mathsf{P} \sqcap \mathsf{Q}$ may act like either P or Q, with the choice being made internally, not under the control of the environment. $\Box\ \mathsf{x{:}X} \bullet \mathsf{P(x)}$ is an indexed external choice, with the choice being made over the processes $\mathsf{P(x)}$ for x in X. The process $\mathbf{if}\ \mathsf{b}\ \mathbf{then}\ \mathsf{P}\ \mathbf{else}\ \mathsf{Q}$ represents a conditional. The process $\mathsf{b}\ \&\ \mathsf{P}$ is a guarded process, that makes P available only if b is true; it is equivalent to $\mathbf{if}\ \mathsf{b}\ \mathbf{then}\ \mathsf{P}\ \mathbf{else}\ \mathsf{STOP}$.

The process $\mathsf{P} \ [|\ \mathsf{A}\ |]\ \mathsf{Q}$ runs P and Q in parallel, synchronising on events from A. The process $\parallel\ \mathsf{x{:}X} \bullet [\mathsf{A(x)}]\ \mathsf{P(x)}$ represents an indexed parallel composition, where, for each x in X, $\mathsf{P(x)}$ is given alphabet $\mathsf{A(x)}$; processes synchronize on events in the intersection of their alphabets. The process $\mathsf{P}\ |||\ \mathsf{Q}$ interleaves P and Q, i.e. runs them in parallel with no synchronisation. $|||\ \mathsf{x{:}X} \bullet \mathsf{P(x)}$ represents an indexed interleaving.

The process $\mathsf{P} \setminus \mathsf{A}$ acts like P, except the events from A are hidden, i.e. turned into internal τ events.

A *trace* of a process is a sequence of (visible) events that a process can perform. We say that P is refined by Q in the traces model, written $\mathsf{P} \sqsubseteq_T \mathsf{Q}$, if every trace of Q is also a trace of P. FDR can test such refinements automatically, for finite-state processes. Typically, P is a specification process, describing what traces are acceptable; this test checks whether Q has only such acceptable traces.

Traces refinement tests can only ensure that no "bad" traces can occur: they cannot ensure that anything "good" actually happens; for this we need the stable failures or failures-divergences models. A *stable failure* of a process P is a pair (tr, X), which represents that P can perform the trace tr to reach a stable state (i.e. where no internal events are possible) where X can be refused, i.e., where none of the events of X is available. We say that P is refined by Q in the stable failures model, written $\mathsf{P} \sqsubseteq_F \mathsf{Q}$, if every trace of Q is also a trace of P, and every stable failure of Q is also a stable failure of P.

We say that a process *diverges* if it can perform an infinite number of internal (hidden) events without any intervening visible events. If $\mathsf{P} \sqsubseteq_F \mathsf{Q}$ and Q is divergence-free, then if P can stably offer an event a, then so can Q; hence such tests can be used to ensure Q makes useful progress.

2 The lock-free queue

In this section we present the lock-free queue. Our presentation is based on that from [9]. The code, in Scala, is in Figure 2.

The lock-free queue uses *atomic references*[1]. An atomic reference encapsulates a standard reference, say to an object of class A, and provides get and set operations. In addition, it provides an atomic compare-and-set (CAS) operation, which can be thought of as an atomic implementation of the following:

```
def compareAndSet(expected: A, update: A) : Boolean = {
  if (expected == current){ current = update; true} else false
}
```

A thread passes in two values: expected, which the thread believes the atomic reference holds; and update, the value the thread wants to update it to. If the current value is indeed as expected, it is updated. The thread receives an indication as to whether the operation was successful. This operation can be used to allow concurrent threads to interact safely in a lock-free way.

The lock-free queue is built as a linked list of Nodes: each node holds a value field, of (polymorphic) type T; nodes are linked together using next fields which are atomic references to the following node.

The lock-free queue employs two shared variables, both atomic references (lines 9–10 of Figure 2): head is a reference to a dummy header node; and tail is normally a reference to the last node in the linked list, but will temporarily refer to the penultimate node when an item has been partially enqueued.

The main idea of the algorithm, which gives it its lock-free property, is as follows: if one thread partially enqueues an item, but is de-scheduled and leaves the queue in an inconsistent state with tail not referring to the final node, then other threads try to tidy up by advancing tail; one will succeed, and so progress is made, and eventually a thread will complete its operation.

The enqueue operation starts by creating a node to store the new value[2]. It then reads tail and the following node into local variables myTail and myNext. As an optimization, it re-reads tail, in case it has changed, retrying if it has. Otherwise, in the normal case that myNext is **null**, it attempts a CAS operation on myTail.next (line 19), to set it to the new node. If this succeeds, the value is correctly enqueued. It then attempts to advance the tail reference to the new node, and returns regardless of whether this is successful: if the CAS fails, some other thread has already advanced tail. If myNext is not **null** (line 23), it means that a value has been partially enqueued, with tail not advanced to the last node: it attempts to so-advance tail, and retries.

The dequeue operation starts by reading head, tail and the node after head into local variables myHead, myTail, and myNext. It then re-reads head, in case it has changed, retrying if it has. Otherwise, if the head and tail are equal, and

[1] `http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicReference.html`.

