Verifying Michael and Scott's Lock-Free Queue Algorithm using Trace Reduction

Lindsay Groves

School of Mathematics, Statistics and Computer Science,
Victoria University of Wellington,
Wellington, New Zealand
Email: lindsay@mcs.vuw.ac.nz

Abstract

Lock-free algorithms have been developed to avoid various problems associated with using locks to control access to shared data structures. These algorithms are typically more intricate than lock-based algorithms, as they allow more complex interactions between processes, and many published algorithms have turned out to contain errors. There is thus a pressing need for practical techniques for verifying lock-free algorithms and programs that use them.

In this paper we show how Michael and Scott's well known lock-free queue algorithm can be verified using a trace reduction method, based on Lipton's reduction method. Michael and Scott's queue is an interesting case study because, although the basic idea is easy to understand, the actual algorithm is quite subtle, and it demonstrates several way in which the basic reduction method needs to be extended.

Keywords: Concurrency, verification, lock-free, linearisability, reduction

1 Introduction

Increasing use of concurrent software designs has prompted the development of *lock-free* algorithms to implement concurrent data structures to avoid many of the problems associated with the use of locks. Rather than avoid interference using mutual exclusion, lock-free algorithms must behave correctly in the presence of interference, and usually rely on strong synchronisation primitives such as Compare and Swap (CAS). These algorithms tend to be very subtle, and hard to get right; however, proofs of correctness for such algorithms tend to be either so high level as to be unconvincing, or so detailed as to be unenlightening.

In this paper, we consider a slightly simplified version of Michael and Scott's lock-free queue algorithm (Michael & Scott 1998), which is similar to that included in the Java concurrency library. We present a proof that this algorithm is linearisable (Herlihy & Wing 1990), using an extension of the reduction approach proposed by Lipton (Lipton 1975), and further developed by Lamport, Cohen and others (Lamport & Schneider 1989, Cohen & Lamport 1998, Lamport 1990). In this approach, we show that any concurrent execution involving a shared data object, such as a queue, can be transformed into an equivalent execution in which the operations on that object are executed without interruption, and that such uninterrupted executions correctly implement the abstract semantics for the object. This approach allows us to present a proof in sufficient detail to be convincing, highlighting the reasons why the algorithm is correct, without getting lost in a morass of minute detail, and indicating clearly what else needs to be proved to provide a more detailed proof.

We begin in Section 2 by giving an intuitive description of the algorithm and taking a brief look at the code. Then, in Section 3, we discuss our correctness criterion, linearisability, and outline how we prove linearisability using reduction. In Section 4, we present the verification, explaining in more detail how the reduction method is applied and how it is extended in order to verify Michael and Scott's algorithm, and end in Section 5 with some conclusions and comments on related and future work.

2 Michael and Scott's Lock-Free Queue Algorithm

Michael and Scott (Michael & Scott 1998) describe an algorithm which implements a shared queue supporting ENQUEUE and DEQUEUE operations that can be performed concurrently by a finite set of processes. Their algorithm is *lock-free*, which means that no process is ever forced to wait for another process to complete a queue operation. This property precludes the use of traditional synchronisation mechanisms such as locks and semaphores to avoid interference between processes; instead the algorithm is designed to work correctly in the presence of interference, which is detected by using Compare and Swap (CAS) instructions to conditionally update shared locations.

The implementation uses a linked list, with a dummy node at the head, and *Head* and *Tail* pointers. Each node has a *value* field, holding the values stored in the queue in the order they were added to the queue, and a *next* field, linking nodes in the list. Using a dummy node ensures that *Head* and *Tail* are always non-null, which reduces the number of special cases that would otherwise be required; its value is not part of the queue. *Head* always points to the dummy node, and in a quiescent state (i.e. when no operation is in progress) *Tail* points to the last node in the list, as illustrated in Figure 1, which shows an empty queue and a queue containing values *a*, *b* and *c*.

The ENQUEUE and DEQUEUE operations follow a common pattern in which each operation repeatedly attempts to perform its update, succeeding only if the operation is performed without interference. At each attempt, an operation takes a "snapshot" of the part of the global state that it wishes to update, uses this in local computations to prepare a new value, and then uses a CAS to attempt the update. $CAS(loc, old, new)$ atomically compares the contents of the shared location *loc* with the "expected" value, *old*, and if they are the same, *succeeds*, storing the new value, *new*, into the location and returning *true*, and otherwise *fails*, returning *false* and leaving the memory unchanged.

The central problem in designing algorithms based on CAS is to arrange that the shared data structure can be updated atomically using a single CAS operation, with its

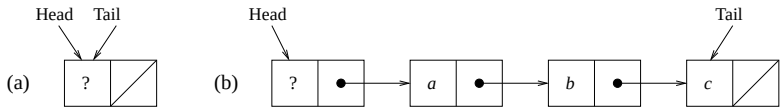

Figure 1: Basic queue representation

test determining when the update is safe. This is easy to do, say, in the case of a shared stack (e.g. (Treiber 1986, Michael & Scott 1998)), where we only need to update a single shared location (the top of stack pointer). But it is not so simple for a queue represented as a linked list. After creating a new node, an ENQUEUE operation must update two shared locations — to make the *next* pointer of the last node point to the new node, and make the *Tail* pointer point to the new node — but we can't perform both updates using a single CAS.

In performing an ENQUEUE, Michael and Scott append the new node using a CAS, and allow other processes to observe the data structure at a point where this update has been performed but the *Tail* pointer has not been advanced. This means that *Tail* may "lag" behind the actual end of the list — this situation is illustrated in Figure 2, which shows a queue containing *a*, *b* and *c*, and one containing just *c*. To avoid other processes being blocked waiting for a process to complete an ENQUEUE, as required for lock-freedom, any process which observes that *Tail* is not pointing to the end of the list attempts to advance *Tail*, effectively completing the operation of the process that performed the append. This ensures that *Tail* never lags behind the end of the list by more than one node.

DEQUEUE is implemented by advancing the *Head* pointer, provided that the queue is not empty, so the node that used to hold the first element of the queue becomes the dummy node. The DEQUEUE operation now has to handle the situation shown in Figure 2(b), where *Head* and *Tail* point to the same node, but the queue is not empty, and in this case checks whether *Tail* is lagging before attempting to perform its update.

The declarations and initialisation are shown in Figure 3, and pseudocode for the ENQUEUE and DEQUEUE operations is given in Figure 4. This code is essentially the same as that given in (Michael & Scott 1998), apart from a few changes in notation and simplifications to make our reasoning easier and more concise. In particular, we assume that a single queue is being implemented, and thus treat *Head* and *Tail* as global variables, encapsulated within a module implementing the queue, rather than as components of a record accessed via a pointer. We also use Algol/Pascal/Ada-like notation for assignment and equality (i.e. $:=$ and $=$, instead of $=$ and $==$), Ada-like parameter modes (**in** and **out**), and assume automatic pointer dereferencing, whereas Michael and Scott use C-like notation.

The most significant difference is that, like the version included in the Java concurrency library (JSR 166), we do not explicitly free popped nodes. This means that heap locations are not reused unless the algorithm is executed on a system with automatic garbage collection (as is the case in Java), and that modification counts are not required.

