Journal of Universal Computer Science, vol. 15, no. 1 (2009), 72-111
submitted: 27/3/08, accepted: 3/9/08, appeared: 1/1/09 © J.UCS

Reasoning about Nonblocking Concurrency

Lindsay Groves
(School of Mathematics, Statistics and Computer Science
Victoria University of Wellington, New Zealand
lindsay.groves@vuw.ac.nz)

Abstract: Verification of concurrent algorithms has been the focus of much research over a considerable period of time, and a variety of techniques have been developed that are suited to particular classes of algorithm, for example algorithms based on message passing or mutual exclusion. The development of *nonblocking* or *lock-free* algorithms, which rely only on hardware primitives such as Compare And Swap, present new challenges for verification, as they allow greater levels of currency and more complex interactions between processes.

In this paper, we describe and compare two approaches to reasoning about nonblocking algorithms. We give a brief overview of the *simulation* approach we have used in previous work. We then give a more detailed description of an approach based on Lipton's *reduction* method, and illustrate it by verifying two versions of a shared counter and two versions of a shared stack. Both approaches work by transforming a concurrent execution into an equivalent sequential execution, but they differ in the way that executions are transformed and the way that transformations are justified.

Key Words: concurrency, shared memory, nonblocking algorithms, lock-free algorithms, verification, linearisability, atomicity, simulation relation, reduction
Category: D.1.3, D.2.4, F.3.1

1 Introduction

Shared memory concurrency is becoming more and more important, as programming languages such as Java and C# make multi-threading accessible to applications programmers, and programmers develop software architectures that depend on concurrency. This trend promises to continue with the development of multi-core processors, and the realisation that further increases in speed will only be gained through parallelism. Increased use of concurrency has in turn highlighted the problems associated with traditional approaches to designing concurrent programs, based on mutual exclusion, which are prone to problems such as deadlock, priority inversion and convoying.

To avoid these problems, researchers have sought to develop concurrent algorithms which do not rely on mutual exclusion, or associated mechanisms such as locks and semaphores, but instead are designed to work correctly in the presence of interference [Shavit and Moir, 2004; Fraser and Harris, 2007]. These algorithms use intricate mechanisms to obtain good performance under different workloads, posing significant challenges to ensure correct behaviour. In this paper, we describe and compare two approaches to verifying nonblocking algorithms: one based on simulation relations between labelled transitions systems; the other based on Lipton's reduction method.

The simulation approach has been studied in a variety of contexts [Jifeng et al., 1986; Lynch and Vaandrager, 1995; de Roever and Engelhardt, 1998], and we have used it in joint work to verify several nonblocking algorithms [Doherty et al., 2004b; Doherty et al., 2004a; Colvin and Groves, 2005; Colvin et al., 2005; Colvin et al., 2006; Colvin and Groves, 2007]. While this approach has been successful, the resulting proofs are hard to understand and to describe in an accessible way, primarily because the proof obligations obtained are hard to relate to the pseudo-code form of the algorithm and involve reasoning about one atomic step of the algorithm at a time.

Lipton's reduction method was developed for reasoning about deadlock freedom in concurrent systems based on locks [Lipton, 1975], and has been extended to handle other safety and liveness properties [Lamport and Schneider, 1989; Cohen and Lamport, 1998], and to handle systems based on message passing [Lamport, 1990]. In recent work, we have investigated how this approach can be extended to verify nonblocking algorithms [Groves, 2007; Groves, 2008]. This approach is attractive because it allows us to separate reasoning about the correctness of the implementation in the absence of interference from reasoning about the effect of interference. We do this by showing that every concurrent execution is equivalent to one in which all operations on the shared object are executed without interruption in a way that preserves the order of non-concurrent operations, and that this execution correctly implements the abstract operations.

This paper aims, firstly, to give an overview of these two approaches and compared them, and secondly, to explain how we have extended the reduction method to handle nonblocking algorithms and show how it can be used to explain the correctness, or incorrectness, of some nonblocking algorithms. We begin in Section 2 by introducing the class of nonblocking algorithms, explaining the strong synchronisation primitives (such as CAS) they rely on, and discussing some aspects of the design of such algorithms. In Section 3, we introduce *linearisability*, the standard correctness criterion for shared objects, and discuss how linearisability can be proved using simulation. In Section 4, we introduce the notion of *atomicity*, show how it can be used to prove linearisability, and discuss how we can prove atomicity by building on Lipton's *reduction* method. In Sections 5 and 6, we apply our version of reduction to two forms of shared counter, and to a shared stack — the latter showing how issues relating to dynamic memory are handled. Finally, in Section 7, we summarise our results, and make some observations about related and future work.

2 Nonblocking algorithms and strong synchronisation primitives

We are concerned with concurrent programs in which multiple processes (or threads — at this level of abstraction, there is no need to distinguish between processes and threads) communicate via shared data structures, generally called shared objects. Processes may run on separate processors, or share a single processor, and may run at different speeds — the correctness of the program should not depend upon assumptions about the number or speeds of processors, or about scheduling.

The fundamental problem encountered in designing such programs is how to deal with the interference that arises when multiple processes try to access a shared object at the same time. In particular, we need to avoid the scenario where a process reads a shared object, computes a new value, and stores the new value into the shared object, but the new value is now invalid because another process has updated the shared object in the interim.

The traditional approach to this problem is to avoid interference between processes by using locks, or similar mechanisms such as semaphores or monitors, to ensure that shared objects are updated under mutual exclusion. To avoid the scenario outlined above, a *lock* is associated with each shared object. A process wanting to update a shared object must first acquire the lock for that object, and then holds the lock until it has completed its update. Any other process wanting to update the object must wait until the first process has completed its update in order to acquire the lock. This approach is called *blocking*, because processes wishing to access a shared object while another process holds the lock are forced to wait until the lock is released. The use of locks has several inherent problems which make it undesirable. For example, if a process dies while holding a lock, the entire system may deadlock, and a slow process holding a lock causes all other processes needing that lock to be blocked. These problems becomes more acute as the number of processes increases, and lock-based solutions are thus regarded as *non-scalable*.

Nonblocking algorithms, in which no process is ever forced to wait for another to complete an operation, have been developed as an alternative approach aimed at achieving scalable designs (e.g. [Treiber, 1986; Michael and Scott, 1996; Michael and Scott, 1998; Shann et al., 2000; Hendler et al., 2004; Moir et al., 2005]). These algorithms are designed to work correctly in the presence of interference from other processes, rather than to avoid it by using locks. Typically, to avoid the scenario outlined above, a strong synchronisation primitive such as LL/SC or CAS (see Section 2.1) is used to perform an update provided that no interference has occurred. When interference is detected, the operation is usually attempted again, however, in some cases a process will "help" a slower process by completing part of its operation (e.g. [Michael and Scott, 1996; Michael and

Scott, 1998; Shavit and Zemach, 1999; Shann et al., 2000]).

Different classes of nonblocking algorithms have been identified, according to the progress conditions that they guarantee. *Wait-free* algorithms [Herlihy, 1991] ensure that every operation completes within a finite number of its own steps. While this is clearly a desirable property, very few efficient wait-free algorithms are known for implementing common shared objects such as stacks and queues. *Lock-free* algorithms ensure that some operation will always complete within a finite number of steps of the system. This weaker condition ensures that the system as a whole makes progress even though individual operations may not terminate, and efficient lock-free algorithms are known for many common data structures [Shavit and Moir, 2004]. Lock-free algorithms are guaranteed to be free from deadlock and livelock; wait-free algorithms, in addition, are free from individual starvation.

The terms nonblocking and lock-free are often used interchangeably. We use nonblocking as a general term to describe algorithms that do not rely on mutual exclusion, and lock-free as a more specific term to describe algorithms that satisfy this progress condition.

2.1 Strong synchronisation primitives

These nonblocking progress conditions preclude the use of locks, and nonblocking algorithms typically rely instead on the use of strong synchronisation primitives such as Load Linked/Store Conditional (LL/SC) or Compare and Swap (CAS), which allow a process to atomically update a shared location only if no harmful interference has occurred. These primitives differ in the kind of interference that can be detected.

A *Load Linked* instruction, $LL(loc)$, reads the contents of a shared location, *loc*. A subsequent *Store Conditional* instruction, $SC(loc, new)$, stores a new value, *new*, into *loc* and succeeds (returning *true*) if no SC has been performed on *loc* since the last LL on *loc* performed by that process, and otherwise fails (returning *false* and leaving *loc* unchanged). Thus, a process can load the value in *loc* using LL, compute a new value, and update *loc* by storing the new value only if *loc* has not changed since that process performed its LL.

A *Compare And Swap* instruction, $CAS(loc, old, new)$, tests whether a shared location *loc* contains the value *old* and if so replaces the value by *new* and succeeds (returning *true*); otherwise it fails (returning *false* and leaving *loc* unchanged). Thus, a process can load the value in a shared location, using an ordinary load, compute a new value, and update the location by storing the new value only if the location contains the same value that it had when that process performed its load. Some versions of CAS store the current value of *loc* in *old* when the comparison fails, or return the current value of *loc* rather than a Boolean; the version described above is sometimes (and more aptly) called *Compare And Set*.

The key difference between LL/SC and CAS is that SC checks whether the shared variable has been updated since the corresponding LL was performed (assuming that all updates on the location are performed using SC), whereas CAS only checks whether the location has the same value that it was previously observed to have. Thus, CAS cannot detect a situation where a shared location is seen to have a value, say A, its value is then changed to some other value, say B, and then back to A again. There are some cases where the check provided by CAS is appropriate, and even preferable to that provided by LL/SC, and other cases where the check provided by LL/SC is required. We will see examples of both in later sections.

Unfortunately, implementations of LL/SC on current hardware are very limited — for example, it may only be possible for one LL to be active at a time and an SC may be allowed to fail spuriously in a way that prevents progress from being guaranteed. CAS, on the other hand, is available on most modern architectures, and is now supported by the Java Virtual Machine.

Herlihy [Herlihy, 1991] showed that any sequential data structure can be turned into a wait-free, or lock-free, implementation using either LL/SC or CAS, and that this is not so for other instructions such as *Fetch And Add* which can sometimes be used to implement concurrent algorithms. Unfortunately, the resulting algorithms are typically very inefficient, and considerable ingenuity is required to obtain efficient and scalable nonblocking algorithms.

2.2 Example

As a simple example, consider a shared counter, with a single operation which increments the counter and returns its new value. This can be implemented using a module in which the shared counter, c, is encapsulated (so it cannot be accessed by any other code) and initialised to 0. The module exports the procedure *inc*, which can be defined (using a Pascal-like pseudo-code) using either LL/SC or CAS, as shown in Fig. 1.