[2] The Scala code ignores the possibility of new nodes not being available, but we will need to consider this possibility in our CSP models.

```
class LockFreeQueue[T]{
  // We build linked lists from Nodes of the following type
  class Node(val value: T, val next: AtomicReference[Node])

  // initial dummy header
  private val firstNode =
    new Node(null.asInstanceOf[T], new AtomicReference[Node](null))
  // Atomic references to head and tail nodes
  private val head = new AtomicReference(firstNode)
  private val tail = new AtomicReference(firstNode)

  /** Add value to the queue */
  def enqueue(value: T) : Unit = {
    val node = new Node(value, new AtomicReference[Node](null))
    while(true){
      val myTail = tail.get; val myNext = myTail.next.get
      if (myTail == tail.get) // in case it has been changed (optimization)
        if (myNext == null){
          if (myTail.next.compareAndSet(null, node)){
            tail.compareAndSet(myTail, node); return
          } // else re-try
        }
        else // myNext != null, try to advance tail
          tail.compareAndSet(myTail, myNext) // and retry
      // else retry
  } }

  /** Dequeue and return a value if the queue is non-empty; else return null */
  def dequeue : T = {
    while(true){
      val myHead = head.get; val myTail = tail.get; val myNext = myHead.next.get
      if (myHead == head.get) // in case it has been changed (optimization)
        if (myHead == myTail){
          if (myNext == null) return null // empty queue, return null
          else // new item partially enqueued
            tail.compareAndSet(myTail, myNext) // try to advance tail; retry
        }
        else{ // non-empty queue; try to remove node from queue
          if (head.compareAndSet(myHead, myNext)) return myNext.value
          // else myNext.value already taken; retry
        }
      // else Head changed; retry
} } }
```

Fig. 2. The lock-free queue in Scala.

```
datatype NodeIDType = Null | N0 | N1 | N2  -- node identities
NodeID = diff(NodeIDType, {Null})          -- real nodes
datatype T = A | B                         -- data values
datatype ThreadID = T0 | T1                -- thread identities
```

Fig. 3. The basic types of the model.

myNext is **null** (line 34), the queue is empty, and the operation returns the special value **null** to indicate this. Alternatively, if head and tail are equal but myNext is non-**null** (line 35), there is a partially enqueued item, so it tries to advance tail and retries. Otherwise, the queue is non-empty (line 38). It tries to advance the head to myNext using a CAS operation; if the CAS succeeds, it returns the value in myNext; otherwise, another thread has taken the value in myNext, so the operation retries.

Note that an individual enqueue or dequeue operation is not guaranteed to terminate: it may repeatedly find partially enqueued items, and so repeatedly retry. However, if this happens infinitely often then infinitely many other operations will terminate, so the data structure is still lock-free.

3 The CSP model

In this section we present our CSP model of the lock-free queue.[3] In Section 3.1 we model the nodes of the list, and the atomic references head and tail. In Section 3.2 we model the program threads that perform the enqueueing and dequeueing operations. In Section 3.3 we describe the technique for re-cycling nodes, which identifies nodes that have been removed from the linked list and that have no relevant references to them, and frees them up, making them available for reuse. We put the system together in Section 3.4.

Our model is parameterized by three types (see Figure 3): the type NodeIDType of node identities; the type T of data held in the queue; and the type ThreadID of thread identities. The type NodeIDType contains a distinguished value Null, which models the **null** reference; we write NodeID for the set of "proper" nodes. We will consider larger values for these types in Section 4.5. The models will be symmetric in the types NodeID, T and ThreadID.

3.1 Nodes, Head, Tail, and the constructor

The CSP model of the nodes of the linked list is presented in Figure 4. We declare channels corresponding to actions by threads upon nodes. The event initNode.t.n.v represents thread t initialising node n to hold v; getValue.t.n.v represents t reading value v from n; getNext.t.n.n1 represents t obtaining the value n1

[3] The CSP script is available from `http://www.cs.ox.ac.uk/people/gavin.lowe/LockFreeQueue/LockFreeQueueLin.csp`.

```
-- Channels used by nodes.
channel initNode : ThreadID . NodeIDType . T
channel getValue : ThreadID . NodeIDType . T
channel getNext : ThreadID . NodeIDType . NodeIDType
channel CASnext : ThreadID . NodeIDType . NodeIDType . NodeIDType . Bool
channel removeNode : ThreadID . NodeIDType
channel free : NodeID
channel noFreeNode : ThreadID

-- A node process, with identity me, currently free.
FreeNode :: (NodeIDType) → Proc
FreeNode(me) =
  initNode?_!me?value → Node(me, value, Null, false)
  □ □ e : diff (alphaNode(me), {|initNode,free,noFreeNode|}) • e → div

-- A node process, identity me, holding datum value and next pointer next;
-- removed indicates whether the node has been removed from the list.
Node :: (NodeIDType, T, NodeIDType, Bool) → Proc
Node(me, value, next, removed) =
  getValue?_!me.value → Node(me, value, next, removed)
  □ getNext?_!me.next → Node(me, value, next, removed)
  □ CASnext?_!me?expected?new!(expected==next) →
       Node(me, value, if expected==next then new else next, removed)
  □ not(removed) & removeNode?_!me → Node(me, value, next, true)
  □ removed & free.me → FreeNode(me)
  □ noFreeNode?_ → Node(me, value, next, removed)

-- Alphabet of node me.
alphaNode(me) = {| free.me, initNode.t.me, removeNode.t.me, noFreeNode.t,
                   getValue.t.me, getNext.t.me, CASnext.t.me | t ← ThreadID |}
-- All nodes
AllNodes = || id : NodeID • [alphaNode(id)] FreeNode(id)
```