Looking at the code, some aspects are readily understood as they are similar to a sequential queue implementation. The declarations should be self explanatory, given the previous description of the data structure used, noting just that *new_node*() is assumed to allocate a new node and return a pointer to it.

Lines E1–E3 of ENQUEUE allocate a new node and initialise its fields. Line E9 attempts to append the new node, provided that *Tail* is not lagging. Line E13 attempts to advance *Tail* if it is found to be lagging before appending the new node, and line E17 attempts to advance *Tail* after appending the new node. The operation retries if either of the tests at lines E7 and E8, or the CAS at E9, fails.

Lines D2–D8 of DEQUEUE check to see whether the queue is empty, and if so returns *false*. If not, line D13 attempts to remove the first node from the queue by advancing *Head*, after reading the value to be returned in line D12. Line D10 attempts to advance *Tail* in the special case described above, where *Head* and *Tail* are the same but the queue is not empty.

While this much can be appreciated quite easily, it is not so clear exactly why the various tests are required or why they are ordered as they are: for example, one might consider the effect of deleting the tests at E7 and D7, and the CASes at E17 and D10, or whether DEQUEUE can be modified so as to avoid accessing *Tail*. So, although one can easily understand the basic ideas underlying the algorithm, it is not entirely obvious that the algorithm is correct, nor what changes could be made to the algorithm without affecting its correctness.

3 Proving Linearisability by Trace Reduction

When multiple processes perform concurrent operations on a shared object, we cannot simply define the correctness of these operations in terms of the state of the object before and after a process performs an operation. For example, when a process performs an ENQUEUE operation, there is no guarantee that the values that were in the queue when the enqueue operation began will still be there when the process gets to add its value to the queue, or that the enqueued value will still be in the queue when the enqueue operation is completed.

The standard safety property for concurrent data structures is *linearisability* (Herlihy & Wing 1990), which requires that each operation on the shared data structure appears to occur instantaneously at some point (called its *linearisation point*) between its invocation and its response, and that the effect of the operation be correct with respect to the state immediately before and after this point. The requirement that an operation's linearisation point be between its invocation and its response ensures that the order of non-concurrent operations is preserved; i.e. if an operation op_1 is completed before another operation op_2 begins, then the linearisation point for op_1 must precede that for op_2.

This condition is sometimes called *atomicity* (e.g. (Lynch 1996, Hesselink 2002)), however, we use that term to refer to the weaker requirement, that an operation appear to occur instantaneously, with no reference to the semantics of an abstract operation being implemented, as in (Lamport & Schneider 1989, Flanagan & Qadeer 2003, Sasturkar et al. 2005).

More precisely, a shared object *O* is *linearisable* if, for every execution of a concurrent system involving *O*, there is an "equivalent" legal sequential history, in which the order of non-concurrent operations is preserved. A *history* (or *trace*) is a sequence of invocations and responses occurring in an execution, and is *sequential* if every response is immediately preceded by an invocation of the same operation by the same process, and *legal* if each invocation-response pair is correct with respect to the abstract semantics for the object. Two histories are *equivalent* if they contain the same sequence of invocations and responses. Thus, we can prove that an implementation of a shared object is linearisable by showing, for any concurrent execution, how to construct an equivalent legal sequential history which respects the abstract semantics for that object and preserves the order of non-concurrent operations.

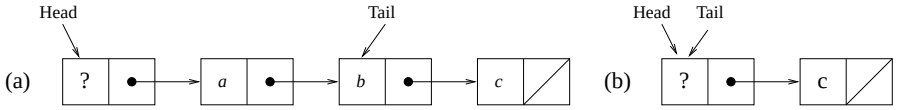

Figure 2: Queue representation with *Tail* lagging

```
type pointer = pointer to node
type node = (value: data_type, next: pointer)
var Head: pointer
var Tail: pointer

initialisation:
    Head := new_node()
    Head.next := null
    Tail := Head
```

Figure 3: Declarations and initialisation

In previous work (Doherty et al. 2004, Colvin et al. 2005, Colvin & Groves 2005, Colvin et al. 2006), we have proved linearisability of several lock-free algorithms using simulation between two labelled transition systems, one modelling the abstract specification and one modelling the implementation.[1] In the simulation approach, we step through an arbitrary execution of the concrete (implementation) model, considering all possible actions that could be taken at each step, and show how to construct a corresponding execution of the abstract (specification) model, using a simulation relation to show that the concrete and abstract states are related appropriately at each step. The simulation relation typically requires that each process is performing the same operation (if any) in both models, that the data structure in the implementation represents the same abstract value as in the abstract model, and that local variables in each process have "appropriate" values.

A well known complication with simulation proofs is that while we are often able to construct the required abstract execution by stepping forwards through the concrete execution (which is called *forward* or *downward* simulation), it is sometimes necessary to instead step backwards (which is called *backward* or *upward* simulation) or to use a combination of both (Jifeng et al. 1986, Lynch & Vaandrager 1995). Although it is widely believed that backward simulation is rarely required in practice (for example, backward simulation rules for data refinement were not defined for Z until 1997 (Stepney et al. 1998), or for B until 2003 (Dunne 2003)), backward simulation turns out to be required frequently in verifying lock-free algorithms, and several of our verifications, including our verification of a version of Michael and Scott's queue (Doherty et al. 2004), have used backward simulation, usually in conjunction with forward simulation.

While the simulation approach has proved to be effective in these verifications, and to be amenable to mechanisation (using PVS), this approach has several drawbacks:

- Translating the algorithm and specification into a transition system formalism obscures the algorithmic structure of the algorithm being verified, so it is often hard to see how verification conditions relate to the algorithm.
- Many of the verification conditions are a consequence of the formalism, rather than the algorithm (e.g. ones to do with program counters).
- The verification has to deal with both the basic operation of the data structure being implemented and the effects of concurrency, whereas it may be more convenient to separate these (this can be avoided by using an extra simulation step, but it is debatable whether this is worthwhile given the overhead introduced).
- Each abstract operation always consists of three steps (an invocation, an internal action corresponding to the linearisation point, and a response), so most steps of the implementation are internal steps, and most of the proof effort is in proving various invariants about the shared and local variables.

The net effect is that the proof requires so much detail that it is hard to identify the essential arguments on which the correctness of the algorithms relies, and it is hard to present such a proof in a way that conveys the important insights into why the algorithm is correct without getting bogged down in the details.

The approach we take in this paper is an attempt to present a proof which is both more concise and more illuminating than our earlier simulation proofs, while also being sufficiently formal to be compelling. Instead of constructing an equivalent legal sequential history by translating one action of the implementation at a time, as in the simulation approach, we first construct an equivalent sequential execution, in which each operation o the abstract object is executed without interruption, by translating an entire operation of the abstract object at a time, and then show that this sequential execution correctly implements the abstract operation. This approach is based on the reduction approach described initially by Lipton (Lipton 1975), and further developed by Lamport, Cohen, and others (Doeppner, Jr. 1977, Lamport & Schneider 1989, Lamport 1990, Cohen & Lamport 1998, Cohen 2000) for synchronisation based on mutual exclusion. We have shown how this approach can be extended to handle lock-free algorithms (Groves 2007*a*) and used it in a constructive way to derive an implementation of a scalable concurrent stack implementation (Groves & Colvin 2006*a*). The rest of this section outlines the basic ideas of reduction, and its use in proving linearisability. We will explain the approach in more detail as we work through the verification in Section 4.