In both versions, we read the shared counter (c) into a local variable (a) and use it to compute the new value ($a + 1$) which is then stored in another local variable (b). We then attempt to update the shared variable using an SC or CAS. In this case, the fact that c has the same value it had when it was read into a is sufficient to ensure that setting c to b will give the correct result, so both versions will increment the counter correctly. Moreover, if we ignore the possibility of wraparound, the two versions will retry in exactly the same situations — since the counter can only be incremented, there is no way that the counter can be modified and then return to its previous value, so the SC

```
$inc(\textbf{out}\ r : nat) \mathrel{\widehat=}$
  var $a, b : nat$;
  repeat
    $a := LL(c)$;
    $b := a + 1$
  until $SC(c, b)$;
  $r := b$
```

```
$inc(\textbf{out}\ r : nat) \mathrel{\widehat=}$
  var $a, b : nat$;
  repeat
    $a := c$;
    $b := a + 1$
  until $CAS(c, a, b)$;
  $r := b$
```

Figure 1: Code for *inc* using LL/SC and CAS

and CAS will succeed (and fail) for exactly the same interleavings with other processes.

This algorithm is not wait-free, since it is possible for a process performing an *inc* operation to continually experience interference, and so never complete its operation. The algorithm is lock-free, assuming that we have a finite number of processes, since some process will always complete an *inc* operation within a finite number of steps of the counter implementation. This is because the CAS (or SC) in one process can only fail if another process performs a successful CAS (or SC), and since there are a finite number of processes, the counter implementation cannot take an infinite number of steps without completing an operation.[1] We will not discuss progress conditions any further in this paper.

2.3 Designing nonblocking algorithms

The code in Fig. 1 illustrates a common pattern found in many simple lock-free algorithms, where we use a loop to repeatedly attempt an operation on a shared object until the operation is able to be completed without interference. Within the loop body, we take a snapshot (in this case, a), of the shared location to be updated (c), and use it to compute the new value (b). We then attempt to update the shared variable, using an SC or CAS to ensure that the update is safe.

In the shared counter example, the value of the variable being updated is the entire shared object, so the guarantee provided by CAS (i.e. that the current value is equal to the snapshot) is sufficient to ensure that the correct update is made. Indeed, if a module implementing an integer-valued counter provided

[1] If there were an infinite number of processes, it would be possible for the counter implementation to take an infinite number of steps in which an infinite number of processes each took one or two steps but did not perform its CAS, and so no operation would complete.

both an *inc* operation, as above, and a *dec* operation to decrement the counter (which would be identical to *inc* with the assignment to b replaced by $b := a-1$), the LL/SC and CAS versions would both behave correctly. However, the LL/SC semantics would disallow perfectly acceptable executions, in which one process reads the value of c, then another process completes an *inc* and a *dec* before the first process performs its SC: the SC would fail and force the process to retry its operation, even though it would have assigned c the correct value. In this situation, a CAS would succeed, and so in this case, a CAS is preferable to LL/SC. This example demonstrates that not all interference is harmful.

In modifying a larger data structure, we have to consider carefully what updates need to be made and when they are safe. A key issue in designing nonblocking algorithms is to identify a small enough part of the state to use as the snapshot so that it can be conditionally updated using a CAS or SC, which usually operates on a single or double word, and to determine when such an update is safe. Ideally, we would like to be able to construct any new values that are required using local variables, and heap locations to which the process has the only pointer, and then update a single shared location. We then wish to be able to infer from a successful SC or CAS that the update is safe. In some cases, knowing that the shared variable has the same value as the snapshot is sufficient and we can use a CAS, as in the shared counter example discussed in Section 2.2, but in others we need to know that the shared variable has not changed since the snapshot was taken so the stronger guarantee provided by LL/SC is needed.

Because of the limitations of current hardware mentioned in Section 2.1, LL/SC tend not to be used in practical nonblocking algorithms, and CAS is more widely used even though LL/SC semantics is often more appropriate. There has been recent work on synthesising a more practical LL/SC using CAS instructions [Doherty et al., 2004c; Michael, 2004], but most practical nonblocking algorithms still use CAS. We thus concentrate on algorithms using CAS instructions, though our techniques could easily be applied to LL/SC as well (cf. [Wang and Stoller, 2005]).

Things become more difficult if more than one shared location needs to be updated. A common solution in this case is to place the part of the state that needs to be updated into a dynamically allocated record, perform the required updates on a newly allocated record and then update the global state by assigning a global pointer variable to point to this record. We will see an example of this in Section 6, and discuss the problems that arise in deciding when the update is safe.

In more sophisticated algorithms, the implementation may not retry the operation immediately. For example, it may choose to delay before retrying, as part of a back-off scheme, or may try some other way of achieving its purpose, such as finding another process attempting to perform a complementary operation [Hendler et al., 2004; Moir et al., 2005]. If the object interface provides operations that may fail when interference is detected, the caller may choose to perform some other application task before retrying the operation.

3 Linearisability

We now consider what it means to say that an implementation of a concurrent data structure, such as a shared counter, is correct. Since the data structure may change many times between when a process begins an operation and when it completes that operation (if, indeed, it does), we can't use the standard notions of pre- and postconditions applying to the state before and after execution of an operation. Instead, we take the view that an implementation of a concurrent data structure is correct if each operation performed by any process appears, to other processes, as though it occurred atomically at some point during its execution, and that the sequence of operations thus obtained respect the sequential semantics for the data type.

This property is called *linearisability* [Herlihy and Wing, 1990], and is the standard safety condition for shared objects; the point at which an operation appears to occur is called its *linearisation point*. The requirement that operations appear to occur instantaneously means that in reasoning about programs that use shared objects, we can ignore the effects of concurrency within the implementation of the objects and treat these operations as though they were atomic. The requirement that the linearisation point is within the operation's execution (technically, between its invocation and its response) ensures that the order of non-concurrent operations is preserved.

The notion of linearisability is formalised in [Herlihy and Wing, 1990] in terms of histories. A *history* for a shared object $\mathcal{O}$ is a sequence of events corresponding to invocations and responses of operations in some execution of a program operating on $\mathcal{O}$. For each operation *op* that can be applied to $\mathcal{O}$, we write $op\_inv_p(args)$ to denote process p invoking operation *op* with arguments *args*, and $op\_resp_p(res)$ to denote process p returning response *resp* with result *res*. An execution is modelled by a history in which the first action of each execution of an operation *op* with arguments *args* by a process p is replaced by $op\_inv_p(args)$, the last action of each completed execution of operation *op* with result *res* by process p is replaced by $op\_resp_p(res)$, and all other actions are deleted. Two histories are *equivalent* if, in both histories, each process performs the same sequence of invocations and responses.

A response *matches* a preceding invocation by the same process, provided there are no intervening events involving that process. An invocation and its matching response represent a *completed* operation; an unmatched invocation represents an uncompleted or *pending* operation. $complete(H)$ is the maximal subsequence of history H consisting only of invocations and matching responses.

A *sequential history* is a sequence of alternating invocations and responses, possibly ending with an unmatched invocation, in which each response matches the preceding invocation. A *sequential specification* for $\mathcal{O}$ is a prefix-closed set of histories for $\mathcal{O}$. A sequential history H is *legal* if it belongs to the sequential specification for $\mathcal{O}$.

A history H induces a partial order, $<_H$, on completed operations such that $op <_H op'$ if the response for op occurs in H before the invocation for op'. Operations not related by $<_H$ are concurrent. H is sequential iff $<_H$ is a total order.

A history H is *linearisable* if it can be extended, by adding zero or more response events, to give a history H' such that $complete(H')$, is equivalent to some legal sequential history S, called a *linearisation* of H, with $<_{H'} \subseteq <_S$. A shared object $\mathcal{O}$ is *linearisable* if every history for $\mathcal{O}$ is linearisable with respect to the sequential specification for $\mathcal{O}$.

The definition in [Herlihy and Wing, 1990] is generalised in a straightforward way to apply to a system acting upon a set of shared objects. It is sufficient for our purposes to consider a single shared object.

3.1 Example

Consider the shared counter described in Section 2.2. There is only one operation, *inc*, which takes no arguments, and it only ever returns one kind of response, which returns an integer result. We will thus denote an invocation of *inc* by process p as inv_inc_p, and its responses returning result c as $inv\_ok_p(c)$.

The sequential specification for a shared counter contains alternating occurrences of *inv_inc* and *inv_ok* by the same process, where successive occurrences of *inv_ok* return contiguous values. Thus, for all processes p and q, the sequential specification contains the histories: $\langle\rangle$, $\langle inv\_inc_p\rangle$, $\langle inv\_inc_p, inv\_ok_p(0)\rangle$, $\langle inv\_inc_p, inv\_ok_p(0), inv\_inc_q\rangle$, $\langle inv\_inc_p, inv\_ok_p(0), inv\_inc_q, inv\_ok_q(1)\rangle$, etc.

Now, consider a program with three processes operating on a shared counter. Fig. 2 shows the part of a possible execution of this program, showing only the actions involved in executing *inc*. For convenience, actions are arranged in columns corresponding to processes, but they should be understood as comprising a single sequence of actions (one action per row), as shown in the first column of Fig. 3. We write $^+$ or $^-$ beside a CAS to indicate whether it succeeded or failed (likewise later, we will write B^+ or B^- to indicate that a test B succeeded or failed, respectively). The effect of an action performed by a given process can be understood by adding the process number as a subscript on each local variable referenced; for example, in Fig. 2, $(a := c)_1$ becomes $a_1 := c$ and $CAS(c, a, b)_2^+$ becomes $CAS(c, a_2, b_2)^+$.

This execution corresponds to the following concurrent history:

Step	Process 1	Process 2	Process 3
1	$(a := c)_1$		
2		$(a := c)_2$	
3		$(b := a + 1)_2$	
4		$CAS(c, a, b)_2^+$	
5			$(a := c)_3$
6			$(b := a + 1)_3$
7	$(b := a + 1)_1$		
8			$CAS(c, a, b)_3^+$
9			$(r := b)_3$
10	$CAS(c, a, b)_1^-$		
11		$(r := b)_2$	
12		$(a := c)_2$	
13	$(a := c)_1$		
14		$(b := a + 1)_2$	
15	$(b := a + 1)_1$		
16	$CAS(c, a, b)_1^+$		
17		$CAS(c, a, b)_2^-$	
18	$(r := b)_1$		

Figure 2: Execution for shared counter

$$H = \langle \mathit{inc\_inv}_1, \mathit{inc\_inv}_2, \mathit{inc\_inv}_3, \mathit{inc\_ok}_3(2), \mathit{inc\_ok}_2(1), \mathit{inc\_inv}_2, \mathit{inc\_ok}_1(3)\rangle$$

All executions of *inc* in this history are completed, except for process 2's second *inc*, which is pending. This pending operation has not yet "taken effect", so in constructing a linearisation, we do not add a response for it, and take $H' = H$. Thus, we have:

$$\mathit{complete}(H') = \langle \mathit{inc\_inv}_1, \mathit{inc\_inv}_2, \mathit{inc\_ok}_2(1), \mathit{inc\_inv}_3, \mathit{inc\_ok}_3(2), \mathit{inc\_ok}_1(3)\rangle$$

We obtain an equivalent sequential history by taking the successful CAS to be the linearisation point for each completed *inc* operation, which gives us:

$$S = \langle \mathit{inc\_inv}_2, \mathit{inc\_ok}_2(1), \mathit{inc\_inv}_3, \mathit{inc\_ok}_3(2), \mathit{inc\_inv}_1, \mathit{inc\_ok}_1(3)\rangle$$

Now, the partial order induced by H' requires that the first operation of process 2 precedes process 3's operation, and this order is respected by the above sequential history. Thus, this particular execution is linearisable.