Fig. 4. Model of the nodes.

of n's next field; CASnext.t.n.expected.new.res represents t performing a CAS on n's next field, trying to change it from expected to new, with res giving the boolean result; removeNode.t.n represents t marking n as removed from the linked list; free.n represents n being recycled; and noFreeNode.t represents t failing to obtain a new node.

A free node (process FreeNode) can be initialised with a particular value and with its next field Null. Our model also allows the node to perform various events corresponding to a thread incorrectly accessing this node, after which it diverges;

```
datatype AtomicRefID = Head | Tail -- The IDs of atomic references
-- Channels used by Head and Tail
channel getNode : ThreadID . AtomicRefID . NodeIDType
channel CAS : ThreadID . AtomicRefID . NodeIDType . NodeIDType . Bool
channel initAR : NodeID

-- An atomic reference to node
AtomicRefNode :: (AtomicRefID, NodeIDType) → Proc
AtomicRefNode(me, node) =
  getNode?t!me.node → AtomicRefNode(me, node)
  □ CAS?t!me?expected?new!(expected=node) →
       AtomicRefNode(me, if expected=node then new else node)

-- The atomic reference variables
HeadAR = initAR?h → AtomicRefNode(Head, h)
TailAR = initAR?h → AtomicRefNode(Tail, h)

AllARs = HeadAR [| {|initAR|} |] TailAR

-- The constructor
Constructor = initAR?h → initNode?_!h?_ → RUN({|beginEnqueue, beginDequeue|})
```

Fig. 5. Model of the head and tail atomic references, and the constructor.

later we verify that the system cannot diverge, and so verify that no such event can occur.

An initialised node can: (1) have its value field read by a thread; (2) have its next field read by a thread; (3) have a CAS operation performed on its next field by a thread: if the expected field matches the current value of next, the value is updated to the new field and the result is true; otherwise next is unchanged and the result is false; (4) be marked as removed from the linked list (if not so already); (5) be freed up, if already marked as removed from the linked list; (6) signal that no free node is available (all nodes will synchronize on this event, so it will be available only if all nodes are in this state).

We combine the nodes in parallel with the natural alphabets.

Figure 5 gives the CSP model of the atomic reference variables, head and tail, together with a "constructor" process Constructor that initialises these variables and the dummy header node. The type AtomicRefID gives identities of these atomic references. Event getNode.t.ar.n represents thread t reading the value n of atomic reference ar. Event CAS.t.ar.expected.new.res represents t performing a CAS operation on ar, trying to change it from expected to new, with res giving the boolean result. Event initAR.h represents the two atomic references being initialised to refer to initial dummy header node h.

```
-- events to signal the start or end of operations
channel beginEnqueue, endEnqueue, endEnqueueFull, endDequeue : ThreadID . T
channel beginDequeue, endDequeueEmpty : ThreadID
channel releaseRefs : ThreadID -- a thread releases its references

-- A thread, which enqueues or dequeues.
Thread(me) =
  beginEnqueue.me?value → Enqueue(me, value)
  □ beginDequeue.me → Dequeue(me)
```

Fig. 6. Model of a thread.

The model of an atomic reference is similar in style to the model of a node, but simpler. The constructor chooses an initial dummy header node h, initialises the two atomic references to refer to it, initialises h to hold a nondeterministic initial value, and then allow begin events to occur; the effect of the last step is to block other threads until the construction is complete.

3.2 Enqueueing and dequeueing threads

Figures 6 and 7 give the models of the threads. In order to later capture the requirements, we include additional events to signal the start or end of an enqueue or dequeue operation, including the end of a dequeue operation that failed because the queue was empty, or an enqueue operation that failed because the queue was full (i.e. there was no free node). We also include events on channel releaseRefs to represent a thread releasing its references (before re-trying).

Each Thread process represents a thread that repeatedly performs enqueue or dequeue operations. The process Enqueue(me, value) represents a thread with identity me trying to enqueue value. It starts by trying to initialise a node to hold value; if this fails, as indicated by the noFreeNode event, it signals that the queue is full. The process Enqueue' corresponds to the **while** loop in the enqueue function of Figure 2. Most of the definition is a direct translation of the Scala code from that figure: the reader is encouraged to compare the two. If the enqueue succeeds, this is signalled with an endEnqueue event. If the enqueue fails, and the thread has to retry, it releases the references it held, so these can potentially be recycled. The dequeue operation is modelled in a very similar way.

