The Design and Verification of Dynamic-sized Nonblocking Data Structures

by

Simon Doherty

A thesis
submitted to the Victoria University of Wellington
in fulfilment of the
requirements for the degree of
Doctor of Philosophy
in Computer Science.

Victoria University of Wellington
2010

Abstract

Modern computer systems often involve multiple processes or threads of control that communicate through shared memory. However, the implementation of correct and efficient data structures that can be shared by several processes is frequently challenging. This thesis is concerned with the design and verification of a class of shared memory algorithms known as *nonblocking algorithms*, which are implementations of shared data structures that provide strong progress guarantees. Nonblocking algorithms offer an appealing alternative to traditional techniques for the implementation of shared memory data structures, but they are difficult to design, and extant algorithms can often be applied in only a limited range of systems. Furthermore, because of their subtlety, it is notoriously difficult to determine whether a given nonblocking algorithm is correct.

This thesis addresses these difficulties in two ways. First, we present techniques for the verification of nonblocking algorithms that dynamically allocate memory. These techniques allow the construction of formal and complete proofs of correctness, so that each proof may be checked by a mechanical proof assistant. Applying techniques first developed for the verification of distributed algorithms, we use labelled-transition systems to model algorithms and their specifications, and simulation relations to prove that an implementation meets its specification. Nonblocking algorithms often require a particular notion of simulation, called *backward simulation*, that is rarely necessary in other contexts. This thesis contributes to the relatively limited collective experience in the use of backward simulation.

The second set of contributions addresses the limitations of many extant nonblocking algorithms. While many nonblocking algorithms allocate memory dynamically, it is difficult to determine in a nonblocking context when it is safe to free memory. We present techniques to accomplish this. Furthermore, many nonblocking algorithms depend on the availability of two powerful synchronisation primitives, known as *load-linked* and *store-conditional*, which are not normally provided by hardware. We present implementations of these primitives that work on commonly available platforms.

Acknowledgements

To all the people who gave me so much encouragement and patience. Especially Mum, Dad and Lindsay.

Contents

Chapter 1

Introduction

Computer systems in which independent processes concurrently access data structures present challenges not found in systems in which data structures are accessed sequentially by one process. Data structures that are to be accessed concurrently by several processes must somehow ensure that concurrent accesses maintain the consistency of the data structure.

The standard technique for implementing concurrent data structures is to use *mutual exclusion*: at most one process is allowed to execute an operation on a given structure at any given time. Mutual exclusion reduces the problem of maintaining consistency during concurrent operations to the problem of maintaining consistency during a single operation.

Unfortunately, mutual exclusion can create several software-engineering and performance issues [Gre96, Fra03]. The most prominent software-engineering issue that arises when using mutual exclusion is the problem of *deadlock*. In some systems it is necessary for several processes to acquire exclusive access to intersecting sets of data structures. In such situations, it may be possible for each process to acquire exclusive access to data structures in an order that prevents any process from making progress. The term *deadlock* describes situations in which this occurs. Techniques do exist to solve this problem (see [Bac98, Lea00] for discussion). However, software engineers still need to reason about the order in which exclusive access is acquired. This is error prone and can lead to bugs that are difficult to reproduce and fix.

Furthermore, data structures based on mutual exclusion tend to perform poorly when being accessed by a large number (dozens or hundreds) of processes. It is possible for a process to be delayed — by an action of a scheduler, a limitation of the underlying hardware, or even process failure — while holding exclusive access to a data structure. When this

happens, all processes awaiting access to that data structure are delayed as well. When many processes are awaiting access to that data structure, performance of the system as a whole can degrade massively.

These issues have motivated researchers to seek ways of implementing shared data structures that do not depend on mutual exclusion. Such implementations are known as *nonblocking algorithms*.

Because they do not rely on mutual exclusion, nonblocking algorithms avoid the problem of deadlock. Furthermore, numerous empirical studies (both simulations and experiments on real machines) have found that there are important situations in which nonblocking algorithms outperform their lock-based counterparts. ([ST95, MS98b, Har01, TZ01a, Fra03, HLMS03, SS03] provide examples.) These experiments suggest that nonblocking algorithms frequently scale better than lock-based solutions, as contention increases.

Nonblocking algorithms are typically significantly more complicated than sequential implementations or implementations based on mutual exclusion. Because of this complexity, it is very difficult to determine if an algorithm is correct. Therefore, careful researchers provide some kind of proof of correctness of novel algorithms. This thesis is partly concerned with techniques for constructing such proofs.

For reasons that we shall discuss shortly, a challenging problem in the development of a nonblocking algorithm is the question of how to determine when it is safe to reclaim memory from a nonblocking data structure. Furthermore, nonblocking algorithms frequently depend on the availability of functionality that modern systems do not provide. This thesis addresses both these issues.