3.2 Proving linearisability

We can show that a shared object $\mathcal{O}$ is linearisable by showing how to translate an arbitrary execution into a corresponding legal sequential history. In previous collaborative work (see [Doherty, 2003; Doherty et al., 2004b; Colvin et al., 2005; Colvin and Groves, 2005; Colvin et al., 2006; Colvin and Groves, 2007]), we have proved linearisability using simulation techniques [Lynch and Vaandrager, 1995] to transform any concurrent execution of a set of processes operating on a shared object into an equivalent augmented history, which is trivially linearisable.

Given a history H for a shared object $\mathcal{O}$, an *augmented history* [III and Scott, 2004] is a sequence H' containing invocations and responses for operations on $\mathcal{O}$, along with *linearisation events*, written do_op_p, such that: (i) H is identical to H' with all linearisation events deleted, and (ii) replacing every subhistory of the form:[2]

$$op\_inv_p(args)\ \phi\ do\_op_p\ \psi\ op\_resp_p$$

where ϕ and ψ are any sequences of actions not involving p, by:

$$\phi\ op\_inv_p(args)\ op\_resp_p\ \psi$$

gives a legal sequential history which is a linearisation of H. It follows from this definition that if we can construct an augmented history corresponding to any concurrent execution, then $\mathcal{O}$ is linearisable.

The implementation of an object $\mathcal{O}$ is modelled using a simplified form of input/output automaton or IOA [Lynch, 1996; Lynch and Vaandrager, 1995] (essentially a labelled transition system whose actions are partitioned into internal and external actions), which generates *augmented executions*. These are possible concurrent executions of a set of processes operating on $\mathcal{O}$, with invocation and response actions inserted before and after the first and last action of each execution of an abstract operation. The specification is modelled by a similar transition system, which generates augmented histories. In its basic form, the simulation technique shows how to construct an equivalent augmented history by stepping through an arbitrary augmented execution, one step of the implementation IOA at a time, directing the specification IOA to take steps that force it to generate the required augmented history.

An alternative approach would be for the implementation IOA to generate executions without adding invocations and responses, and for the simulation to match the first action of the operation ($a := c$, in the shared counter example) with an invocation in the augmented history and the final action of the operation

[2] In describing histories and executions, we often omit sequence brackets and other punctuation (commas and concatenation operators) and use juxtaposition to combine both sequences and individual elements.

$(r := b)$ with a response, as is done in [Derrick et al., 2008]. However, this requires special treatment if the first or last action of an operation occurs in a loop, as in the case of *pop* for a shared stack (see Section 6).

3.3 Example

For the execution shown in Fig. 2, the implementation IOA would insert an invocation, inc_inv_p, before the first action $(a := c)$ performed by each operation that is begun, and a response, $inc\_ok_p(r)$, following the last action $(r := b)$ of each operation completed. These invocation and response actions inserted by the execution IOA need not occur immediately before and after the first and last actions performed by the operation; it is only necessary that an invocation occurs after the response for the previous operation executed by the same process (if any) and that a response occurs before the invocation for the next operation executed by the same process (if any). Thus, for instance, the invocations inc_inv_1, inc_inv_2 and inc_inv_3 could occur in any order before the first action shown in Fig. 2.2. The second column of Fig. 3 shows one possible augmented execution for the execution in Fig. 2.

In constructing the equivalent augmented history, invocations and responses are copied as is, and a linearisation event, do_inc_p, is inserted each time the execution performs a successful CAS, since this is the linearisation point at which the effect of the operation becomes visible to other processes. All other actions of the augmented execution are discarded. Thus, the augmented history constructed from the execution in Fig. 2 might be:

$$inc\_inv_1\ inc\_inv_2\ do\_inc_2\ inc\_inv_3\ do\_inc_3\ inc\_ok_2(1)\ inc\_inv_2$$
$$do\_inc_1\ inc\_ok_1(3)\ inc\_ok_3(2)$$

as shown in the third column of Fig. 3.

Applying the construction in condition (ii) of the definition of augmented history gives the same linearisation as shown in Section 3.1 (see fourth column of Fig. 3).

3.4 Applying simulation

In our simulation proofs, each action in an augmented execution generated by the implementation IOA is either an invocation or response, which must be matched by an identical action in the augmented history generated by the specification IOA, or a primitive action of the implementation. Usually, one such implementation action will correspond to the linearisation event in the augmented history, and must be shown to preserve an *abstraction relation* relating the concrete data structure to the abstract value of $\mathcal{O}$. All other actions are internal, and thus unobservable, so do not correspond to any action of the specification IOA, but must

Concurrent execution	Augmented execution	Augmented history	Sequential history
	$\mathit{inc\_inv}_1$	$\mathit{inc\_inv}_1$	
	$\mathit{inc\_inv}_2$	$\mathit{inc\_inv}_2$	
$(a := c)_1$	$(a := c)_1$		
$(a := c)_2$	$(a := c)_2$		
$(b := a + 1)_2$	$(b := a + 1)_2$		
$CAS(c, a, b)_2^+$	$CAS(c, a, b)_2^+$	$\mathit{do\_inc}_2$	$\mathit{inc\_inv}_2$ $\mathit{inc\_ok}_2(1)$
	$\mathit{inc\_inv}_3$	$\mathit{inc\_inv}_3$	
$(a := c)_3$	$(a := c)_3$		
$(b := a + 1)_3$	$(b := a + 1)_3$		
$(b := a + 1)_1$	$(b := a + 1)_1$		
$CAS(c, a, b)_3^+$	$CAS(c, a, b)_3^+$	$\mathit{do\_inc}_3$	$\mathit{inc\_inv}_3$ $\mathit{inc\_ok}_3(2)$
$(r := b)_3$	$(r := b)_3$		
$CAS(c, a, b)_1^-$	$CAS(c, a, b)_1^-$		
$(r := b)_2$	$(r := b)_2$		
	$\mathit{inc\_ok}_2(1)$	$\mathit{inc\_ok}_2(1)$	
	$\mathit{inc\_inv}_2$	$\mathit{inc\_inv}_2$	
$(a := c)_2$	$(a := c)_2$		
$(a := c)_1$	$(a := c)_1$		
$(b := a + 1)_2$	$(b := a + 1)_2$		
$(b := a + 1)_1$	$(b := a + 1)_1$		
$CAS(c, a, b)_1^+$	$CAS(c, a, b)_1^+$	$\mathit{do\_inc}_1$	$\mathit{inc\_inv}_1$ $\mathit{inc\_ok}_1(3)$
$CAS(c, a, b)_2^-$	$CAS(c, a, b)_2^-$		
$(r := b)_1$	$(r := b)_1$		
	$\mathit{inc\_ok}_1(3)$	$\mathit{inc\_ok}_1(3)$	
	$\mathit{inc\_ok}_3(2)$	$\mathit{inc\_ok}_3(2)$	

Figure 3: Simulation for shared counter

also be shown to preserve the abstraction relation (in this case showing that the abstract value does not change), along with various invariant properties that are required to ensure that only legal executions (and thus histories) are generated and to support the proof that the abstraction relation is preserved.

While this approach has proved to be effective, and is amenable to mechanisation (we have mechanised our proofs using PVS), a large proportion of the effort is devoted to verifying the proof obligations for implementation actions that are not linearisation points. Many of these are only required to ensure that executions are well formed, while others are required to show that they preserve various invariants required for the verification of the few interesting cases which correspond to linearisation points.

Whereas, in the shared counter example, we could construct the required augmented history using *forward simulation*, i.e. by stepping forwards through the augmented execution, it is sometimes necessary to use *backward simulation* [Jifeng et al., 1986; Lynch and Vaandrager, 1995; de Roever and Engelhardt, 1998], in which we step backwards through the augmented execution. This is typically required because we cannot always tell, when an action is executed, whether it is a linearisation point or not. Sometimes we need (or choose, for convenience) to combine forward and backward simulation, so the augmented execution is translated into an intermediate history using forward simulation, and the intermediate history is translated into the required augmented history using backward simulation. Since backward simulation tends to be less intuitive, and more complex to verify, we try to make the intermediate history as close to the augmented history as we can so that more of the work is done in the simpler forward simulation (see [Doherty, 2003; Doherty et al., 2004b; Colvin and Groves, 2005; Colvin et al., 2006]).

There are also further subtleties in the identification of linearisation points. While in many cases the linearisation point for an operation is an action of that operation which updates a shared variable, this is not always the case. Firstly, if the operation does not change the data structure (for example, an operation to test whether a shared counter has a particular value, test whether a stack or queue is empty, or whether a set contains a particular value, or an operation that fails, such as a pop on an empty stack), the linearisation point is an action that read the part of the data structure from which its outcome is determined. Secondly, the linearisation point may be an action performed by another process, and a step of one process may be the linearisation point for an arbitrary number of operations performed by other processes. For example, in the elimination stack algorithm [Hendler et al., 2004; Colvin and Groves, 2007], in which *push* and *pop* operations can be paired off and "eliminated" without altering the stack data structure, the linearisation point for an eliminated operation is a step performed by another process which selects it as an elimination partner. And in the lazy list algorithm [Heller et al., 2005; Colvin et al., 2006], which provides a highly concurrent linked list set implementation, one step of an operation which deletes an element from the set can be the linearisation point for any number of operations by other processes which are testing whether that element is in the set.

4 Atomicity

We will now investigate an alternative approach in which linearisability is proved in two steps: first, we show that every concurrent execution of the implementation is equivalent to a *serial execution*, in which each operation is executed without interruption; and second, we show that when executed without interruption the implementation of each operation satisfies its sequential specification.

The property established in the first step, that every concurrent execution of the implementation is equivalent to one in which each operation is executed without interruption, is called *atomicity*.[3] The second step is equivalent to proving that the abstract specification is satisfied by a sequential implementation.

It is easy to see that atomicity plus sequential correctness implies linearisability. A serial execution is a concatenation of several sequences of actions, each comprising the execution of one abstract operation by some process. Constructing a history from such an execution, as described in Section 3 will result in each such sequence being replaced by an invocation-response pair. The resulting history will already be a legal sequential history and thus provide the linearisation required by the definition of linearisability. Since this history is already sequential, the partial order it induces is already a total order.

Separating these steps allows us to address concurrent and sequential aspects of the implementation separately. In particular, showing that the abstract specification is satisfied can be addressed using standard sequential reasoning techniques, and we may choose to take a lightweight approach, using techniques such as code inspection, testing or model checking, rather than full formal verification. As we will see, we can also use a combination of static analysis, model checking and deductive verification in verifying atomicity.

In the rest of this paper, we will assume that sequential reasoning is well understood. We therefore focus on showing that a concurrent (and usually nonblocking) implementation of a shared data structure is atomic, and only reason informally about sequential correctness of these implementations.