Given a system in which a finite set of processes, *PROC*, operate on a shared queue, our aim, as indicated above is to show that any concurrent execution α of the system is equivalent to an execution α' in which every operation on the queue is performed without interruption, and that the effect of such an uninterrupted execution correctly implements the abstract queue semantics. The key to the reduction approach is therefore to show that the actions in a concurrent execution can be rearranged so that the steps of each operation are executed contiguously. By repeating this transformation for each operation in α, we can then produce an execution in which every operation is executed without interruption.

Suppose that α is an execution of the form $\beta_0\, a_1\, \beta_1\, \ldots\, a_n\, \beta_n$, where $a_1, \ldots, a_n$ are the atomic actions comprising an execution of a queue operation *op* by process *p*, $\beta_0 \ldots \beta_n$ are sequences of actions, and $\beta_1 \ldots \beta_{n-1}$ contain no *p*-actions. We wish to show that there is an "equivalent" execution $\alpha' = \beta_0\, \ldots\, \beta_{k-1}\, a_1\, \ldots\, a_n\, \beta_k\, \ldots\, \beta_n$ in which the atomic steps of *op* are executed contiguously. Thus, we must be able to show that it makes no difference if actions $a_1, \ldots, a_{k-1}$

[1] The transition systems we used were a simplified form of Input/Output Automata (IOAs) (Lynch & Vaandrager 1995), which were convenient because simulation between IOAs is defined in terms of trace inclusion rather than in terms of states, but most other labelled transition system formalisms could be used instead.

```
ENQUEUE(in value: data_type)
E1:  node := new_node()
E2:  node.value := value
E3:  node.next := null
E4:  loop
E5:     tail := Tail
E6:     next := tail.next
E7:     if tail = Tail then
E8:        if next = null then
E9:           if CAS(tail.next, next, node) then
E10:             break
E11:          endif
E12:       else
E13:          CAS(Tail, tail, next)
E14:       endif
E15:    endif
E16: endloop
E17: CAS(Tail, tail, node)
```

```
DEQUEUE(out pvalue: data_type): boolean
D1:  loop
D2:     head := Head
D3:     tail := Tail
D4:     next := head.next
D5:     if head = Head then
D6:        if head = tail then
D7:           if next = null then
D8:              return false
D9:           endif
D10:          CAS(Tail, tail, next)
D11:       else
D12:          pvalue := next.value
D13:          if CAS(Head, head, next) then
D14              break
D15:          endif
D16:       endif
D17:    endif
D18: endloop
D19: return true
```

Figure 4: Queue operations

are executed after the relevant β_i (i.e. for $1 \leq i < k$ and $i \leq j < k$, a_i can be executed after b_j), and actions $a_{k+1}, \ldots, a_n$ are executed before the relevant β_i (i.e. for $k \leq i \leq n$ and $k < j < i$, a_i can be executed before b_j). Since all of the actions of *op* are grouped at the position of a_k, any operation that begins before a_1 (ends after a_n) in α also begins before a_1 (ends after a_n) in α'. Thus, a_k can be taken as the linearisation point for *op*, and the order of non-current operations is preserved, as required for linearisability, so we don't need to explicitly model invocations and responses.

To define the idea of rearranging the steps in an execution more precisely, we write $\sigma \xrightarrow{a} \tau$ to mean that execution of action a may take the system from state σ to state τ. For a sequence of actions, $\alpha = a_1 \cdots a_n$, we write $\sigma \xrightarrow{\alpha} \tau$ to mean that there is a sequence of states $\rho_0, \cdots, \rho_n$ such that $\rho_0 = \sigma$, $\rho_n = \tau$, and $\rho_{i-1} \xrightarrow{a_i} \rho_i$, for all $i \in 1 \,..\, n$. For sequences of actions, α and β, we write $\alpha \leq \beta$ to mean that for any states σ and τ, $\sigma \xrightarrow{\alpha} \tau$ implies $\sigma \xrightarrow{\beta} \tau$. An action a is *enabled* in a state σ if there exists a state τ such that $\sigma \xrightarrow{a} \tau$.

If $ab \leq ba$, we say that *a right commutes* with *b*, and *b left commutes* with *a*. If *a* right commutes and left commutes with *b*, we just say *a commutes* with *b*, and write $ab = ba$. We can show that for sequences of actions, $\alpha = a_1 \ldots a_m$ and $\beta = b_1 \ldots b_n$, $\alpha\beta \leq \beta\alpha$ if $a_i b_j \leq b_j a_i$ for all $i \in 1 \,..\, m$ and $j \in 1 \,..\, n$.

Given a system with actions *ACT*, an action *a* is called a *right mover* if it right commutes with every action of every other process (i.e. $a_p b_q \leq b_q a_p$ for all $b \in ACT$ and $p \neq q$), a *left mover* if it left commutes with every action of every other process (i.e. $b_q a_p \leq a_p b_q$ for all $b \in ACT$ and $p \neq q$), and a *both mover* if it is both a right mover and a left mover (i.e. $a_p b_q = b_q a_p$ for all $b \in ACT$ and $p \neq q$).

In showing that actions move in particular ways, we appeal to some standard properties of independent operations. For example:

- An action that only accesses local variables or heap locations accessed via a unique pointer in a local variable is a both mover.
- An action that reads a shared variable commutes with any action that does not assign to that variable.
- An action that assigns to a shared variable commutes with any action that does not refer to that variable.

As presented above, the equivalent sequential execution is obtained by rearranging the actions of the concurrent execution. It has been shown (Wang & Stoller 2005) that this is not sufficient to verify some lock-free algorithms. However, the technique can be extended to extend its applicability (Groves 2007*a*). Firstly, we can use the outcome of CAS and other tests to infer properties of the interleaved actions of other processes. Secondly, we observe that while the steps in the sequential execution need to be steps that could be taken by the implementation when executed without interruption, they do not have to be the same steps as in the concurrent execution. In many lock-free algorithms we need to be able to delete actions. We will see in Section 4 that to verify Michael and Scott's queue algorithm, we also need to be able to modify actions so as to assign an action to a different process.

4 Verification

We now consider how the version of Michael and Scott's algorithm presented in Section 2 can be verified using the trace reduction approach described in Section 3. As outlined earlier, our aim is to show that any concurrent execution can be transformed into one in which the atomic steps of each queue operation are executed contiguously, and that when executed without interruption these operations correctly implement the abstract queue semantics. Here, we focus on the former; the latter involves a straightforward data refinement proof, which we present elsewhere (Groves 2007*b*).

We will regard each assignment, test and CAS as an atomic action, which is reasonable since they all access at most one shared variable. We also assume, as in (Michael & Scott 1998) that allocating a new node can be treated as an atomic action. For convenience, the atomic actions and their labels are shown in Figure 5.