The remainder of this chapter is organised as follows. Section 1.1 provides a short introduction to the field of nonblocking algorithms. This provides context for Section 1.2, which outlines the contributions presented in the thesis. Finally, Section 1.3 defines notation used in the thesis.

1.1 Nonblocking Algorithms

Nonblocking algorithms provide various *progress guarantees* about the ability of any process to complete operations in the presence of failure or delay by other processes. As discussed in Section 1.1.1, these progress guarantees come in various strengths, all of which preclude the use of mutual exclusion.

In order to support nonblocking implementations of nontrivial data structures, the under-

lying system needs to provide *strong synchronisation primitives*. These are operations that allow processes to read and modify memory locations atomically, and are typically provided by hardware. Such primitives are discussed in Section 1.1.2. Moreover, considerable ingenuity must be employed in the development of these algorithms. Section 1.1.3 describes one classic algorithm (adapted from an algorithm in [MS98a] that is itself adapted from [Tre86]), illustrating important issues associated with the design of nonblocking algorithms.

1.1.1 Nonblocking Progress Guarantees

Several nonblocking progress guarantees have been treated in the literature. Currently, the most well established are *wait-freedom* and *lock-freedom* [Her91].[1] An algorithm is *wait-free* iff for every execution, every operation is guaranteed to complete after a finite number of its own steps, regardless of the delay or failure of any other operation. An algorithm is *lock-free* iff for every execution, some operation is guaranteed to complete after a finite number of steps of the execution, regardless of the delay or failure of any other operation.

Lock-freedom is the weaker condition: lock-freedom allows the possibility that some processes *never* complete their operations. So long as some processes are completing, the others may be prevented from making progress. Wait-freedom precludes this property: every process is guaranteed to complete. Every wait-free algorithm is lock-free.

Both wait-freedom and lock-freedom preclude the use of mutual exclusion. A process that failed while holding exclusive access to a data structure would prevent all other processes from completing operations that required access to that data structure.

1.1.2 Synchronisation Primitives

Nonblocking algorithms normally make substantial use of powerful synchronisation primitives. We describe the most important such operations: the *compare-and-swap* (CAS) operation; and the *Load-linked/Store-conditional* (LL/SC) operation pair. Herlihy [Her91] has shown that any sequential data structure can be implemented using either CAS or LL/SC, and that other common synchronisation primitives (such as *test-and-set* or *swap*) are insufficient to construct nonblocking implementations of many important data structures.

[1] There is ambiguity in the literature between the terms *lock-free* and *nonblocking*. They have sometimes been used synonymously. However, we follow an existing convention whereby *nonblocking* describes the whole family of algorithms that do not rely on mutual exclusion, and *lock-free* describes a class within that family.

```
boolean CAS(val *loc,
            val old,
            val new) {
  atomically {
    if (*loc = old){
      *loc := new;
      return true;
    } else return false;
  }
}
```

Figure 1.1: Semantics of the CAS operation. Here (and through much of this thesis) we use a C-syle pseudocode. A declaration like `val *loc` specifies that `loc` is a pointer to a value of type `val`. An expression like `*loc` evaluates to the value referenced by `loc`. The expression `*loc` may be used on the left-hand side of an assigment, in which case the value at the address is changed to the value of the right-hand-side expression. We break with the C convention by denoting assignment with the symbol "`:=`", and the test for equality (returning a boolean) with "`=`".

Pseudocode representing the semantics of the CAS operation is presented in Figure 1.1. The CAS operation takes three arguments, a location `loc` (sometimes called the *target* of the CAS), and two values, `old` and `new`, and returns a boolean value. The value currently at `loc` is tested against `old`. If they are equal, then the value at `loc` is updated to `new` and the CAS returns `true` (in this case, we say that the CAS *succeeds*); otherwise, no change to the value at `loc` occurs and the CAS returns `false` (in this case we say that the CAS *fails*). These comparisons and updates happen atomically. That is, no other operation on memory appears to occur during the CAS operation.

Pseudocode representing the semantics[2] of the LL/SC operations is presented in Figure 1.2. LL and SC are used in pairs: every invocation of SC on a location `loc` by some process must follow an LL operation to `loc` by the same process, with no intervening SC to `loc` by that process. In this case, we say that the SC *matches* the earlier LL, and that the LL *matches* the SC. The LL operation reads the value from the location; the SC operation conditionally stores a new value to the location, *succeeding* and returning `true`, if no other SC to the location has succeeded since the matching LL. The SC *fails* and returns `false` otherwise, leaving the location unchanged. We say that an LL is *outstanding* if it has no matching SC.