4.1 Example

In the shared counter example, we might show that the concurrent execution shown in Fig. 2.2 is equivalent to the serial execution shown in Fig. 4. Notice that we have deleted the steps of the pending operation of process 2, and that each process otherwise performs the same steps in the same order. It is easy to see that the effects of the first two operations (by processes 2 and 3) are both

[3] Although *atomicity* is widely used with this meaning (e.g. [Lamport and Schneider, 1989; Flanagan and Qadeer, 2003; Sasturkar et al., 2005]), it is also sometimes equated with linearisability (e.g. [Lynch, 1996; Hesselink, 2002]), so care must be taken when referring to different works.

equivalent to $c := c + 1$, and thus correctly implement the *inc* operation. The effect of the operation by process 1 is not quite so straightforward, and we will show how this is handled in Section 5.

Step	Process 1	Process 2	Process 3
1		$a_2 := c$	
2		$b_2 := a_2 + 1$	
3		$CAS(c, a_2, b_2)^+$	
4		$r_2 := b_2$	
5			$a_3 := c$
6			$b_3 := a_3 + 1$
7			$CAS(c, a_3, b_3)^+$
8			$r_3 := b_3$
9	$a_1 := c$		
10	$b_1 := a_1 + 1$		
11	$CAS(c, a_1, b_1)^-$		
12	$a_1 := c$		
13	$b_1 := a_1 + 1$		
14	$CAS(c, a_1, b_1)^+$		
15	$r_1 := b_1$		

Figure 4: Execution for shared counter

Using a similar construction to that outlined above, we can see that this serial execution is represented by the sequential history:

$$inc\_inv_2\ inc\_ok_2(1)\ inc\_inv_3\ inc\_ok_3(2)\ inc\_inv_1\ inc\_ok_1(3)$$

which is the same as the sequential history S constructed in Section 3.1. Thus, we can show that a shared object is linearisable by showing that every concurrent execution is equivalent to a serial execution, and that such executions correctly implement the abstract operations.

4.2 Lipton's reduction method

We begin by considering the idea of *reduction*, originally described by Lipton [Lipton, 1975], and then show how it can be extended to handle a wider range of algorithms. In later sections, we look at further examples.

The reduction method was first proposed by Lipton [Lipton, 1975] as a way of proving safety properties of concurrent programs using semaphores for synchronisation. The main idea is that in order to establish some property of a concurrent program P containing a statement R, we prove that any execution of P is "equivalent" to one in which R is executed without interruption, and that the desired property holds of the *reduction* of P by R, denoted by P/R, in which R is treated as an atomic action.

To show that any execution of P is equivalent to one in which R is executed without interruption, we show that any execution of P can be transformed into an equivalent execution in which the atomic steps of R are contiguous. This is done by analysing the effects of atomic operations to see how the steps of different processes can be reordered without affecting the result of the execution.

We write $\sigma \xrightarrow{a} \tau$, to mean that execution of action a may take the system from state σ to state τ. An action a is *enabled* in a state σ if there exists a state τ such that $\sigma \xrightarrow{a} \tau$.

For actions a and b, and states σ and τ, we write $\sigma \xrightarrow{ab} \tau$ if there is a state ρ such that $\sigma \xrightarrow{a} \rho$ and $\rho \xrightarrow{b} \tau$. We extend this notation to sequences of actions, and write $\alpha \leq \beta$ to mean that for any reachable states σ and τ, $\sigma \xrightarrow{\alpha} \tau$ implies $\sigma \xrightarrow{\beta} \tau$, and say that α *approximates* β. When $\alpha \leq \beta$ and $\beta \leq \alpha$, we say that α is *equivalent* to β.[4]

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. We can show that for sequences of actions, $\alpha = a_1 \dots a_m$ and $\beta = b_1 \dots b_n$, if $a_i b_j \leq b_j a_i$ for all $i \in 1\,..\,m$ and $j \in 1\,..\,n$, then $\alpha\beta \leq \beta\alpha$.

In the context of a given program, P, an action a is called:

- a *right mover* if it right commutes with every action of every other process, i.e. $a_p x_q \leq x_q a_p$ for all actions x of P and distinct processes p and q,
- a *left mover* if it left commutes with every action of every other process, i.e. $x_q a_p \leq a_p x_q$ for all actions x of P and distinct processes p and q, and
- a *both mover* if it is both a right mover and a left mover.

If an execution of statement R by process p consists of actions $a_1, \dots, a_n$, where for some k, actions $a_1, \dots, a_{k-1}$ are right movers and actions $a_{k+1}, \dots, a_n$ are left movers, then any execution, E, of program P containing this execution of R is equivalent to an execution, E', in which R is executed without interruption. More precisely, if $E = \beta_0\, a_1\, \beta_2\, \dots\, a_n\, \beta_n$, where $\beta_0 \dots \beta_n$ are sequences of actions and $\beta_2 \dots \beta_{n-1}$ contain no p-actions, then $E \leq E'$, where $E' = \beta_0\, \dots\, \beta_{k-1}\, a_1\, \dots\, a_n\, \beta_{k+1}\, \dots\, \beta_n$.

Since Lipton was concerned with synchronisation based on semaphores, his results show when P and V operations on semaphores can be treated as right and left movers, respectively. Lamport and Schneider [Lamport and Schneider, 1989]

[4] Note that, for largely historical reasons, we often talk about executions being equivalent when strictly speaking we only require approximation.

extend the range of safety properties that can be handled. Lamport [Lamport, 1990] extends these results to distributed systems, showing when receive and send actions can be treated as right and left movers, respectively, and Lamport and Cohen [Cohen and Lamport, 1998] extend the method to handle liveness properties. Cohen [Cohen, 2000] presents an algebraic approach to proving reductions, based on omega-algebra. Back [Back, 1993] integrated reduction for terminating operations into the refinement calculus.

The reduction idea has also been used as a basis for tools for verifying atomicity, and related properties, using static analysis [Flanagan and Qadeer, 2003; Flanagan and Freund, 2004; Sasturkar et al., 2005], run-time checking [Wang and Stoller, 2006] and model checking [Hatcliff et al., 2004]. While these tools are necessarily limited in the cases they can handle, and are also prone to false alarms, they can be useful for detecting some common errors, and the underlying theory is applicable to more rigorous verification techniques. For example, some of this work includes techniques for detecting when a process holds a unique pointer to a heap location, in which case operations on that location can be treated as local actions, and when pointers “escape”, i.e. become visible to other processes.

4.3 Applying reduction to nonblocking concurrency

To apply the reduction approach to a shared object, such as a stack or queue, we must show that any execution of a program involving that shared object is equivalent to one in which every operation on the shared object is executed without interruption. Thus, the component R in the above description of the reduction method is always the body of a procedure implementing an abstract operation on that shared object.

We need to consider carefully what we mean by an “equivalent”, or approximating, execution. Lipton’s definition of $\leq$ in terms of identical states is clearly too strong, since we can obviously ignore the effects of some assignments to local variables, and we can ignore the order in which dynamic storage locations are allocated. Ultimately, what we care about is whether two executions produce the same result, and this can be captured in terms of histories. Thus, we adopt the more general definition that $\alpha \leq \beta$ iff for any sequences ϕ and ψ, if $\phi\,\alpha\,\psi$ is a valid execution, then $\phi\,\beta\,\psi$ is also a valid execution and has the same history as $\phi\,\alpha\,\psi$. Of course, if two sequences of actions do lead to identical states, that is still sufficient to prove atomicity.

Since all of the actions of R are grouped at the position of some action a_k, any operation that begins before a_1 (ends after a_n) in E also begins before a_1 (ends after a_n) in E'. Thus, the order of non-current operations is preserved, as required for linearisability, and a_k can be taken as the linearisation point for R.

We assume that the shared object being implemented is encapsulated in some kind of module structure, so the variables used to implement the object can only be accessed via the procedures exported by that module. In the case of algorithms that use dynamic memory, we assume that storage is allocated by a *new* operation which returns a pointer to a piece of storage to which no other pointer exists (in this program or any other program or process running on the same system), and that the only way to create a pointer to a piece of storage is via the *new* operation (so there is no pointer arithmetic or any way of converting a number into a pointer, etc.). Thus, the only concurrency we need to consider is concurrent execution of these procedures by other processes.

To apply the reduction approach to nonblocking concurrency, we need to identify commutativity properties of the concurrency primitives and other actions used. It is easy to see that actions that only affect local variables are both movers. This justifies the common assumption (which we make in our examples) that any test or assignment containing no more than one reference to a shared location can be treated as an atomic action [Lamport and Schneider, 1989]. Because of this result, in the code for *inc* in the shared counter (Fig. 1), we can omit the assignment to b and write the CAS as $CAS(c, a, a+1)$, and replace the final assignment by $r := a + 1$.

More generally, we can see that:

– An action that only accesses local variables, immutable shared variables, and 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 affected by the change to that variable.

The first case is generally easy to apply, and places where this rule can be applied can often be detected using static analysis techniques (e.g. [Flanagan and Qadeer, 2003; Flanagan and Freund, 2004; Sasturkar et al., 2005]).

The second case requires ways of showing that a particular sequence of actions does not alter a shared variable. In Lipton's method, this is done by appealing to properties of locks. In the absence of locks, we have to appeal to other information about the program. Sometimes the outcome of a CAS is sufficient; sometimes we have to appeal to more general behavioural properties of the program, and an interesting aspect of our verifications is to identify the properties that are required and consider how they can be verified.

In most nonblocking algorithms, shared variables are usually updated using CAS (or SC), and updates to shared variables are often linearisation points so we do not need to show that they can be moved. There are, however, some cases where updates to shared variables are not linearisation points (and sometimes not performed using CAS or SC), and in these cases we need to be able to show that these actions can be moved. The interesting issue then is what we mean by saying that an action is “not affected” by a change to a shared variable.

In addition to these issues, we find that we don’t need retain exactly the same execution steps — we may need to delete or perhaps modify some steps — so long as the result is an equivalent (or approximating) sequential execution of the implementation code.

5 Counters

We will now illustrate the reduction approach by considering two variants of the simple shared counter introduced in Section 2.

5.1 A simple counter

First, consider a shared counter, which may be operated on by a finite set of processes $\mathcal{P}$, where the only operation is *inc*, as shown in Section 2.2. A completed execution of *inc* consists of one or more iterations of the loop, followed by an execution of $r := b$. To describe this more precisely, let A, B and R stand for the assignments $a := c$, $b := a + 1$ and $r := b$, and C^+ and C^- stand for a successful and unsuccessful CAS, respectively, performed by process p. Each loop iteration is either a *failed iteration*, consisting of A followed by B followed by C^-, or a *successful iteration*, consisting of A followed by B followed by C^+. So, any completed execution of *inc* by process p consists of zero or more failed iterations of the loop, followed by one successful iteration, followed by a R, i.e. $(A\,B\,C^-)^n\,A\,B\,C^+\,R$, for some $n \geq 0$.