4.1 Commutativity properties

The first step in applying the reduction method is to examine the commutativity properties of the atomic actions. From the general properties given at the end of Section 3, and some other general properties, we can see that:

- Actions E8 (*next* = *null*), D6 (*head* = *tail*) and D7 (*next* = *null*) are both movers, as they only involve local variables. Thus, for any action X and distinct processes, p and q, we have:

  $E8_p\, X_q = X_q\, E8_p$, $D6_p\, X_q = X_q\, D6_p$ and $D7_p\, X_q = X_q\, D7_p$

- Actions E2 (*node.value* := *value*) and E3 (*node.next* := *null*) are both movers, since at the point where they are executed, *node* is a unique

E1	node := *new_node*()
E2	node.value := value
E3	node.next := *null*
E5	tail := Tail
E6	next := tail.next
E7	tail = Tail
E8	next = *null*
E9	CAS(tail.next, next, node)
E13	CAS(Tail, tail, next)
E17	CAS(Tail, tail, node)

D2	head := Head
D3	tail := Tail
D4	next := head.next
D5	head = Head
D6	head = tail
D7	next = *null*
D10	CAS(Tail, tail, next)
D12	*pvalue* := next.value
D13	CAS(Head, head, next)

Figure 5: Atomic actions

pointer (*node* is a new node when it is allocated in E1, and cannot be seen by any other process until it is appended to the end of the list by the CAS at E9), and *value* and *null* are local. Thus, for any action X and distinct processes, p and q, we have:

$E2_p\, X_q = X_q\, E2_p$ and $E3_p\, X_q = X_q\, E3_p$

- Actions E5 (*tail* := *Tail*), E7 (*tail* = *Tail*) and D3 (*tail* := *Tail*) commute with any actions that do not alter *Tail*. Thus, for any action X other than E13 or E17 and distinct processes, p and q, we have:

  $E5_p\, X_q = X_q\, E5_p$, $E7_p\, X_q = X_q\, E7_p$ and $D3_p\, X_q = X_q\, D3_p$

- Actions D2 (*head* := *Head*) and D5 (*head* = *Head*) commute with any action that does not alter *Head*. Thus, for any action X other than D13 and distinct processes, p and q, we have:

  $D2_p\, X_q = X_q\, D2_p$ and $D5_p\, X_q = X_q\, D5_p$

- Actions E6 (*next* := *tail.next*) and D4 (*next* := *head.next*) commute with any action that does not alter the *next* field of a node. Thus, for any action X other than E9 and distinct processes, p and q, we have:

  $E6_p\, X_q = X_q\, E6_p$ and $D4_p\, X_q = X_q\, D4_p$

- Action D12 (*pvalue* := *next.value*) commutes with any action that does not alter the *value* field of a node. Since the only action that alters the *value* field of a node is E2, and we have already shown that E2 is a both mover because it updates *value* via a unique pointer, it follows that D12 commutes with all actions. Thus, for any action X and distinct processes, p and q, we have:

  $D12_p\, X_q = X_q\, D12_p$

  Note that we assume that the value of an **out** parameter is not observable to the caller (or any other process) until the queue operation is completed, so in the case of D12, we can treat *pvalue* as being local.

- Action E9 ($CAS(tail.next, next, node)$) commutes with any action that does not access the *next* field of a node. Thus, for any action X other than E6, E9 or D4 and distinct processes, p and q, we have:

  $E9_p\, X_q = X_q\, E9_p$

  Note that E3 is not included in the list of exceptions since we have already shown that E3 is a both mover.

- Actions E13, E17 and D10 ($CAS(Tail, tail, node)$) commute with any action that does not access *Tail*. Thus, for any action X other than E5, E7, E13, E17, D3 or D10 and distinct processes, p and q, we have:

  $E13_p\, X_q = X_q\, E13_p$, $E17_p\, X_q = X_q\, E17_p$ and $D10_p\, X_q = X_q\, D10_p$.

- Action D13 ($CAS(Head, head, next)$) commutes with any action that does not access *Head*. Thus, for any action X other than D2, D5 or D13 and distinct processes, p and q, we have:

  $D13_p\, X_q = X_q\, D13_p$

- Action E1 (*node* := *new_node*()) is a both mover, since it makes no difference what address is allocated provided that it is previously unused. Thus, for any action X and distinct processes, p and q, we have:

  $E1_p\, X_q = X_q\, E1_p$

4.2 Applying reduction to Michael and Scott's algorithm

Next, we consider how to use these properties to rearrange the steps in a concurrent execution to obtain an equivalent sequential execution. Here we find that these properties are not sufficient to show that Michael and Scott's algorithm is atomic — a completed execution of ENQUEUE or DEQUEUE may contain any number of CAS actions, which may be interleaved with CAS actions of other processes, and the above commutativity properties do not allow us to permute the order of CAS actions. We therefore need to perform a more complex transformation than just reordering the steps of a concurrent execution.

In considering how to transform a concurrent execution into a sequential one, we first observe that any completed execution of a queue operation consists of zero or more "failed" iterations of the loop (i.e. ones where the loop does not exit), preceded in the case of ENQUEUE by three initial actions (E1-E3), and followed by one "successful" iteration (i.e. one where the loop does exit). We also observe that in a sequential execution, every operation succeeds the first time it is attempted, so there are no failed iterations, and *Tail* is always updated by the process that appends a node onto the list (at E17), so E13 and D10 are never executed.

We will show how to transform an arbitrary concurrent execution into this form using three transformations, each of which requires an extension to Lipton's basic method. Firstly, we show that "failed" iterations in which *Tail* is not updated can be deleted; secondly, we show that the remaining actions can be rearranged so that the steps of each operation execution are contiguous, except for "successful" iterations in which *Tail* is not updated; and lastly, we show that actions that advance *Tail* can be performed by any process, and in particular by the process that last appended a node, which allows this exception to be addressed. Finally, we consider how to assemble the remaining fragments into complete operations.

4.3 Primitive paths

In describing these transformations, we need to consider different paths that an operation may take through the code. So we break the code into primitive (loop-free) paths and describe how each path is transformed.

We need to identify a set of primitive paths, each comprising a sequence of atomic actions performed by the

same process, such that every execution of a queue operation can be described as a concatenation of primitive paths. We will segment the code so that each primitive path consists of either a sequence of actions performed before the loop (which only occurs in ENQUEUE) or one iteration of the loop (including actions taken after exiting the loop in the case where the loop in DEQUEUE terminates). We further divide loop iterations into four classes which will be handled differently:

- failed iterations in which *Tail* is not updated (i.e. Enq2, Enq3, Enq5, Deq1, Deq3 and Deq5);
- failed iterations in which *Tail* is updated (i.e. Enq4 and Deq2);
- normal successful iterations, which behave as they would in a sequential execution (i.e. Enq7, Deq4 and Deq6); and
- abnormal successful iterations, which do not behave as they would in a sequential execution (i.e. Enq6).

The resulting paths are shown in Figure 6, and are labelled for later reference. In describing execution paths, we use the line numbers shown in Figure 4 to stand for the action on that line, and indicate whether test and CAS actions succeed or fail by appending $^+$ or $^-$, respectively. Where necessary, we indicate the process that performs an operation by adding a process identifier (usually *p* or *q*) as a subscript (these should not be confused with the numerical subscripts used in describing arbitrary actions and action sequences).