3.3 Recycling nodes

We now describe our mechanism for recycling nodes in the model. While the mechanism is very similar to memory management techniques in implementations, the intention is different: our aim is to increase the coverage of our model,

```
-- An enqueueing thread
Enqueue :: (ThreadID, T) → Proc
Enqueue(me, value) =
  initNode . me ? node ! value → Enqueue'(me, value, node)
  □ noFreeNode . me → endEnqueueFull . me . value → Thread(me)

Enqueue' :: (ThreadID, T, NodeIDType) → Proc
Enqueue'(me, value, node) =
  getNode . me . Tail ? myTail → getNext . me . myTail ? myNext →
  getNode . me . Tail ? myTail' →
  if myTail=myTail' then -- in case it's been changed (optimization)
    if myNext=Null then
      CASnext . me . myTail . Null . node ? result →
      if result then -- enqueue succeeded, so advance tail
        CAS . me . Tail . myTail . node ? _ → endEnqueue . me . value → Thread(me)
      else -- CASnext failed; retry
        releaseRefs . me → Enqueue'(me, value, node)
    else -- myNext≠Null, try to advance tail
      CAS . me . Tail . myTail . myNext ? _ →
      releaseRefs . me → Enqueue'(me, value, node)
  else -- Tail changed; retry
    releaseRefs . me → Enqueue'(me, value, node)

-- A dequeuing thread
Dequeue :: (ThreadID) → Proc
Dequeue(me) =
  getNode . me . Head ? myHead → getNode . me . Tail ? myTail →
  getNext . me . myHead ? myNext → getNode . me . Head ? myHead' →
  if myHead=myHead' then -- in case it's been changed (optimization)
    if myHead=myTail then
      if myNext=Null then endDequeueEmpty . me → Thread(me) -- empty queue
      else -- new item partially enqueued
        CAS . me . Tail . myTail . myNext ? _ → -- try to advance tail; retry
        releaseRefs . me → Dequeue(me)
    else -- non-empty queue; try to remove node from queue
      CAS . me . Head . myHead . myNext ? result →
      if result then
        getValue . me . myNext ? value → removeNode . me . myHead →
        endDequeue . me . value → Thread(me)
      else -- myNext . value already taken; retry
        releaseRefs . me → Dequeue(me)
  else releaseRefs . me → Dequeue(me) -- Head changed; retry
```

Fig. 7. Model of a thread (continued).

```
HPs :: (ThreadID, NodeIDType, NodeIDType, NodeIDType, Bool) → Proc
HPs(me, h, t, n, enq) =
  beginEnqueue . me ? _ → HPs(me, h, t, n, true)
  □ beginDequeue . me → HPs(me, h, t, n, false)
  □ getNode . me . Tail ? t' → HPs(me, h, if enq then t' else Null, n, enq)
  □ getNode . me . Head ? h' → HPs(me, if enq then Null else h', t, n, enq)
  □ getNext . me ? _:NodeID ? n' → HPs(me, h, t, if enq then Null else n', enq)
  □ ( □ e : releaseEvents (me) • e →
       HPs(me, Null, Null, Null, if e = ( releaseRefs . me) then enq else false ))
  □ free ? _: diff (NodeID, {t, h, n}) → HPs(me, h, t, n, enq)

-- The events on which me releases all its hazard pointers.
releaseEvents (me) = {|endEnqueue . me, endEnqueueFull . me, endDequeue . me,
                      endDequeueEmpty . me, releaseRefs . me|}

-- All hazard pointer processes, synchronizing on free events.
alphaHP(me) = union( releaseEvents(me),
    {| beginEnqueue . me, beginDequeue . me, getNode . me, getNext . me, free |})
HazardPointers = || me ← ThreadID • [alphaHP(me)] HPs(me,Null,Null,Null,false)
```

Fig. 8. The hazard pointers.

capturing executions with an arbitrary number of enqueue and dequeue operations.

We use a technique inspired by *hazard pointers* [17]. The idea (as an implementation technique) is that each thread has a few pointer variables, known as hazard pointers: no node referenced by such a pointer should be recycled. When a thread removes a node from a data structure, it can add the node to a list of removed nodes. The thread intermittently reads the hazard pointers of all threads, and recycles any removed node that is not referenced by a hazard pointer.

In the lock-free list, the hazard pointers should be each thread's myTail during an enqueue operation, and its myHead and myNext during a dequeue operation. It is obvious that it would be hazardous to recycle any of these nodes, since fields of each are accessed by the thread. Our subsequent analysis shows that these are sufficient hazard pointers.

Figure 8 gives the relevant part of the CSP model. The process HPs(me, h, t, n, enq) records the hazard pointers of thread me; the parameters h, t and n store the thread's myHead, myTail and myNext variables, where relevant; the parameter enq records whether the thread is enqueueing; these are updated by synchronizing with the relevant events of the thread. The hazard pointer parameters are reset to Null when the thread releases the references. This process allows any node other than its hazard pointers to be freed. All HPs

```
-- All threads
AllThreads0 = ||| id : ThreadID • Thread(id)
AllThreads = AllThreads0 [| {|beginEnqueue, beginDequeue|} |] Constructor

-- synchronisation set between Threads and HazardPointers.
HPSyncSet = union( {| beginEnqueue, beginDequeue, getNode, getNext |},
                   Union({ releaseEvents(t) | t <- ThreadID }) )
-- synchronisation set between Threads/HazardPointers and Nodes/AtomicRefs
syncSet = union( {| initNode.t.n, getValue.t.n, getNext.t.n, CASnext.t.n |
                    t <- ThreadID, n <- NodeID |},
                 {| getNode, CAS, free, removeNode, noFreeNode, initAR |} )
-- Put components together in parallel
System0 = (AllThreads [| HPSyncSet |] HazardPointers)
            [| syncSet |] (AllNodes ||| AllARs)

-- Prioritise releaseRefs, free and removeNode over all other events.
PriEvents = {|releaseRefs, free, removeNode|}
System1 = prioritise (System0, <PriEvents, diff (Events, PriEvents)>)
System = System1 \ union(syncSet, {|releaseRefs|})
```