In a program execution containing a completed execution of *inc* by process p, these steps may be interleaved with actions of other processes (indeed, in order for a failed iteration to occur, they must be). So, such an execution is of the following form, where the actions comprising the execution of *inc* are underlined for contrast:

$$\alpha\ \underline{A}\ \beta_1\ \underline{B}\ \gamma_1\ \underline{C^-}\ \delta_1\ \cdots\ \underline{A}\ \beta_n\ \underline{B}\ \gamma_n\ \underline{C^-}\ \delta_n\ \underline{A}\ \beta_{n+1}\ \underline{B}\ \gamma_{n+1}\ \underline{C^+}\ \gamma_{n+1}\ \underline{R}\ \epsilon$$

for some $n \geq 0$, where α, β_i, γ_i, δ_i and ϵ are sequences of atomic actions of *inc*, i.e. sequences over $\{A_q, B_q, R_q, C_q^+, C_q^- \mid q \in \mathcal{P}\}$, where β_i, γ_i and δ_i contain no p-actions and any p-action in α is part of an operation contained entirely within α (this condition ensures that we are reducing all of the actions comprising this execution of *inc*).

We now wish to show that this execution is equivalent to one in which *inc* is executed without interruption. Following Lipton’s approach, we need to identify a step (i.e. a linearisation point) such that all preceding steps are right movers and all subsequent steps are left movers.

In this algorithm, a successful CAS is clearly the linearisation point for an *inc*. We can see this intuitively from the fact that this is where the counter is updated, and thus where the effect of the operation becomes visible to other processes; it is also indicated by the fact that two successful CAS actions will never commute. Thus, we want to show that all steps of this *inc* before the successful CAS are right movers and all steps after it are left movers.

The only step after the successful CAS is R (i.e. $r := a + 1$), which is a both mover, and thus also a left mover, as it only involves local variables. So we have:

$$\gamma_{n+1}\ \underline{R} \leq \underline{R}\ \gamma_{n+1} \tag{1}$$

since R commutes with every action in γ_{n+1}.

Now consider the loop body. The assignment $b := a + 1$ only involves local variables so is a both mover and can be moved right over any actions between it and the next action of process p, which is a CAS. Thus, for $i = 1, \cdots, n + 1$, we have:

$$\underline{B}\ \gamma_i \leq \gamma_i\ \underline{B} \tag{2}$$

since B commutes with every action in γ_i.

The assignment $a := c$ is not so easy because it reads a shared variable. To justify moving it right, we need to show that the steps between it and the next action of process p (which is B) do not alter c — or, at least, that any changes to c do not affect the outcome of the operation.

In the last (successful) iteration of the loop, the successful CAS finds that c has the same value that it had when it was read in the preceding occurrence of A. As noted in Section 2.2, this is sufficient to ensure that the update performed by the CAS is correct. Thus, in this context, A right commutes with $\beta_{n+1}\,\gamma_{n+1}$:

$$\underline{A}\ \beta_{n+1}\ \gamma_{n+1}\ \underline{B}\ \underline{C^+} \leq \beta_{n+1}\ \gamma_{n+1}\ \underline{A}\ \underline{B}\ \underline{C^+} \tag{3}$$

So long as *inc* is the only operation on the shared counter, C^+ is the only action that can modify the counter, and no sequence of actions by other processes can modify the counter and return it to its previous value (ignoring possible wrap-around), so we can infer that no action in $\beta_{n+1}\,\gamma_{n+1}$ changes the counter. If we add a *dec* operation to the shared counter, however, as outlined in Section 2.3, we cannot show that $\beta_{n+1}\,\gamma_{n+1}$ does not change the counter, but the fact that any such change returns the counter to the value it had when the snapshot was taken means that reading c after execution of $\beta_{n+1}\,\gamma_{n+1}$ will lead to the same result. Thus, in this case we argue that A can be moved over all of $\beta_{n+1}\,\gamma_{n+1}$, rather than showing that it can be moved over each action of $\beta_{n+1}\,\gamma_{n+1}$.

Now consider a failed iteration of the loop. We want to move the steps of these iterations right over the steps of other processes (i.e. β_i and γ_i). We can move B right over γ_i, as before. However, in such an iteration, the steps between the A that loads c and the failed CAS must alter c, so we can't move A right over those steps. More precisely, it is not the case that $\underline{A}\,\beta_i\,\gamma_i\,\underline{B}\,\underline{C^-} \leq \beta_i\,\gamma_i\,\underline{A}\,\underline{B}\,\underline{C^-}$, because in the latter, the CAS will not fail, so C^- will not be enabled. If we make this change, we will get a different execution, and possibly a different history, because the CAS will succeed, since $\underline{A}\,\beta_i\,\gamma_i\,\underline{B}\,\underline{C^-} \leq \beta_i\,\gamma_i\,\underline{A}\,\underline{B}\,\underline{C^+}$ does hold. The CAS then becomes the linearisation point, and the rest of the execution of *inc* will not happen.

Similarly, we can't usually move a failed CAS right over steps of other processes, since those steps may alter c, so the value of c tested may change, so it may not be the case that $\underline{C^-}\,\gamma_i \leq \gamma_i\,\underline{C^-}$ (in fact, this does not create a problem in this example, but in general it may).

We can avoid the problem by observing that in constructing the serial execution, we do not have to retain exactly the same steps. So long as the reduced execution is still an execution of the code for that operation and produces the same result, we will obtain the same history. In this case, iterations in which the CAS fails have no observable effect — A and B assign to local variables which are not live at the end of the loop body, since the next accesses to them are assignments in the next iteration (this means that a and b could have been declared as local to the loop body, so each iteration gets different local variables). Thus, we can obtain an equivalent execution by simply deleting them; i.e. for $i = 1, \cdots, n$, we have:

$$\underline{A}\,\beta_i\,\underline{B}\,\gamma_i\,\underline{C^-} \;\leq\; \beta_i\,\gamma_i \tag{4}$$

Combining (1), (2), (3) and (4), with an induction on the number of failed iterations (n), we get:

$$\begin{aligned}&\alpha\,\underline{A}\,\beta_1\,\underline{B}\,\gamma_1\,\underline{C^-}\,\delta_1\,\cdots\,\underline{A}\,\beta_n\,\underline{B}\,\gamma_n\,\underline{C^-}\,\delta_n\,\underline{A}\,\beta_{n+1}\,\underline{B}\,\gamma_{n+1}\,\underline{C^+}\,\gamma_{n+1}\,\underline{R}\,\epsilon\\ &\leq \alpha\,\beta_1\,\gamma_1\,\delta_1\,\cdots\,\beta_n\,\gamma_n\,\delta_n\,\underline{A}\,\underline{B}\,\underline{C^+}\,\underline{R}\,\epsilon\end{aligned}$$

which shows that the implementation of *inc* is atomic.

Thus, we obtain an equivalent execution in which the implementation of *inc* is executed without interruption, in which case the loop will only ever perform one iteration as the CAS will succeed. The observable effect of an *inc* which retries n times is the same as if it did not make the failed attempts, but waited until it was able to execute without interruption. Repeating this transformation for every operation execution will reduce the execution to an equivalent sequential execution.

It is easy to show that this execution correctly implements the abstract operation of incrementing the counter. For example, we can prove this using a simple relational semantics. Let the semantics of an assignment $x := e$ be given by the relation $x' = e$, and $CAS(loc, old, new)$ by $loc = old \wedge loc' = new \vee loc \neq old \wedge loc' = loc$ (we will assume implicitly that any variable not appearing in primed form remains unaltered), then the sequence $A\,B\,C^+\,R$ is equivalent to $(a'_p = c) ⨾ (b'_p = a_p{+}1) ⨾ (c = a_p) ⨾ (r'_p = b_p)$, which implies $c' = c{+}1 \wedge r'_p = c{+}1$.

5.2 A bounded counter

Next, consider a shared bounded counter, which is constrained to be between 0 and N, for some natural number N. To avoid the possibility of an *inc* operation blocking when the counter reaches its bound, we extend the definition of *inc* so that when it would produce a result that is out of range, it leaves the counter unchanged and returns a special value to indicate that this occurred. We write this value as $\perp$ (where $\perp \notin nat$) and change the result type of *inc* to be $nat_\perp = nat \cup \{\perp\}$. To simplify the example, we will exploit the result given in Section 4.3 to omit the assignment to b and replace its remaining occurrences by $a+1$. The resulting code is shown in Fig. 5.

```
$inc$(**out** $r : nat_\perp$) $\widehat{=}$
  **var** $a : nat$;
  **repeat**
      $a := c$;
      **if** $a = N$ **then** $r := \perp$; **return fi**
  **until** $CAS(c, a, a+1)$;
  $r := a + 1$
```

Figure 5: *inc* operation for bounded counter

We now need to consider three kinds of executions of the loop. We have successful and failed iterations, as in the previous example. We also have new kind of iteration, which returns from the operation without altering the shared object. We call this an *exceptional iteration.*

A failed iteration is of the form:

$$(a := c)_p \;\beta\; (a = N)_p^- \;\gamma\; C_p^-$$

As before, the steps of an unsuccessful iteration have no observable effect (since a is not live at the end of the loop body) and can be discarded:

$$(a := c)_p \; \beta \; (a = N)^-_p \; \gamma \; C^-_p \; \leq \beta; \; \gamma \tag{5}$$

A successful iteration, along with the final assignment, is of the form:

$$(a := c)_p \; \beta \; (a = N)^-_p \; \gamma \; C^+_p \; \delta \; (r := a + 1)_p$$

Since $a := c$, $a = N$ and $r := a + 1$ only involve local variables, we can move the $a := c$ and $a = N$ right and $r := a + 1$ left to the position of the CAS:

$$\begin{array}{l} (a := c)_p \; \beta \; (a = N)^-_p \; \gamma \; C^+_p \; \delta \; (r := a + 1)_p \\ \leq \beta \; \gamma \; (a := c)_p \; (a = N)^-_p \; C^+_p \; (r := a + 1)_p \; \delta \end{array} \tag{6}$$

An *exceptional iteration* is of the form:

$$(a := c)_p \; \beta \; (a = N)^+_p \; \gamma \; (r := \bot)_p$$

This time we cannot move $a := c$ right over β, since β might alter c in a way that invalidates the result of the test $a = N$. We can, however, move $(a = N)_p$ and $(r := \bot)_p$ left over β and $\beta \; \gamma$, respectively:

$$(a := c)_p \; \beta \; (a = N)^+_p \; \gamma \; (r := \bot)_p \leq (a := c)_p \; (a = N)^+_p \; (r := \bot)_p \beta \; \gamma \tag{7}$$

Thus, the linearisation point is the assignment which reads c. This is a common pattern for (cases of) operations that do not change any shared data — but one that is often handled incorrectly in informal proofs of linearisability. In this case, the effect is the same as if the value of c was tested as soon as it was loaded.

Thus, we can reduce any execution containing a complete execution of *inc* to one in which *inc* is executed without interruption, so the implementation of *inc* is atomic.

In the case of a successful iteration, the sequence of p-actions is equivalent to $c \neq N \wedge c' = c + 1 \wedge r'_p = c + 1$, which correctly implements *inc* for the case where the counter has not reached its bound.

In the case of an exceptional iteration, the sequence of p-actions is equivalent to $c = N \wedge c' = c \wedge r'_p = \bot$, which correctly implements *inc* for the case where the counter has reached its bound.