It follows from the semantics of our programming constructs that any execution of ENQUEUE consists of the initial segment (Enq1) followed by zero or more failed iterations (Enq2 to Enq5), followed by a single successful iteration (Enq6 or Enq7). Similarly, any execution of DEQUEUE consists of zero or more failed iterations (Deq1, Deq2, Deq3 or Deq5), followed by a single successful iteration (Deq4 or Deq6). Treating the path names as symbols, we can describe the structure of possible executions of ENQUEUE and DEQUEUE with the following regular expressions:

$$\mathit{Enq}1(\mathit{Enq}2 \mid \mathit{Enq}3 \mid \mathit{Enq}4 \mid \mathit{Enq}5)^*(\mathit{Enq}6 \mid \mathit{Enq}7)$$

$$(\mathit{Deq}1 \mid \mathit{Deq}2 \mid \mathit{Deq}3 \mid \mathit{Deq}5)^*(\mathit{Deq}4 \mid \mathit{Deq}6)$$

4.4 Deleting failed iterations that do not advance *Tail*

We first show that any failed iteration that does not advance *Tail* can be deleted. This is easy to see intuitively — an execution in which an operation is attempted unsuccessfully is indistinguishable from one in which the unsuccessful operation was never attempted.

More precisely, let α be an execution which contains a failed iteration that does not advance *Tail* in ENQUEUE (i.e. Enq2, Enq3, Enq5) or in DEQUEUE (i.e. Deq1, Deq3 or Deq5), and let α' be the result of deleting the steps of this failed iteration from α. Then we wish to show that $\alpha \leq \alpha'$.

We will only consider path Enq2 in detail — the arguments for the other failed iterations are similar.

Path Enq2 is: E5, E6, E7$^-$. Let α be an execution containing an execution of Enq2 as part of a completed execution of ENQUEUE by process *p*, say $\alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^-\, \alpha_4$, where α_2 and α_3 contain no *p*-actions. Removing $E7_p^-$ from this execution does not alter its effect. These occurrences of $E5_p$ and $E6_p$ can then also be removed — since this occurrence of Enq2 is part of a completed ENQUEUE operation, the next two *p*-actions must be E5 and E6, so the values loaded by these occurrences of E5 and E6 will not be referenced again. Thus, we have:

$$\alpha_1\, E5\, \alpha_2\, E6\, \alpha_3\, E7^-\, \alpha_4 \leq \alpha_1\, \alpha_2\, \alpha_3\, \alpha_4$$

This result means that we can ignore all of the failed iterations that do not advance *Tail*, i.e. Enq2, Enq3, Enq5, Deq1, Deq3 and Deq5. Following this transformation, every execution of ENQUEUE or DEQUEUE has the form described by the following regular expressions:

$$\mathit{Enq}1(\mathit{Enq}4)^*(\mathit{Enq}6 \mid \mathit{Enq}7)$$

$$(\mathit{Deq}3)^*(\mathit{Deq}4 \mid \mathit{Deq}6)$$

Notice that the result of this transformation (like the subsequent ones) is a valid execution of the algorithm.

This transformation can be generalised to show that any failed iteration which has no observable effect can be deleted. Such iterations are called "pure" in (Freund & Qadeer 2005), where a similar approach is used in a static analysis technique for determining atomicity.

4.5 Reducing primitive paths

We now consider the remaining basic paths and attempt to show how a concurrent execution, in which the atomic actions of that path may be interleaved with actions of other processes, can be transformed into one in which the atomic steps of that path are executed without interruption. This uses the basic reduction method, augmented with a more detailed analysis of paths containing CASes, and succeeds for all of the remaining paths except failed iterations that update *Tail* (i.e. Enq4 and Deq2), which are considered further in Section 4.6.

The linearisation point for a completed ENQUEUE is the successful CAS at E9, so we want to move everything before that to the right (or delete it), and everything after (i.e. the CAS at E17) to the left. Similarly, the linearisation point for a completed DEQUEUE returning *true* is the successful CAS at D13, and for a DEQUEUE returning *false* is D3, so we want to move everything before that to the right (or delete it).

The important points can be illustrated by considering five cases; the other cases are detailed in (Groves 2007*b*).

4.5.1 Pre-loop path in ENQUEUE

Path Enq1 is E1, E2, E3, where we have:

E1	node := *new_node*()
E2	node.value := value
E3	node.next := *null*

Let α be an execution containing an execution of Enq1 by process *p*, say $\alpha = \alpha_1\, E1_p\, \alpha_2\, E2_p\, \alpha_3\, E3_p\, \alpha_4$, where α_2 and α_3 contain no *p*-actions. We have shown that E1, E2 and E3 are both-movers, so these actions can be moved right over α_2 and α_3 as required. Thus, we have:

$$\alpha_1\, E1_p\, \alpha_2\, E2_p\, \alpha_3\, E3_p\, \alpha_4 \leq \alpha_1\, \alpha_2\, \alpha_3\, E1_p\, E2_p\, E3_p\, \alpha_4$$

4.5.2 Normal successful iteration in ENQUEUE

Path Enq7 is E5, E6, E7$^+$, E8$^+$, E9$^+$, E17$^+$, where we have:

E5	tail := Tail
E6	next := tail.next
E7$^+$	tail = Tail
E8$^+$	next = *null*
E9$^+$	CAS(tail.next, next, node)$^+$
E17$^+$	CAS(Tail, tail, node)$^+$

Let α be an execution containing an execution of Enq7 by process *p*, say $\alpha = \alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^+\, \alpha_4\, E8_p^+\, \alpha_5\, E9_p^+\, \alpha_6\, E17_p^+\, \alpha_7$, where α_2 to α_6 contain no *p*-actions.

Enq1	E1, E2, E3	Pre-loop
Enq2	E5, E6, $E7^-$	Failed iteration, not updating *Tail*
Enq3	E5, E6, $E7^+$, $E8^-$, $E13^-$	Failed iteration, not updating *Tail*
Enq4	E5, E6, $E7^+$, $E8^-$, $E13^+$	Failed iteration, updating *Tail*
Enq5	E5, E6, $E7^+$, $E8^+$, $E9^-$	Failed iteration, not updating *Tail*
Enq6	E5, E6, $E7^+$, $E8^+$, $E9^+$, $E17^-$	Normal successful iteration
Enq7	E5, E6, $E7^+$, $E8^+$, $E9^+$, $E17^+$	Abnormal successful iteration
Deq1	D2-D4, $D5^-$	Failed iteration, not updating *Tail*
Deq2	D2-D4, D5+, $D6^+$, $D7^-$, $D10^+$	Failed iteration, updating *Tail*
Deq3	D2-D4, D5+, $D6^+$, $D7^-$, $D10^-$	Failed iteration, not updating *Tail*
Deq4	D2-D4, $D5^+$, $D6^+$, $D7^+$	Normal successful iteration
Deq5	D2-D4, $D5^+$, $D6^-$, D12, $D13^-$	Failed iteration, not updating *Tail*
Deq6	D2-D4, $D5^+$, $D6^-$, D12, $D13^+$	Normal successful iteration

Figure 6: Basic paths for ENQUEUE and DEQUEUE

We can move E8 because it only involves local variables, but the other actions require move careful consideration.