Fig. 9. The complete system.

processes synchronize on the free events, so a node can be freed when it is not referenced by *any* hazard pointer.

3.4 The complete system

We combine the system together in parallel in Figure 9.

We prioritise releaseRefs, free and removeNode events over all other events, for two reasons. Firstly, we want to ensure that nodes are recycled as soon as possible, so a thread does not fail to obtain a new node when there is one waiting to be recycled. Secondly, this acts as a form of partial-order reduction, and markedly reduces the size of the state space. This prioritisation is sound since forcing these events to occur does not disable any of the standard events on nodes (on channels getValue, getNext and CASnext).

We then hide other events: in the resulting process System, the only visible events are the begin and end events.

4 Analysis

We now describe our FDR analysis of the model. In Section 4.1 we show that the datatype is a linearizable queue. In Section 4.2 we show that the queue is lock-free, and also that no thread attempts to access a freed node (i.e. there are no dangling pointers). In Section 4.3 we show that no node attempts to de-reference a null reference. In Section 4.4 we discuss the use of symmetry reduction in the checks. In Section 4.5 we discuss our results. Finally, in Section 4.6 we discuss how the model can be adapted to alternative designs, and so understand why some details of the datatype are as they are.

4.1 A linearizable queue

Recall that a datatype is linearizable if each method call appears to take place atomically at some point between the call's invocation and response; these points are called linearization points. We prove that our model is a linearizable queue by building a suitable specification in two steps: first we build a specification of a queue, where the events correspond to the linearization points; and then we ensure that these events occur between the corresponding begin and end events. (Alur et al. [1] have proposed a similar technique.)

Our specification (Figure 10) introduces events to correspond to the linearization points. The process QueueSpec models a queue, based on these events, where the parameter q records the sequence of values currently in the queue. A dequeue attempt succeeds when the queue is non-empty; otherwise it signals that the queue is empty. An enqueue may succeed or fail, depending upon the queue's current length. An enqueue is guaranteed to succeed when $\#q+2*card(ThreadID)-1 < card(NodeID)$, since a free node will always be available in this case. When $\#q+1 < card(NodeID) \leq \#q+2*card(ThreadID)-1$, an enqueue may either succeed or fail, depending upon how many deleted nodes are still referenced by hazard pointers of other threads.

We then ensure that the events of QueueSpec occur between the corresponding begin and end events. The process Linearizer(me) does this for events of thread me. We combine the components together in parallel, hiding the events of QueueSpec. The resulting specification requires that each trace (of begin and end events) is linearizable: each operation appears to happen atomically at the point of the corresponding (hidden) linearization event; the results of these operations are consistent with an execution of a sequential queue (as enforced by the QueueSpec process); each linearization event occurs between the corresponding begin and end events (as enforced by the corresponding Linearizer process).

Note that we carry out the check in the stable failures model. On the assumption that System is divergence-free, this also ensures liveness properties: that the relevant events eventually become available (if not preempted by other events). We discharge the divergence-freedom assumption below.

Spec is nondeterministic. Each state that is reachable after a particular trace *tr* corresponds to a state of Queue that represents a possible linearization that is consistent with *tr*. FDR normalises the specification: each state of the normalised specification corresponds to the *set* of states that the original specification can reach after a particular trace, i.e. the set of linearizations that are consistent with the visible trace so far.

```
channel enqueue, dequeue : ThreadID . T
channel dequeueEmpty, enqueueFull : ThreadID

QueueSpec = Queue(<>)
Queue(q) =
  (if q ≠ <> then dequeue?t!head(q) → Queue(tail(q))
   else dequeueEmpty?t → Queue(q) )
  □
  if #q+2*card(ThreadID)−1 < card(NodeID) then enqueue?t?x → Queue(q^<x>)
  else if #q+1 < card(NodeID) then
    enqueue?t?x → Queue(q^<x>) ⊓ enqueueFull?t → Queue(q)
  else enqueueFull?t → Queue(q)

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

 AllLinearizers = ||| id: ThreadID • Linearizer(id)
specSyncSet = {| enqueue, dequeue, dequeueEmpty, enqueueFull |}
Spec = ( AllLinearizers [| specSyncSet |] QueueSpec) \ specSyncSet
assert Spec ⊑F System
```

Fig. 10. Testing for linearizability.