If the only operation provided for the counter is *inc*, if the counter reaches its bound it will remain there indefinitely. If the counter also provided a *dec* operation, as outlined earlier, it would be possible for the counter to reach its bound and at some later stage be reduced so that it is again below the bound. In this case the fact that *inc* returns $\bot$ would just mean that it saw $c = N$ at some stage, not that c is equal to N when the operation returns. Similarly, the fact that *inc* does not return $\bot$ does not mean that c has not reached its bound when the operation returns, just that it had not reached its bound at the point where it was read. Both of these behaviours are perfectly acceptable under the definition of linearisability.

6 A Simple Lock-Free Stack

We now consider a simple lock-free stack, based on that described by Michael and Scott [Michael and Scott, 1998] and attributed to Treiber [Treiber, 1986].[5] The stack implementation employs a simple linked list representation, using the declarations shown in Fig. 6.

The implementations of the stack operations are also shown in Fig. 6. We assume automatic dereferencing; for example, we write $n.val$ to refer to the val field of the node pointed to by n.

```
**type** $Ptr$ = **pointer to** $Node$
**type** $Node = (val : T;\ next : Ptr)$
**var** $Top : Ptr := null$
```

```
$push($**in** $x : T) \mathrel{\widehat=}$
  **var** $n, ss : Ptr$;
  $n :=$ **new** $Node()$;
  $n.val := x$;
  **repeat**
    $ss := Top$;
    $n.next := ss$
  **until** $CAS(Top, ss, n)$
```

```
$pop($**out** $y : T_\perp) \mathrel{\widehat=}$
  **var** $ss, ssn : Ptr$;
  **repeat**
    $ss := Top$;
    **if** $ss = null$ **then** $y := \perp$; **return fi**;
    $ssn := ss.next$
  **until** $CAS(Top, ss, ssn)$;
  $y := ss.val$
```

Figure 6: Lock-free stack implementation

These operations both have the same structure as the *inc* operation discussed earlier. They each repeatedly take a snapshot, *ss*, of the top of the stack, use it to prepare a new value, and then use a CAS to update the top of the stack, providing it has the same value it had when the snapshot was taken.

[5] Treiber's version is given in System/370 assembler code; our version is based on the pseudocode version presented in [Michael and Scott, 1998].

A *push* operation allocates a new node and stores the value to be pushed in its *val* field. On each iteration of the loop, *push* reads the current top of stack, stores it in the *next* field of the new node, and attempts to make *Top* point to the new node.

A *pop* operation repeatedly reads the current top of the stack and tests to see if the stack is empty, in which case *pop* returns $\perp$. If the stack is not empty, *pop* reads the *next* field from the first node and attempts to make *Top* point to that node. When the loop exits, the *val* field of the node just removed from the list is returned as the popped value.

We will now consider how to show that *push* and *pop* are atomic.

6.1 Atomicity of *push*

A completed execution of *push* consists of two initial assignments, followed by zero or more failed iterations of the loop, followed by a successful iteration.

A failed iteration again has no visible effect — it assigns to a local variable (ss) and a field of a heap location to which the process has a unique pointer ($n.next$), neither of which is live at the end of the loop body since the next accesses to them are assignments in the next iteration — so can be deleted.

$$(ss := Top)_p\, \alpha\, (n.next := ss)_p\, \beta\, CAS(Top, ss, n)^-_p \leq \alpha\, \beta \tag{8}$$

Thus, we only need to consider an execution consisting of the two initial assignments and a successful iteration of the loop. The successful CAS is clearly the linearisation point, so we want to show that the other statements can be moved right over steps of other processes.

The assignments to $n.val$ and $n.next$ are both movers, and thus right movers, since n is a unique pointer.

$$\begin{gathered}(n.val := x)_p\, \alpha \leq \alpha\, (n.val := x)_p \\ (n.next := ss)_p\, \alpha \leq \alpha\, (n.next := ss)_p\end{gathered} \tag{9}$$

Two executions of either of these assignments by different processes can commute because they must be assigning to fields of different nodes. Likewise, an execution of $y := ss.val$ or $ss := ss.next$ by another process can commute with $n.val := x$ or $n.next := ss$, respectively, since ss and n must point to different nodes. This is a consequence of our assumption that the storage allocator always returns a new node to which no other pointer exists.

The assignment which allocates a new node can also be treated as a both mover, and thus a right mover. It clearly commutes with any action other than another storage allocation. Two storage allocating assignments performed by different processes commute since we assume that nodes are allocated nondeterministically. Thus, if there is an execution in which p is allocated pointer l_1 and

q is later allocated pointer l_2, there is also an execution in which q is allocated l_2 and p is later allocated l_1.[6]

$$(n := \mathbf{new}\ \mathit{Node}())_p\, \alpha \leq \alpha\, (n := \mathbf{new}\ \mathit{Node}())_p \tag{10}$$

Finally, in a successful iteration, we know that the value of *Top* when the successful CAS is executed is the same as it was when *Top* was read into *ss*. We can therefore move $ss := \mathit{Top}$ right over any steps between it and the next execution of $n.\mathit{next} := ss$ by that process, which we have already shown can be moved right to be next to the subsequent CAS.

$$\begin{aligned}&(ss := \mathit{Top})_p\ \alpha\ (n.\mathit{next} := ss)_p\ \mathit{CAS}(\mathit{Top}, ss, n)_p^+\\ &\leq \alpha\ (ss := \mathit{Top})_p\ (n.\mathit{next} := ss)_p\ \mathit{CAS}(\mathit{Top}, ss, n)_p^+\end{aligned} \tag{11}$$

In order for the CAS to update the stack correctly, we require that *ss* and *ss.next* are both equal to *Top*. The former is guaranteed by the CAS; the latter is guaranteed by the fact that we can commute assignments to *n.next* by different processes, which follows from the fact that n is guaranteed to be a unique pointer because storage is not reused.

Notice that in this case, we argue that $ss := \mathit{Top}$ can be moved over all of α, not over individual steps of α. In this implementation, it is possible for elements to be pushed onto the stack and then popped off again between p reading its snapshot and performing its CAS, so the stack data structure (and thus the abstract stack) has changed to a different value and back the its former value. This is similar to the case of a counter with *inc* and *dec* operations, and in this case the guarantee provided by CAS is just what we want — using LL/SC would cause the operation to retry unnecessarily in this situation.

Combining (8), (9), (10) and (11), we conclude that the implementation of *push* is atomic, since any execution containing a completed execution of *push* can be reduced to one in which the following sequence is executed serially:

$$n := \mathbf{new}\ \mathit{Node}();\ n.\mathit{val} := x;\ ss := \mathit{Top};\ n.\mathit{next} := ss;\ \mathit{CAS}(\mathit{Top}, ss, n)^+$$

This sequence is equivalent to executing the obvious sequential implementation of *push* shown in Fig. 7. We thus conclude that the implementation of *push* is linearisable.

6.2 Atomicity of *pop*

A completed execution of *pop* consists of zero or more unsuccessful iterations of the loop, followed by either: a successful iteration and an occurrence of $y :=$

[6] Alternatively, we can argue that changing the location that is allocated does not alter the observable behaviour, so both executions have the same history.

```
$push(\mathbf{in}\ x : T) \mathrel{\widehat{=}}$
    $\mathbf{var}\ n : Ptr;$
    $n := \mathbf{new}\ Node();$
    $n.val := x;$
    $n.next := Top;$
    $Top := n$
```

```
$pop(\mathbf{out}\ y : T_\perp) \mathrel{\widehat{=}}$
    $\mathbf{if}\ Top = null\ \mathbf{then}$
        $y := \perp$
    $\mathbf{else}$
        $y := Top.val;$
        $Top := Top.next$
    $\mathbf{fi}$
```

Figure 7: Sequential stack implementation

$ss.val$, returning a normal result which is an element of T; or an exceptional iteration, returning $\perp$ because an empty stack has been detected.

As in *push*, an unsuccessful iteration has no visible effect — in this case, it assigns to local variables ss, ssn and y, none of which is live at the end of the loop body — so can be deleted.

$$\begin{aligned}&(ss := Top)_p\ \alpha\ (ss = null)^-_p\ \beta\ (ssn := ss.next)_p\ \gamma\ CAS(Top, ss, ssn)^-_p\\ &\leq \alpha\ \beta\ \gamma\end{aligned} \tag{12}$$

In a successful iteration, the successful CAS is the linearisation point — it is where the effect of the *pop* becomes visible to other processes, and we cannot commute it with steps of other processes that access c. Thus, we have to show that $ss := Top$, $ss = null$ and $ssn := ss.next$ are right movers. We also need to show that the $y := ss.val$ following a successful iteration (which we will henceforth consider to be part of the successful iteration) is a left mover.

Again, as in *push*, the assignment $ss := Top$ can be moved right over any steps between it and the next execution of $ss = null$ by that process because the successful CAS guarantees that is reads the same value of Top. We will show below that we can move $ss = null$ and $ssn := ss.next$ right so that they are adjacent to the subsequent CAS, so the result we need is:

$$\begin{aligned}&(ss := Top)_p\ \alpha\ (ss = null)^-_p\ (ssn := ss.next)_p\ CAS(Top, ss, n)^+_p\\ &\leq \alpha\ (ss := Top)_p\ (ss = null)^-_p\ (ssn := ss.next)_p\ CAS(Top, ss, n)^+_p\end{aligned} \tag{13}$$

The test $ss = null$ only accesses local variables, so is a right (and left) mover:

$$(ss = null)_p\ \alpha \leq \alpha\ (ss = null)_p \tag{14}$$

The assignment $ssn := ss.next$ is more subtle, since ss is not a unique pointer. However, we can show that in this implementation, no process ever modifies the *next* field of a node once it has been assigned a non-null value, since the only assignment to *next* is in *push*, when n is a unique pointer. This means that an execution of $ssn := ss.next$ cannot be followed by an execution of $n.next := Top$ in which ssn and n are equal. Thus, we can treat $ssn := ss.next$ a right mover.

$$(ssn := ss.next)_p\ \alpha \leq \alpha\ (ssn := ss.next)_p \tag{15}$$

Finally, the assignment $y := ss.val$ is a left (and right) mover because in this implementation the *val* field of a node is never altered after it becomes visible to other processes.

$$\alpha\ (y := ss.val)_p \leq (y := ss.val)_p\ \alpha \tag{16}$$

Combining (12), (13), (14), (15) and (16), we can see that any execution containing a completed execution of *pop* which performs a successful iteration can be reduced to one in which the following sequence is executed serially:

$$ss := Top;\ (ss = null)^-;\ ssn := ss.next;\ CAS(Top, ss, n)^+;\ y := ss.val \tag{17}$$

In order for the CAS to update the stack correctly, we require that ss be equal to Top and that ssn be equal to $Top.next$. Again, the former is guaranteed by the CAS. The latter is guaranteed by the fact that we can commute $ssn := ss.next$ with occurrences of $n.next := ss$, which follows from the fact that no action can modify the *next* field of a node once it has been added to the stack by a *push* operation.