Since E17 succeeds, we know that *Tail* has the same value at E17 as it had at E5; however, we can go further and infer that *Tail* is not modified by α_2 to α_6. To see why, we observe that *Tail* can only be modified by a successful CAS at E13, E17 or D10, and that the last such CAS must set *Tail* to *tail*. We can show that this is impossible, by showing that the program maintains the invariant property that the list contains no cycles and *new_node*() always returns a new node, so advancing *Tail* cannot cause it to return to a previous value. This is called the "ABA freedom property", and holds because we assume that memory is not recycled. It follow that E5 and E7 can move right over α_2 to α_5 as required, and E17 can move left over α_6.

Similarly, since E9 succeed, we can infer that *tail.next* is not modified by α_3 to α_5, since this can only be done by a successful CAS at E9, which always sets *tail.next* to a new node previously allocated at E1 which no other process can see. It follows that E6 can move right over α_3 to α_5.

Thus, we move E5 to E8 right and E17 left to the position of E9, so the steps of Enq7 are contiguous; so we have:

$$\alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^+\, \alpha_4\, E8_p^+\, \alpha_5\, E9_p^+\, \alpha_6\, E17_p^+\, \alpha_7 \;\leq\; \alpha_1\, \alpha_2\, \alpha_3\, \alpha_4\, \alpha_5\, E5_p\, E6_p\, E7_p^+\, E8_p^+\, E9_p^+\, E17_p^+\, \alpha_6\, \alpha_7$$

Path Deq6 is handled in essentially the same way.

4.5.3 Failed iteration in ENQUEUE advancing *Tail*

We consider path Enq4, i.e. E5, E6, $E7^+$, $E8^-$, $E13^+$, where we have:

E5	tail := Tail
E6	next := tail.next
$E7^+$	tail = Tail
$E8^-$	next $\neq$ *null*
$E13^+$	CAS(Tail, tail, next)$^+$

Let α be an execution containing an execution of Enq4 by process *p*, say $\alpha = \alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^+\, \alpha_4\, E8_p^-\, \alpha_5\, E13^+\, \alpha_6$, where α_2 to α_5 contain no *p*-actions. We will show that actions E5–E8 can be moved right to the position of E13. We know that E8 is a both mover, since if only involve local variables.

We can also treat E6 as a right mover, since we can show that for any node *n*, *n.next* is only ever assigned twice: once at E3 when it is set to *null*, and once at E9 when it is set to a non-*null* value (note that E9 can only be executed when *next* = *null*). Thus, since we know from $E8^+$ that *tail.next* was not *null* when it was read at E6, it cannot be assigned again.

Finally, since the CAS at E13 succeeds, we can infer from the ABA freedom property that *Tail* is not assigned by α_2 to α_5. Thus, we have:

$$\alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^+\, \alpha_4\, E8_p^-\, \alpha_5\, E13_p^+\, \alpha_6 \;\leq\; \alpha_1\, \alpha_2\, \alpha_3\, \alpha_4\, \alpha_5\, E5_p\, E6_p\, E7_p^+\, E8_p^-\, E13_p^+\, \alpha_6$$

Path Deq2 is handled in essentially the same way.

4.5.4 Normal successful iteration in DEQUEUE

Path Deq4 is D2-D4, $D5^+$, $D6^+$, $D7^+$, where we have:

D2	head := Head
D3	tail := Tail
D4	next := head.next
$D5^+$	head = Head
$D6^+$	head = tail
$D7^+$	next = *null*

Let α be an execution containing an execution of Deq4 by process *p*, say $\alpha = \alpha_1\, D2_p\, \alpha_2\, D3_p\, \alpha_3\, D4_p\, \alpha_4\, D5_p^+\, \alpha_5\, D6_p^+\, \alpha_6\, D7_p^+\, \alpha_7$, where α_2 to α_6 contain no *p*-actions. We can infer from the tests, and the ABA freedom property, that *Head* is not modified by α_2 to α_4 and *head.next* is not modified by α_3, but we don't know anything about whether *Tail* is changed. We therefore move D2 right over α_2, and D4 to D7 left over α_3 to α_7 as required. Thus, we have:

$$\alpha_1\, D2_p\, \alpha_2\, D3_p\, \alpha_3\, D4_p\, \alpha_4\, D5_p^+\, \alpha_5\, D6_p^+\, \alpha_6\, D7_p^+\, \alpha_7 \;\leq\; \alpha_1\, \alpha_2\, D2_p\, D3_p\, D4_p\, D5_p^+\, D6_p^+\, D7_p^+\, \alpha_3\, \alpha_4\, \alpha_5\, \alpha_6\, \alpha_7$$

4.5.5 Abnormal successful iteration

Path Enq6 is E5, E6, $E7^+$, $E8^+$, $E9^+$, $E17^-$, where we have:

E5	tail := Tail
E6	next := tail.next
$E7^+$	tail = Tail
$E8^+$	next = *null*
$E9^+$	CAS(tail.next, next, node)$^+$
$E17^-$	CAS(Tail, tail, node)$^-$

Let α be an execution containing an execution of Enq6 by process *p*, say $\alpha = \alpha_1\, E5_p\, \alpha_2\, E6_p\, \alpha_3\, E7_p^+\, \alpha_4\, E8_p^+\, \alpha_5\, E9_p^+\, \alpha_6\, E17_p^-\, \alpha_7$, where α_2 to α_6 contain no *p*-actions. E8 which is a both mover, since it only involves local variables. Since E7 succeeds, we know that *Tail* is not modified by α_2 or α_3, and since E9 succeeds, we know that α_3 to α_5 do not modify *tail.next*. Thus, there are various ways in which we can move E5 to E9 so that they are contiguous.

However, the fact that E17 fails means that *Tail* is changed some where in α_4 to α_6. Therefore, we can't move E5 to E9 right over α_6, not can we move E17 left over α_6. So at this stage, we cannot rearrange an execution of Enq6 to make its steps contiguous.

4.6 Reassigning *Tail* advance steps

We have now reduced all of the primitive paths so that their steps are contiguous, with the exception of Enq6, where the CAS at E17 fails because *Tail* has been updated by another process. We will address this case by showing that there is an equivalent execution in which the process that appends a node also updates *Tail* (i.e. all E17 actions succeed), which means that paths Enq4, Enq6 and Deq2 never occur.