4.2 Lock-freedom and dangling pointer freedom

Recall that a concurrent datatype is said to be lock-free if it always guarantees that some method call finishes in a finite number of steps, even if some (but not all) threads are permanently desscheduled. A failure of lock freedom can occur in two ways:

– One or more threads perform an infinite sequence of events without an operation ever finishing;
– A thread reaches a state where it is unable to perform any event, so if all other threads are permanently descheduled, the system as a whole makes no progress.

The former type of failure of lock freedom is easy to capture: a violation of this property would involve an unbounded number of events without an end event, which would represent a divergence of SystemE, below.

```
-- System with only end events visible
SystemE = System \ {| beginEnqueue, beginDequeue|}
assert SystemE :[divergence free]
```

In order to capture the latter type of failure, we need to be able to model the permanent descheduling of threads. We do this via a process Scheduler that allows all but one thread to be descheduled: the descheduling of thread t is captured by the event dies.t; the regulator allows events of a thread only if it has not been descheduled. We then check that the resulting system is deadlock free.

```
-- Alphabet of thread t
alpha(t) =
  {| initNode.t, getValue.t, getNext.t, CASnext.t, getNode.t, removeNode.t,
     CAS.t, beginEnqueue.t, endEnqueue.t, endEnqueueFull.t, endDequeue.t,
     beginDequeue.t, endDequeueEmpty.t |}

channel dies: ThreadID -- A particular thread dies

-- A regulator for the lock freedom property.
Scheduler(alive) =
  (□ t: alive, e: alpha(t) • e → Scheduler(alive))
  □ card(alive) > 1 & dies?t:alive → Scheduler(diff(alive,{t}))

SchedulerSyncSet = Union({alpha(t) | t ← ThreadID})
SystemLF0 = System0 [| SchedulerSyncSet |] Scheduler(ThreadID)
SystemLF = prioritise(SystemLF0, <PriEvents , diff(Events,PriEvents)>)
                  \ union(syncSet, {|releaseRefs, beginEnqueue, beginDequeue|})
assert SystemLF :[deadlock free]
```

Recall that dereferencing a dangling pointer (i.e. referencing a node that has been freed) leads to a divergence. The above divergence-freedom check therefore also ensures freedom from such dangling pointer errors. This check also guarantees that System is divergence-free, giving the liveness properties mentioned at the end of the last subsection.

4.3 Null reference exceptions

Finally, we check that no thread ever tries to de-reference the Null reference: we hide all other events and check that no event can occur.

```
nullRefs =
  {|initNode.t.Null,getValue.t.Null,getNext.t.Null,CASnext.t.Null | t ← ThreadID|}
assert STOP ⊑_T System0 \ diff(Events,nullRefs)
```

Parameters	Check	No sym. red.		Sym. red.	
		#states	time	#states	time
2, 2, 3	lin. queue	207K	0.5s	9.3K	0.5s
2, 2, 3	divergences	150K	0.9s	6.7K	0.4s
2, 2, 3	lock freedom	406K	0.5s	18K	0.4s
2, 2, 3	null refs	150K	0.5s	6.7K	0.2s
3, 3, 4	lin. queue	5465M	7196s	6.7M	55s
3, 2, 4	divergences	234M	647s	1.1M	22s
3, 2, 4	lock freedom	1354M	1015s	5.2M	28s
3, 3, 4	null refs	1454M	1679s	2.1M	10s
3, 3, 5	lin. queue	—	—	109M	1520s
3, 3, 6	divergences	—	—	98M	1638s
3, 3, 6	lock freedom	—	—	584M	3265s
3, 3, 6	null refs	—	—	98M	460s

Fig. 11. Results of analyses. The "Parameters" column shows the sizes of ThreadID, T and NodeID, respectively. In the "Check" column, "lin. queue" represents the check for being a linearizable queue (Section 4.1), "divergences" represents the divergences-based check (Section 4.2), "lock freedom" represents the deadlock-based check (Section 4.2), and "null refs" represents the check for null reference exceptions (Section 4.3).

4.4 Using symmetry reduction

Each of the above refinement checks can be run either with or without symmetry reduction. In order to use symmetry reduction, the refinement assertion is labelled with

```
: [symmetry reduce]:  diff (NodeIDType,{|Null|}), T, ThreadID
```

This tells FDR to perform symmetry reduction in the types of real node identities (excluding Null), data and thread identities. The script uses no constant from these types, other than within the definition of the types; it is shown in [8] that the model is symmetric in the types under this condition, and so the symmetry reduction is sound.

4.5 Results

As noted at the start of Section 3, the model is parameterized by three types: the type NodeID of "proper" nodes; the type T of data; and the type ThreadID of thread identities. We have used FDR to check the above assertions, for various sizes of these types. All the checks we tried succeeded.

Figure 11 gives information about some of the checks we carried out, including the number of states and the time taken (on a 32-core machine, with two 2GHz Intel(R) Xeon(R) E5-2650 0 CPUs, with 128GB of RAM, FDR version 3.4.0). Figures are given both without and with symmetry reduction.

The first block of entries is for our standard test case: here the checks are effectively instantaneous, either with or without symmetry reduction. The second block of entries is indicative of the maximum sizes of parameters that can be checked without symmetry reduction. Here, symmetry reduction gives significant reductions in both the number of states and the checking time. The third block is indicative of the maximum sizes of parameters that can be checked with symmetry reduction.