In an exceptional iteration, we cannot treat the assignment $ss := Top$ as a mover, so we take it as the linearisation point, for the same reason that we took $a := c$ as the linearisation point for *inc* in Section 5.2, when the counter reached its bound. We can now treat the $ss = null$ and $y := \perp$ as left movers, since they are local.

$$\begin{aligned} &\alpha\ (ss = null)_p \leq (ss = null)_p\ \alpha \\ &\alpha\ (y := \perp)_p \leq (y := \perp)_p\ \alpha \end{aligned} \tag{18}$$

Applying (18), we can see that any execution containing a completed execution of *pop* which performs an exceptional iteration can be reduced to one in which the following sequence is executed serially:

$$ss := Top;\ (ss = null)^+;\ y := \perp \tag{19}$$

Since any completed execution of *pop* performs either a successful iteration or a exceptional iteration, any execution containing a completed execution of *pop* can be reduced to one containing either the serial code shown in (17) or that in

(19). Since both of these executions can be obtained by executing the code for *pop* without interruption, it follows that the implementation *pop* is atomic.

The serial executions in (17) and (19) are equivalent, respectively, to:

$$(Top = null)^{-};\ y := Top.val;\ Top := Top.next$$

and

$$(Top = null)^{+};\ y := \bot$$

and these are precisely the executions we can obtain from the obvious sequential implementation of *pop* shown in Fig. 7. We thus conclude that the implementation of *pop* is linearisable.

Note that in simplifying the equivalent sequential execution we have swapped the order of $y := Top.val$ and the CAS in order to get the correct value for y without using an extra local variable. It is easy to see that we could have put this assignment before the CAS, which would still be correct since $y := ss.val$ is also a right mover, but then it would be needlessly executed in every failed iteration.

6.3 Memory management and the ABA problem

The above verification relied on the assumption that heap locations are never reused. This is, of course, undesirable since it means that the storage used is proportional to the total number of *push* operations executed — such a *memory leak* is generally regarded as an error and renders the implementation impractical for most applications. We will now explain why this assumption is important, and explore some of the ways in which this assumption might be relaxed and their consequences.

The fundamental problem is that when a node is popped off the stack, other processes may hold pointers to it in their local variables. So if *pop* were to free the popped node, as it would in a sequential implementation, another process holding a pointer to that node may subsequently attempt to access a piece of storage which is no longer part of this program's memory space, in which case it will probably get some kind of memory violation error. It is also possible that the storage is reallocated to the program by a subsequent *push* operation, violating our assumption that *new* returns a unique pointer and invalidating the parts of the proof that relied on that assumption.

The easiest way to relax the assumption that heap locations are not reused is to assume that the program is executed in an environment with automatic garbage collection. In this case, we can assume that a node is only reclaimed if there is no pointer to it within the program. We can therefore also continue to assume that *new* returns a unique pointer, and our atomicity arguments are still valid — assuming, of course, that the garbage collection mechanism is correct and does not interfere with heap locations to which the program still holds a pointer.

6.3.1 Using a free list

If garbage collection is not available, we can avoid memory violations and reduce memory usage by keeping our own free list. The *pop* operation can then place the popped node on the free list, and the *push* operation can take nodes from the free list when possible and only allocate new storage when the free list is empty — thus, we would replace the statement $n := \textbf{new}\ Node()$ by a call on an operation which takes a node from the free list if there is one and otherwise calls **new** $Node()$ to allocate a new node, and add a call at the end of *pop* to an operation which adds *ss* to the free list. In this case, the memory required is proportional to the maximum size is that the stack reaches.

Unfortunately, this does not entirely solve the problem, because it is still possible for a node to be popped from the stack and pushed onto it again, after the rest of the stack has changed, while some process still holds a pointer to it. Since we can no longer assume that *push* holds a unique pointer to the node it allocates, we need to revisit parts of our atomicity proof that relied on it.

In *push*, we can no longer argue that two processes performing assignments to $n.next$ must be assigning to different nodes because the allocated pointers are unique. We can, however, argue that it is not possible for two *push* operations that have not yet performed a successful CAS to hold pointers to the same node. This is because, in order for a second process to obtain a pointer to the node, the node would have to become visible to a *pop* operation which can then pop it, so that it can then be pushed again, and this can only happen after the first pushing process completes a successful CAS.

Our argument that executions of $n.next := ss$ and $ssn := ss.next$ can commute also fails, and in the case of *pop*, we can no longer guarantee correct behaviour. It is now possible for a *pop* to read *Top* into *ss*, then for that node to be popped off the stack, and appear again at the top of the stack with a different *next* (and *val*) value. Thus, although the CAS in *pop* succeeds, we cannot be sure that the stack is updated correctly — we may end up losing values from the stack and/or incorrectly adding values to the stack.

This kind of situation is called an *ABA problem* since it arises when a variable being updated using a CAS can change from one value, A, to another value, B, and then back to its former value, A (cf. Section 2.1). CAS can only tell whether the value of a variable is the same as it was at some earlier point, but this is not sufficient to ensure that the update performed by the CAS is correct. The reason why this is a problem here is that, unlike the shared counter example, the correctness of the operation relies on the fact that other things have not changed — in this case, the *next* fields of certain nodes.

We could obtain a correct implementation by using LL/SC to read the snapshot and update *Top*. Since the SC would only succeed if *Top* had not changed since it was read by the preceding LL, it would fail in the situation when a node was popped off the stack and pushed onto it again between a process performing its LL and its SC. This would cause the implementation to retry in some situations where it doesn't need to, but this seems inevitable.[7] As discussed earlier, practical implementations of LL/SC are not widely available.

6.3.2 Adding modification counts

A popular pragmatic "solution" to the ABA problem is to emulate the behaviour of LL/SC by associating a *modification count* with each pointer variable that is to be updated using a CAS. Each time the variable is modified, its modification count is incremented, and the CAS tests the pointer along with its modification count. This way, if a pointer changes from pointing to a given node and later changes back to pointing to it again, its modification count will have changed, and the CAS will fail.

In the stack implementation, we only update the *Top* pointer, so we can associate a modification counter with it, and increment it each time *Top* is changed. Thus, in the scenario outlined above, where a node is popped from the stack and later pushed onto it again while a process still has a pointer from a snapshot taken before the node was popped, that process's CAS will now fail because *Top*'s modification count has changed. In terms of the atomicity proof, we can once again argue that actions that read and write the *next* fields of nodes can commute, because the CAS will always fail if anything changes that has the potential to invalidate its update.

The resulting code is shown in Fig. 8. We have changed the type of *Top* to a new type, *PtrC*, consisting of a pointer along with a modification count, and replaced *ss* by *ss.ptr* in the places where we want to test or copy the pointer value. In both CASes, the new value for *Top* now has a modification count, computed by incrementing the modification count of the snapshot. We write this as though it is computed in the call to CAS, as is common in presenting such algorithms — the implementation may involve first constructing this value in a local variable, but we know that this statement would be a right mover and can thus be elided. We have also made the changes describe above to utilise a free list.

With these modifications, we can again treat $ss := Top$ as a right mover, because we now know that it is not possible for *Top* to change from an earlier

[7] This could be avoided by using a Double Compare and Swap (DCAS), or a variant of CAS which tests two locations and updates one of them — it would then be possible for *pop* to test that *ssn* is equal to *Top.next*, as well as testing that *ss* is equal to *Top*.

```
**type** $Ptr$ = **pointer to** $Node$
**type** $Node = (val : T;\ next : Ptr)$
**type** $PtrC = (ptr : Ptr;\ mc : int)$
**var** $Top : PtrC := (null, 0)$
```

```
$push(\textbf{in}\ x : T) \mathrel{\widehat{=}}$
  var $n, ss : Ptr$;
  $n := new\_node()$;
  $n.val := x$;
  repeat
    $ss := Top$;
    $n.next := ss.ptr$
  until $CAS(Top, ss, (n, ss.mc{+}1))$
```

```
$pop(\textbf{out}\ y : T_\perp) \mathrel{\widehat{=}}$
  var $ss, ssn : Ptr$;
  repeat
    $ss := Top$;
    if $ss.ptr = null$ then $y := \perp$; return fi;
    $ssn := ss.ptr.next$
  until $CAS(Top, ss, (ssn, ss.mc{+}1))$;
  $y := ss.ptr.val$;
  $free\_node(ss)$
```

Figure 8: Stack implementation with modification counts

value and back again, since if *Top* is changed its modification count will have increased and can never be decreased to its earlier value. The rest of the atomicity argument remains unchanged.

This argument, however, is only valid if we assume that modification counts are unbounded, whereas to be practical the approach requires that a pointer and its modification count can be tested and updated using a single CAS. If a pointer requires 32 bits and a CAS operates on 64-bit values, we only have 32 bits for the modification count, so it is still possible for the modification count to wrap around and return to the same value as the snapshot. The chance of this actually occurring can be shown to be extremely small [Moir, 1997], and is generally assumed to be small enough to make this solution acceptable for practical purposes.

The implementation given in Fig. 8 is similar to the stack implementation given in [Michael and Scott, 1998], attributed to Treiber [Treiber, 1986]; the main difference being that their version associates modification counts with all of the nodes in the linked list even though they are never used. Treiber's version is given in System/370 assembler language and actually implements a free list, so nodes do not have values associated with them, and his equivalents of *push* and *pop* take and return pointers and do no storage allocation.

Notice that, like LL/SC, this version will cause some operations to retry when they don't really need to, because the stack has changed in a way that leaves the previous version intact, but this seems to be unavoidable. It does, however, serve to emphasise that "ABA" situations are not always bad — and, for example, it would be silly to add modification counts to the version in Fig. 6 where no memory reuse is done or if automatic garbage collection is used.

An alternative approach that does not suffer from the boundedness of modification counts can be found in [Herlihy et al., 2002]. Efficient management of dynamic storage in lock-free algorithms is the subject of on-going research (e.g. [Detlefs et al., 2001]) and further discussion is beyond the scope of this paper.

7 Conclusions

We have discussed two approaches to proving linearisability of nonblocking implementations of shared data structures. We began by presenting the simulation approach which we have used before, but explaining it in terms of augmented executions and augmented histories.

We then showed how Lipton's reduction method can be extended to show linearisability of nonblocking algorithm, and used a series of simple examples to illustrate the kinds of properties required to show that a concurrent execution can be transformed into an equivalent serial execution. We saw that for this kind of algorithm, we need to treat the same action differently in different executions, and had to remove actions that we could show had no observable effect. We also saw that the commutativity and other transformations we used were justified by a range of program properties. Some of these can be verified using simple forms of static analysis (such as actions only accessing local variables or heap locations accessed via unique pointers, or liveness analysis); others relied on deeper dynamic properties, such as the fact that a field of a node is never changed once it is assigned a non-null value, or that if a node that was in a stack earlier is still in the stack, it has been there continuously since the first observation. Some of these properties may also be amenable to static analysis, others may be able to be verified using model checking, or may require the full strength of deductive verification.

In these examples, it was straightforward to identify a linearisation point in the code of each operation, but as we noted in Section 3, this is not always the case. In a related paper [Groves and Colvin, 2006], we have shown that reduction can be used in a constructive way and used it to derive a more complicated stack implementation, based on the one in [Hendler et al., 2004], which uses an elimination mechanism whereby *push* and *pop* operations that experience interference can pair off and be eliminated without altering the central stack.