Let α be an execution which contains a failed E17 action performed by process *p*. The fact that this action fails means that some another process, say *q*, has updated *Tail*, with a successful CAS at E13 or D10, as part of an Enq4 or Deq2 path, since *p* performed its successful CAS at E9. If more than one process has updated *Tail* since *p* performed its successful CAS at E9, we chose *q* to be the first such process. If *q* performs an E13 action, α is of the form $\alpha_1\, E9_p^+\, \alpha_2\, E13_q^+\, \alpha_3\, E17_p^-\, \alpha_4$, where α_2 does not contain any E10 or D10 action (note that α_2 also cannot contain a successful E17 action). We can now construct an equivalent execution α' in which *p* performs a successful E17 action at the point where *q* performed its successful E13 in α, and *q* performs an unsuccessful E13 action at the point where *p* performed its unsuccessful E17 action in α. Thus, we have:

$$\alpha_1\, E9_p^+\, \alpha_2\, E13_q^+\, \alpha_3\, E17_p^-\, \alpha_4 \leq \alpha_1\, E9_p^+\, \alpha_2\, E17_p^+\, \alpha_3\, E13_q^-\, \alpha_4$$

The case where *q* performs a D10 action is symmetrical, giving:

$$\alpha_1\, E9_p^+\, \alpha_2\, D10_q^+\, \alpha_3\, E17_p^-\, \alpha_4 \leq \alpha_1\, E9_p^+\, \alpha_2\, E17_p^+\, \alpha_3\, D10_q^-\, \alpha_4$$

The key observation here is that it doesn't matter what process performs a step that advances *Tail*. By assigning this step to the process which performed the closest preceding E9, we ensure that the resulting execution can be generated by the queue algorithm.

The effect of this transformation is to either swap an occurrence of Enq6 and an occurrence of Enq4 for an occurrence of Enq7 and an occurrence of Enq3, or swap an occurrence of Enq6 and an occurrence of Deq2 for an occurrence of Enq7 and an occurrence of Deq2. The result is that all Enq6 paths become Enq7 paths, which can now be reduced as shown in Section 4.5.2, and all Enq4 and Deq2 paths become Enq3 and Deq3 paths, respectively, which can now be deleted as shown in Section 4.4.

4.7 Assembling the remaining fragments

Following the above transformation, every execution of ENQUEUE or DEQUEUE has the form shown by the following regular expressions:

$$\mathit{Enq}1\ \mathit{Enq}6$$

$$\mathit{Deq}4 \mid \mathit{Deq}6$$

Finally, we observe that since all of the steps in Enq1 are both-movers, these steps can be moved right over any steps that occur between the executions of Enq1 and Enq6 by the same process. Thus, provided α_2 contains no *p* actions, we have:

$$\alpha_1\, \mathit{Enq}_p\, \alpha_2\, \mathit{Enq}6_p\, \alpha_3 \leq \alpha_1\, \alpha_2\, \mathit{Enq}_p\, \mathit{Enq}6_p\, \alpha_3$$

With a little simplification, it follows that ENQUEUE is equivalent to:

```
node := new_node()
node.value := value
node.next := null
tail.next := node
Tail := node
```

and DEQUEUE is equivalent to:

```
if Head = Tail then
  Head.next = null
  return false
else
  pvalue := Head.next.value
  Head := Head.next
  return true
```

It is easy to see that these correctly implement the queue operations with the chosen data representation.

5 Conclusions

We have shown how a version of Michael and Scott's lock-free queue can be proved to be linearisable, using a reduction method based on that of Lipton, Lamport, Cohen, and others. This approach separates reasoning about the concurrent and non-concurrent aspects of the algorithm, and addresses the concurrent part by focusing on the interactions between actions performed by different processes. This allows us to explain why the algorithm is correct in a way that is more compelling that a higher level proof, and provides more insight than a simulation proof, since it highlights properties (such as ABA freedom, unique pointers and fields not changing) on which the correctness of the algorithm relies. Some of these properties can be easily checked by inspection, or verified more rigorously using static analysis techniques; others require more sophisticated verification using model checking or theorem proving. Moreover, similar supporting properties are required in the verification of other lock-free algorithms.

Our trace reduction method extends Lipton's reduction method in several ways: we used a form of conditional reduction, where reductions depend on the outcomes of tests and CASes; we allow loop iterations that have no effect to be deleted (this is called *purity* in (Freund & Qadeer 2005)); we also allow certain actions to be assigned to other processes. This can be done because these actions could in fact be performed by any process, and would be required in verifying other algorithms using similar "helper" mechanisms, such as Shann et al's array-based queue (Shann et al. 2000) and Ladan-Mozes and Shavit's optimistic queue (Ladan-Mozes & Shavit 2004). In other work (Groves & Colvin 2006*b*) we have shown that algorithms such as the scalable stack described in (Hendler et al. 2004), where the linearisation point for one operation may be a step of another process, can be handled by by reducing two operations simultaneously.

We have simplified the original algorithm by assuming that storage is never recycled (or that the implementation language provides automatic garbage collection), which allows us to justify the ABA Freedom assumption. To justify this assumption while recycling storage, Michael and Scott add version numbers to pointer variables, which are incremented every time a pointer is modified. This can be introduced in our context as a further data refinement, but is only strictly correct if version numbers are unbounded. An alternative approach which avoids this problem is described in (Herlihy et al. 2002).

Michael and Scott (Michael & Scott 1998) gave a brief proof of some safety properties, but they were not sufficient to ensure linearisability. Yahav and Sagiv (Yahav & Sagiv 2003) describe an approach to verifying Michael and Scott's safety properties using model checking, but

their analysis appears to be very limited as they don't appear to have run the system with both ENQUEUEs and DEQUEUEs being performed.

Wang and Stoller (Wang & Stoller 2005) describe a static analysis technique for checking atomicity, and apply it to a variant of Michael and Scott's algorithm which avoids the ABA problem by using the less widely available Linked Load/Store Conditional instructions instead of CAS. However, their variant also avoids the main problem addressed in the paper by using a separate process to update *Tail*, which destroys the lock-freedom of the algorithm (since if that process dies the entire system will deadlock).

Doherty et al (Doherty et al. 2004) describe a fully mechanical proof of a variant of Michael and Scott's which is intended to reduce contention in the DEQUEUE operation by testing $next = null$ at D6, to determine whether the queue is empty, rather than $head = tail$, and only reading *Tail* if this test succeeds. This optimisation was discovered while attempting to prove the original algorithm. In our context, this modification would simplify the reasoning about path Deq2. This verification uses simulation between Input/Output Automata (Lynch 1996, Lynch & Vaandrager 1995), and requires a combination of forward and backward simulation to handle DEQUEUE on an empty queue since at the time *Tail* is read it is not known whether the algorithm will return *false*. Our proof requires no special treatment for this case.

Abrial and Cansell (Abrial & Cansell 2005) describe a constructive verification of a variant of Michael and Scott's algorithm using Event-B. They prove a variant of linearisability in which they require the linearisation point to be the last step taken by an operation, and delete line E17 from the algorithm so that *Tail* is always advanced by the next operation that notices *Tail* lagging, at E9 or D10. They also introduce an additional test in DEQUEUE, which requires *Tail* to be read again, before returning *false*. This is precisely the case that required a backward simulation in the verification in (Doherty et al. 2004), and this modification appears to have been required to avoid the need for backward simulation.

It would require a straightforward modification of our proof to show that the variants of Michael and Scott's algorithm described by (Wang & Stoller 2005), (Doherty et al. 2004) and (Abrial & Cansell 2005) are correct, and that the handling of DEQUEUE on an empty queue can be further simplified so that it never needs to access *Tail*.

Our future work will include mechanising our reduction proofs using PVS, and applying the approach to more sophisticated algorithms, such as the optimistic queue described in (Ladan-Mozes & Shavit 2004) and the scalable queue described in (Moir et al. 2005), to see whether other extensions are required and what other properties are required to justify its application.