4.6 Alternative designs

It is straightforward to adapt the above model so as to consider alternative designs for the lock-free queue: this can help us understand some of the details of the original design from [16].

For example, if an enqueue finds that $\mathsf{myNext} \neq \mathsf{null}$, it attempts to advance tail via a CAS operation (line 23 of Figure 2). To investigate why this is necessary, we can remove the corresponding event from the definition of Enqueue. FDR then finds that the datatype is no longer lock-free. It finds a divergence of SystemE which corresponds to the following trace of System0

```
< beginEnqueue.T0.A, beginEnqueue.T1.A, initNode.T1.N2.A, initNode.T0.N1.A,
  getNode.T1.Tail.N0, getNext.T1.N0.Null, getNode.T0.Tail.N0,
  getNode.T1.Tail.N0, CASnext.T1.N0.Null.N2.true, getNext.T0.N0.N2,
  getNode.T0.Tail.N0, releaseRefs.T0, getNode.T0.Tail.N0 >,
```

after which the last four events can be repeated indefinitely. Thread T1 partially enqueues node N2, but fails to advance Tail to it. As a result, thread T0 repeatedly reads N0 from Tail, finds that its next reference is non-null, and retries.

A similar behaviour explains why the dequeue operation attempts to advance tail if $\mathsf{myNext} \neq \mathsf{Null}$ (line 36 of Figure 2).

5 Conclusions

In this paper we have used CSP and its model checker FDR to analyse a lock-free queue. Novel aspects include the modelling of a dynamic datatype with a mechanism for recycling nodes. We have shown how to capture linearizable specifications and lock-freedom using CSP refinement checks.

We should be clear about the limitations of our analysis. We have verified the datatype for the small values of the parameters listed in Figure 11 and a few others. This does not necessarily imply that the datatype is correct for larger parameters. However, the analyses should certainly give us great confidence in its correctness — more confidence than standard testing. It seems likely that any error in such a datatype would manifest itself for small values of the parameters.

– If there is a flaw caused by one thread's execution being interfered with by other threads, then it is likely that the actions of the interfering threads could have been performed by a single thread, and so such a flaw would manifest itself in a system of just two threads. One approach to formalise this argument would be *counter abstraction* [18, 14, 15], which counts the number of processes in each state, but in an abstracted domain.

– The enqueue and dequeue operations affect just the first two and last two nodes in the list. Hence any flaw is likely to manifest itself when there are at most two nodes in the list, and so will be captured by a system with $2 + \#\mathsf{ThreadID}$ nodes (since each enqueueing thread can hold a node that it has not yet enqueued).
– The implementation is data independent: it performs no operations on the data itself. Any flaw concerning data values (of type T) must manifest itself in an inappropriate value being dequeued, since this is the only part of the specification that cares about data values. One can create a corresponding flaw when T contains just two values A and B, by renaming the incorrectly dequeued value to A, and every other value to B. (Lazić and Roscoe have developed a general theory of data independence, formalising arguments like this; however, there results are not applicable here, because our Spec process does not satisfy their **Norm** property [19, Section 15.2].)

It might be interesting to try and formalise some of these ideas, although these are very challenging problems.

I believe that concurrent data structures are a very good target for CSP-style model checking. The algorithms are small enough that they can be accurately modelled and analysed using a model checker such as FDR. The requirements are normally clear. Yet the algorithms are complex enough that it is not obvious that they are correct.

Translating from executable code to a CSP model is straightforward. Indeed, I believe that there are good prospects for performing this translation automatically.

It also seems straightforward to produce the corresponding specifications. In particular, CSP seems well suited for capturing linearizability. The components of the specification correspond to the different aspects of the requirements: QueueSpec captures the queue behaviour, and each Linearizer captures that the actions of a particular thread are linearized. Adapting this to a different linearizable specification requires only replacing the QueueSpec, and adapting the events of Linearizer appropriately. It is interesting that this specification uses parallel composition and hiding: this creates a much clearer structure than the corresponding sequential process.

Building on the work in this paper, Chen [4] and Janssen [11] have studied a number of other concurrent datatypes, including several implementations of a set based on a linked list, a stack, a combining tree, an array-based queue, and a lock-based hash set. Some of these datatypes made use of a potentially unbounded sequence counter. However, this counter can be captured in a finite model using the observation that (in most cases) the counter is actually used as a *nonce*: the exact value of the counter is unimportant; what matters is whether the value changes between two reads. Hence we can use the technique of Roscoe and Broadfoot [21] (developed for the analysis of security protocols) for simulating an unbounded supply of nonces within a finite model.

Our approach does not require identifying the linearization points (the points at which the operations seem to take effect). However, we suspect that when there are identifiable linearization points, our check can be made more efficient: the implementation events corresponding to the linearization events can be renamed to the corresponding specification events, and those events left visible in the specification; this will often reduce the size of the search. However, it is not always possible to identify linearization points at the time they are performed; for example, for an unsuccessful dequeue the linearization point is the read of **null** for myNext (line 31 of Figure 2), but only if the subsequent re-read of head (line 32) gives an unchanged value: one can only be sure that the read of myNext was a linearization point at a *later* point in the trace.