In this case we had to reduce two operations at once — essentially, we had a situation where a sequence of three actions performed by two processes, $\phi_p\, \theta_q\, \psi_p$, formed a joint action consisting of a *push* immediately followed by a *pop*.

We have also used this approach to verify Michael and Scott's lock-free queue algorithm [Michael and Scott, 1998]. In that verification [Groves, 2008] we needed to alter the actions in an execution as well as rearranging and/or deleting them. In this algorithm, the action of advancing the tail pointer after a node has been appended in an *enqueue* operation may be done by a different process, so we show that if there is an execution in which this is done by one process there is an equivalent execution in which it is done by the process performing the *enqueue*. More general treatment of this kind of situation (which also occurs in several other algorithms, e.g. [Valois, 1995; Shann et al., 2000; Ladan-Mozes and Shavit, 2004]) may rely on identifying actions that are *process independent*, in the sense that it doesn't matter which process executes them.

Compared to the simulation approach described in Section 3, the reduction approach allows us to work with algorithms in their original form, and utilises their algorithmic structure, rather than translating them into a transition system form where this structure is lost. The verification is also more directly concerned with the behaviour of the algorithm, rather than with additional control mechanisms needed to ensure that only valid executions are considered. Whereas the simulation approach translates an (augmented) execution into an equivalent sequential history by considering one step of the execution at a time, focusing on invariants that are maintained, the reduction approach performs a similar translation by considering the entire execution of one abstract operation at a time, focusing on interferences that may occur between processes.

The only published work we know of that addresses the application of reduction to nonblocking algorithms is [Wang and Stoller, 2005], which uses static analysis to show atomicity. They give results claiming that a successful $SC(v, val)$ and its matching $LL(v)$ can be treated as a left-mover and right-mover, respectively,[8] but do not consider CAS other than when used in conjunction with modification counts. They use a notion of *purity*, taken from [Freund and Qadeer, 2005], to justify discarding failed executions which have no observable effect.

While it is clear that many of the properties we have used in proving reductions could not be detected by a static analyser, it would be interesting to investigate whether some form of annotation system would allow a programmer to easily provide sufficient information for this kind of static analysis to be applied more widely.

In future work, we intend to undertake further verifications to see what other

[8] This result appears to be false, since $SC(v, val)$ cannot be commuted with an $LL(v)$ performed by a different process.

extensions to the reductions method may needed, what kinds of program properties are required in order to justify the transformations required, and what kinds of methods are best suited to verifying these properties. We also intend to mechanise these proofs to allow a more direct comparison with our mechanised simulation proofs.

Acknowledgements

Thanks are due to Sun Microsystems Laboratories for financial support, to Simon Doherty and David Friggens for helpful comments and corrections on earlier drafts, to Robert Colvin, Mark Moir and Victor Luchangco for helpful discussion relating to our work, and to Ernie Cohen for helpful advice, references and suggestions regarding reduction.

References

[Back, 1993] Back, R.-J. (1993). Atomicity refinement in a refinement calculus framework. Reports on Computer Science and Mathematics 141, Åbo Akademi.

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

[Cohen and Lamport, 1998] Cohen, E. and Lamport, L. (1998). Reduction in TLA. In *International Conference on Concurrency Theory (CONCUR)*, pages 317–331.

[Colvin et al., 2005] Colvin, R., Doherty, S., and Groves, L. (2005). Verifying concurrent data structures by simulation. In Boiten, E. and Derrick, J., editors, *Proc. Refinement Workshop (REFINE 2005)*, volume 137(2) of *Electronic Notes in Theoretical Computer Science*, pages 93–110, Guildford, UK. Elsevier.

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

[Colvin and Groves, 2007] Colvin, R. and Groves, L. (2007). A scalable lock-free stack algorithm and its verification. In *SEFM*, pages 339–348. IEEE Computer Society.

[Colvin et al., 2006] Colvin, R., Groves, L., Luchangco, V., and Moir, M. (2006). Formal verification of a lazy concurrent list-based set algorithm. In Ball, T. and Jones, R. B., editors, *Proc. 18th International Conference on Computer Aided*

Verification (CAV), volume 4144 of *Lecture Notes in Computer Science*, pages 475–488. Springer.

[de Roever and Engelhardt, 1998] de Roever, W.-P. and Engelhardt, K. (1998). *Data Refinement Model-Oriented Proof methods and their Comparison*. Cambridge University Press. (With the assistance of J. Coenen and K.-H. Buth and P. Gardiner and Y. Lakhnech and F. Stomp).

[Derrick et al., 2008] Derrick, J., Schellhorn, G., and Wehrheim, H. (2008). Mechanising a correctness proof for a lock-free concurrent stack. In *10th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems (FMOODS'08)*. to appear.

[Detlefs et al., 2001] Detlefs, D. L., Martin, P. A., Moir, M., and Steel Jr, G. L. (2001). Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing,*.

[Doherty, 2003] Doherty, S. (2003). Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington.

[Doherty et al., 2004a] Doherty, S., Detlefs, D., Groves, L., Flood, C. H., Luchangco, V., Martin, P. A., Moir, M., Shavit, N., and Jr., G. L. S. (2004a). DCAS is not a silver bullet for nonblocking algorithm design. In Gibbons, P. B. and Adler, M., editors, *Proc. Sixteenth Annual ACM Symposium on Parallel Algorithms (SPAA)*, pages 216–224. ACM.

[Doherty et al., 2004b] Doherty, S., Groves, L., Luchangco, V., and Moir, M. (2004b). Formal verification of a practical lock-free queue algorithm. In de Frutos-Escrig, D. and Núñez, M., editors, *Formal Techniques for Networked and Distributed Systems (FORTE)*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer.

[Doherty et al., 2004c] Doherty, S., Herlihy, M., Luchangco, V., and Moir, M. (2004c). Bringing practical lock-free synchronization to 64-bit applications. In Chaudhuri, S. and Kutten, S., editors, *PODC*, pages 31–39. ACM.

[Flanagan and Freund, 2004] Flanagan, C. and Freund, S. (2004). Atomizer: a dynamic atomicity checker for multithreaded programs. In *Proc. 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–267.

[Flanagan and Qadeer, 2003] Flanagan, C. and Qadeer, S. (2003). A type and effect system for atomicity. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–349.

[Fraser and Harris, 2007] Fraser, K. and Harris, T. (2007). Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25(2).

[Freund and Qadeer, 2005] Freund, S. N. and Qadeer, S. (2005). Exploiting purity for atomicity. *IEEE Trans. Softw. Eng.*, 31(4):275–291.

[Groves, 2007] Groves, L. (2007). Reasoning about nonblocking concurrency using reduction. In *ICECCS*, pages 107–116. IEEE Computer Society.

[Groves, 2008] Groves, L. (2008). Verifying Michael and Scott's lock-free queue algorithm using trace reduction. In Harland, J. and Manyem, P., editors, *Computing: The Australasian Theory Symposium (CATS 2008)*, Wollongong, Australia.

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

[Guerraoui, 2004] Guerraoui, R., editor (2004). *Distributed Computing, 18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004, Proceedings*, volume 3274 of *Lecture Notes in Computer Science*. Springer.

[Hatcliff et al., 2004] Hatcliff, J., Robby, and Dwyer, M. B. (2004). Verifying atomicity specifications for concurrent object-oriented software using model checking. In *Proc. Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, pages 175–190.

[Heller et al., 2005] Heller, S., Herlihy, M., Luchangco, V., Moir, M., III, W. N. S., and Shavit, N. (2005). A lazy concurrent list-based set algorithm. In Anderson, J. H., Prencipe, G., and Wattenhofer, R., editors, *OPODIS*, volume 3974 of *Lecture Notes in Computer Science*, pages 3–16. Springer.

[Hendler et al., 2004] Hendler, D., Shavit, N., and 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*, pages 206–215.

[Herlihy, 1991] Herlihy, M. (1991). Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149.

[Herlihy et al., 2002] Herlihy, M., Luchangco, V., and 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)*, volume 2508 of *Lecture Notes in Computer Science*, pages 339–353, Toulouse, France.

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

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

[III and Scott, 2004] III, W. N. S. and Scott, M. L. (2004). Nonblocking concurrent data structures with condition synchronization. In [Guerraoui, 2004], pages 174–187.

[Jifeng et al., 1986] Jifeng, H., Hoare, C., and Sanders, J. (1986). Data refinement refined. In *ESOP 86*, volume 213 of *Lecture Notes in Computer Science*, pages 187–196. Springer-Verlag.

[Ladan-Mozes and Shavit, 2004] Ladan-Mozes, E. and Shavit, N. (2004). An optimistic approach to lock-free fifo queues. In [Guerraoui, 2004], pages 117–131.

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

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

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

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

[Lynch and Vaandrager, 1995] Lynch, N. A. and Vaandrager, F. W. (1995). Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233.

[Michael, 2004] Michael, M. M. (2004). Practical lock-free and wait-free LL/SC/VL implementations using 64-bit CAS. In [Guerraoui, 2004], pages 144–158.

[Michael and Scott, 1996] Michael, M. M. and Scott, M. L. (1996). Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Symposium on Principles of Distributed Computing*, pages 267–275.

[Michael and Scott, 1998] Michael, M. M. and Scott, M. L. (1998). Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26.

[Moir, 1997] Moir, M. (1997). Practical implementations of non-blocking synchronization primitives. In *Proc. 15th Annual ACM Symposium on the Princi-*

ples of Distributed Computing (PODC 1997), Santa Barbara, CA., pages 219–228.

[Moir et al., 2005] Moir, M., Nussbaum, D., Shalev, O., and 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)*, pages 253–262, Las Vegas, Nevada, USA. ACM Press.

[Sasturkar et al., 2005] Sasturkar, A., Agarwal, R., Wang, L., and 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*, pages 83–94, New York, NY, USA. ACM Press.

[Shann et al., 2000] Shann, C.-H., Huang, T.-L., and Chen, C. (2000). A practical nonblocking queue algorithm using compare-and-swap. In *Seventh International Conference on Parallel and Distributed Systems (ICPADS)*, pages 470–475.

[Shavit and Moir, 2004] Shavit, N. and Moir, M. (2004). Concurrent data structures. In Mehta, D. and Sahni, S., editors, *Handbook of Data Structures and Applications*. Chapman & Hall/CRC Press.

[Shavit and Zemach, 1999] Shavit, N. and Zemach, A. (1999). Scalable concurrent priority queue algorithms. In *In Proceedings of the 18th Annual ACM Symposium on Principals of Distributed Computing (PODC), Atlanta*, pages 113–122. ACM Press.

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

[Valois, 1995] Valois, J. D. (1995). Lock-free linked lists using compare-and-swap. In *In Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222.

[Wang and Stoller, 2005] Wang, L. and 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*, pages 61–71, New York, NY, USA. ACM Press.

[Wang and Stoller, 2006] Wang, L. and Stoller, S. D. (2006). Runtime analysis of atomicity for multithreaded programs. *IEEE Trans. Softw. Eng.*, 32(2):93–110.