Acknowledgements We are grateful to Sun Microsystems Laboratories for financial support, and to Rob Colvin and Mark Moir for helpful discussions relating to this work.

References

Abrial, J.-R. & Cansell, D. (2005), 'Formal construction of a non-blocking concurrent queue algorithm', *Journal of Universal Computer Science* **11**(5), 744–770.

Cohen, E. (2000), Separation and reduction, *in* 'Proc. 5th International Conference on Mathematics of Program Construction (MPC)', Springer-Verlag, London, UK, pp. 45–59.

Cohen, E. & Lamport, L. (1998), Reduction in TLA, *in* 'International Conference on Concurrency Theory (CONCUR)', pp. 317–331.

Colvin, R., Doherty, S. & Groves, L. (2005), Verifying concurrent data structures by simulation, *in* E. Boiten & J. Derrick, eds, 'Proc. Refinement Workshop (REFINE 2005)', Vol. 137(2) of *Electronic Notes in Theoretical Computer Science*, Elsevier, Guildford, UK, pp. 93–110.

Colvin, R. & Groves, L. (2005), Formal verification of an array-based nonblocking queue, *in* 'Proc. International Conference on Engineering of Complex Computer Systems (ICECCS)', ACM Press, New York, NY, USA, pp. 92–101.

Colvin, R., Groves, L., Luchangco, V. & Moir, M. (2006), Formal verification of a lazy concurrent list-based set algorithm, *in* T. Ball & R. B. Jones, eds, 'Proc. 18th International Conference on Computer Aided Verification (CAV)', Vol. 4144 of *Lecture Notes in Computer Science*, Springer, pp. 475–488.

Doeppner, Jr., T. W. (1977), Parallel program correctness through refinement, *in* 'Proc. 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)', ACM Press, pp. 155–169.

Doherty, S., Groves, L., Luchangco, V. & Moir, M. (2004), Formal verification of a practical lock-free queue algorithm., *in* D. de Frutos-Escrig & M. Núñez, eds, 'Formal Techniques for Networked and Distributed Systems (FORTE)', Vol. 3235 of *Lecture Notes in Computer Science*, Springer, pp. 97–114.

Dunne, S. (2003), Introducing backward refinement into B, *in* D. Bert, J. P. Bowen, S. King & M. A. Waldén, eds, 'ZB', Vol. 2651 of *Lecture Notes in Computer Science*, Springer, pp. 178–196.

Flanagan, C. & Qadeer, S. (2003), A type and effect system for atomicity, *in* 'Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation', pp. 338–349.

Freund, S. N. & Qadeer, S. (2005), 'Exploiting purity for atomicity', *IEEE Trans. Softw. Eng.* **31**(4), 275–291.

Groves, L. (2007*a*), Reasoning about nonblocking concurrency using reduction, *in* 'Proc. 12th Twelfth IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS 2007)', Auckland, New Zealand, pp. 107–116.

Groves, L. (2007*b*), Verifying Michael and Scott's lock-free queue algorithm using trace reduction — the details, Technical report, Victoria University of Wellington. (To appear).

Groves, L. & Colvin, R. (2006*a*), Derivation of a scalable lock-free stack algorithm, *in* 'International Refinement Workshop (Refine 2006)', Electronic Notes in Theoretical Computer Science, Elsevier.

Groves, L. & Colvin, R. (2006*b*), Derivation of a scalable lock-free stack algorithm, *in* 'International Refinement Workshop (Refine 2006)', Electronic Notes in Theoretical Computer Science, Elsevier.

Hendler, D., Shavit, N. & Yerushalmi, L. (2004), A scalable lock-free stack algorithm, *in* 'SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain', pp. 206–215.

Herlihy, M., Luchangco, V. & Moir, M. (2002), The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures, *in* '16th International Conference on Distributed Computing (DISC 2002)', Vol. 2508 of *Lecture Notes in Computer Science*, Toulouse, France, pp. 339–353.

Herlihy, M. P. & Wing, J. M. (1990), 'Linearizability: a correctness condition for concurrent objects', *TOPLAS* **12**(3), 463–492.

Hesselink, W. H. (2002), 'An assertional criterion for atomicity', *Acta Informatica* **28**(5), 343–366.

Jifeng, H., Hoare, C. & Sanders, J. (1986), Data refinement refined, *in* 'ESOP 86', Vol. 213 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 187–196.

Ladan-Mozes, E. & Shavit, N. (2004), An optimistic approach to lock-free fifo queues, *in* 'Proc. of the 18th International Conference on Distributed Computing', pp. 117–131.

Lamport, L. (1990), 'A theorem on atomicity in distributed algorithms', *Distributed Computing* **4**(2), 59–68.

Lamport, L. & Schneider, F. B. (1989), Pretending atomicity, Technical Report TR89-1005, DEC, SRC.

Lipton, R. J. (1975), 'Reduction: a method of proving properties of parallel programs', *Communications of the ACM* **18**(12), 717–721.

Lynch, N. A. (1996), *Distributed Algorithms*, Morgan Kaufmann.

Lynch, N. A. & Vaandrager, F. W. (1995), 'Forward and backward simulations – Part I: Untimed systems.', *Information and Computation* **121**(2), 214–233.

Michael, M. & Scott, M. (1998), 'Nonblocking algorithms and preemption safe locking on multiprogrammed shared memory multiprocessors', *Journal of Parallel and Distributed Computing* **51**(1), 1–26.

Moir, M., Nussbaum, D., Shalev, O. & Shavit, N. (2005), Using elimination to implement scalable and lock-free fifo queues, *in* 'Proc. 17th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA 2005)', ACM Press, Las Vegas, Nevada, USA, pp. 253–262.

Sasturkar, A., Agarwal, R., Wang, L. & Stoller, S. D. (2005), Automated type-based analysis of data races and atomicity, *in* 'PPoPP '05: Proceedings of the tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming', ACM Press, New York, NY, USA, pp. 83–94.

Shann, C.-H., Huang, T.-L. & Chen, C. (2000), A practical nonblocking queue algorithm using compare-and-swap, *in* 'Seventh International Conference on Parallel and Distributed Systems (ICPADS'00)', pp. 470–475.

Stepney, S., Cooper, D. & Woodcock, J. (1998), More powerful Z data refinement: pushing the state of the art in industrial refinement, *in* J. P. Bowen, A. Fett & M. G. Hinchey, eds, 'The Z Formal Specification Notation, 11th International Conference of Z Users, Berlin, Germany, September 1998', Vol. 1493 of *LNCS*, Springer, pp. 284–307.

Treiber, R. K. (1986), Systems Programming: Coping with Parallelism. RJ5118, Technical report, IBM Almaden Research Center.

Wang, L. & Stoller, S. D. (2005), Static analysis of atomicity for programs with non-blocking synchronization, *in* 'PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming', ACM Press, New York, NY, USA, pp. 61–71.

Yahav, E. & Sagiv, M. (2003), Automatically verifying concurrent queue algorithms, *in* B. Cook, S. Stoller & W. Visser, eds, 'Electronic Notes in Theoretical Computer Science', Vol. 89, Elsevier.