We proved that the queue is lock-free. A related condition is *wait freedom*. A datatype is wait-free if each method call terminates within a finite number of steps. The queue we have studied is not wait-free: for example, an enqueue operation will not terminate if the CAS operation in line 19 of Figure 2 repeatedly fails: however, this would imply that other threads are repeatedly successfully enqueueing other items. Surprisingly, it turns out to be impossible to capture wait freedom using a CSP refinement check. Roscoe and Gibson-Robinson have shown [22] that every finite- or infinite-traces-based property that can be captured by a CSP refinement check can also be captured by the combination of a finite-traces refinement check and satisfaction of a *deterministic* Büchi automaton. It is reasonably straightforward to show that capturing the infinite-traces wait freedom property requires a *nondeterministic* Büchi automaton.

Acknowledgements

This work has benefited from my working with Bill Roscoe for over 25 years. It would have been impossible without his contributions to CSP and FDR. Many of the techniques used in this paper are based on his ideas.

I would like to thank Tom Gibson-Robinson for useful discussions on this work, in particular concerning our joint work on adding symmetry reduction to FDR, and for producing the CSP model that allowed comparison with the work of [12]. I would also like to thank Ke Chen and Ruben Janssen for more general discussions concerning CSP model checking of concurrent datatypes.

References

1. Rajeev Alur, Ken McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. In *Proceedings of the 11th Annual IEEE Symposium on Logic in Computer Science*, LICS '96, 1996.
2. Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-Up: A complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10)*, pages 330–340, 2010.
3. Pavol Černý, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list implementations. In *Proceedings of the 22nd International Conference on Computer-Aided Verification (CAV '10)*, pages 465–479, 2010.

4. Ke Chen. Analysing concurrent datatypes in CSP. Master's thesis, University of Oxford, 2015.
5. R. Colvin, S. Dohery, and L. Groves. Verifying concurrent data structures by simulation. *Electronic Notes in Theoretical Computer Science*, 137:93–110, 2005.
6. J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Transactions on Programming Languages and Systems*, 33(1):4:1–4:43, 2011.
7. Thomas Gibson-Robinson, Philip Armstrong, Alexandre Boulgakov, and A.W. Roscoe. *Failures Divergences Refinement (FDR) Version 3*, 2013.
8. Thomas Gibson-Robinson and Gavin Lowe. Symmetry reduction in csp model checking. Submitted for publication, 2016.
9. Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, revised first edition, 2012.
10. Maurice Herlihy and Jeannette Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.
11. Ruben Janssen. Verification of concurrent datatypes using CSP. Master's thesis, University of Oxford, 2015.
12. Yang Liu, Wei Chen, Yanhong A. Liu, Jun Sun, Shao Jie Zhang, and Jin Song Dong. Verifying linearizability via optimized refinement checking. *IEEE Transactions on Software Engineering*, 39(7):1018–1039, 2013.
13. Gavin Lowe. Implementing generalised alt — a case study in validated design using CSP. In *Communicating Process Architectures*, pages 1–34, 2011.
14. Tomasz Mazur and Gavin Lowe. Counter abstraction in the CSP/FDR setting. In *Proceedings of the Seventh International Workshop on Automated Verification of Critical Systems (AVoCS 2007)*, volume 250 of *Electronic Notes on Theoretical Computer Science*, pages 171–186, 2007.
15. Tomasz Mazur and Gavin Lowe. CSP-based counter abstraction for systems with node identifiers. *Science of Computer Programming*, 81:3–52, 2014.
16. Maged Michael and Michael Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 267–275, 1996.
17. Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of Principles of Distributed Computing (PODC 2002)*, 2002.
18. Amir Pnueli, Jessie Xu, and Lenore D. Zuck. Liveness with (0, 1, ∞)-counter abstraction. In *Proceedings of the 14th International Conference on Computer Aided Verication (CAV '02)*, pages 107–122, 2002.
19. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1998.
20. A. W. Roscoe. *Understanding Concurrent Systems.* Springer, 2010.
21. A. W. Roscoe and P. J. Broadfoot. Proving security protocols with model checkers by data independence techniques. *Journal of Computer Security*, 7(2,3):147–190, 1999.
22. A. W. Roscoe and Thomas Gibson-Robinson. The relationship between CSP, FDR and Büchi automata. Draft paper, 2016.
23. A. W. Roscoe and David Hopkins. SVA, a tool for analysing shared-variable programs. In *Proceedings of AVoCS 2007*, pages 177–183, 2007.
24. Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. *Journal of the ACM*, 53(3):379–405, 2006.
25. Håkan Sundell and Philippas Tsigas. Lock-free deques and doubly linked lists. *Journal of Parallel and Distributed Computing*, pages 1008–1020, 2008.

26. V. Vafeiadis. Automatically proving linearizability. In *Proceedings of Computer Aided Verification (CAV 2010)*, volume 6174 of *LNCS*, pages 450–46. Springer, 2010.
27. Martin Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *Proceedings of the 16th International SPIN Workshop on Model Checking Software (SPIN'09)*, pages 261–278, 2009.
28. Peter Welch and Jeremy Martin. A CSP model for Java multithreading. In *Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, pages 114–122. IEEE, 2000.