[2]There are several possible variations on the semantics of LL/SC, that describe how LL/SC interacts with ordinary store operations, or that provide additional operations. We ignore these extensions for now.

```
val LL(val *loc) {
  return *loc;
}


bool SC(val * loc, val newval) {
  atomically {
    if (no SC has returned true
        since the last LL of
        this process) {
      *loc := newval;
      return true;
    } else {
      return false;
    }
  }
}
```

Figure 1.2: Semantics of the LL/SC operations.

(Note that if an SC matches an LL, then the SC is executed by the same process that executed the LL.)

Most contemporary multiprocessors offer either CAS or LL/SC as primitive instructions. Unfortunately, as far as we know, no hardware implementation of the LL/SC operations provides the strong semantics described above. To make hardware implementations feasible, restrictions are added [Moi97]: for instance, programmers may be limited to using only one LL at a time, without a matching SC (so only one location can be the subject of an unmatched LL at a time); reads or writes to memory may be disallowed between the time when an LL is executed and its matching SC completes; or an SC may fail *spuriously*, that is, without an SC being executed to the location since the matching LL. In practice, these restricted LL/SC operations are normally used to implement a CAS operation (as in [MS96a, Moi97]).

Both CAS and LL/SC share an important restriction: they only allow atomic modification of one location at a time. One generalisation of the CAS operation, the *double compare-and-swap* (or DCAS), does not suffer from this restriction. DCAS behaves just like CAS, but compares and modifies two independent locations, succeeding iff *both* locations contain their respective *old* values. Pseudocode representing the semantics of DCAS is presented in Figure 1.3.

As a rule, DCAS is not provided by multiprocessor systems, the only exceptions known

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1 = old1) &&
        (*addr2 = old2)) {
      *addr1 := new1;
      *addr2 := new2;
      return true;
    } else return false;
  }
}
```

Figure 1.3: Semantics of the DCAS instruction.

to us being systems based on the Motorola 68030 processor. However, the operation has received attention from researchers because of its potential to make the development of nonblocking algorithms significantly easier.

1.1.3 An Illustrative Example

We now describe several versions of a lock-free stack algorithm known as the *Treiber stack*. The algorithm was originally presented in [Tre86], but is described in a more accessible fashion in [MS98a].[3] The versions presented here are adapted from the latter. The Treiber stack is very simple, taking only a couple of paragraphs to describe, but it illustrates several important techniques used in nonblocking algorithms, and important difficulties that the designer must overcome.

A stack, a classic data structure in computer science, is an object that contains a sequence of values from some type, and provides *push* and *pop* operations which add and remove elements in a *last-in-first-out* fashion. In Treiber's algorithm, the stack is represented as a linked-list of nodes accessed by a `Head` variable. Each node has a `val` field which contains some application specific value, and a `next` field, which points to the next node in the list. The structure of the nodes contained in the stack, the global variable and the initial state are declared in Figure 1.4. Pseudocode for the stack operations is presented in Figure 1.5.

The presentations given in [Tre86] and [MS98a] describe the algorithm in terms of the

[3]The original paper presents the algorithm using System/370 assembler code.

```
 struct node {                          node *Head;
 value val; node *next                  initially Head = null;
}
```

Figure 1.4: The node structure, the global variable `Head`, and the initial condition for the stack implementation.

```
void push(value v) {                   value pop() {
H1.  nd := new node();                 P1.  while (true) {
H2.  nd->val := v;                     P2.    head := LL(&Head);
H3.  while(true) {                     P3.    if (head = null)
H4.    head := LL(&Head);              P4.      return empty;
H5.    nd->next := head;               P5.    next := head->next;
H6.    if (SC(&Head, nd))              P6.    if (SC(&Head, next))
         break;                                 break;
H7.  }                                 P7.  }
H8.  return;                           P8.  return head->val;
}                                      }
```

Figure 1.5: Pseudocode for the stack operations.

CAS operation. We describe it here using LL/SC because this is somewhat simpler. Another point to note is that the code just presented does not explicitly recycle memory. The steps that must be taken to obtain an algorithm that uses CAS and recycles memory are informative, and are discussed below.

We first describe the *push* operation. A process *p* executing *push* first allocates a new node (line H1), sets its `val` field to the value being pushed (H2) and then attempts to link the new node onto the stack. Process *p* does this by repeatedly using LL to load the current `Head` (H4); setting the `next` field of the new node to the pointer it read from `Head` (H5); and using SC to swing the `Head` pointer to the new node (H6). Once this has been achieved, the value has been successfully added to the stack, the loop terminates and *p* returns.

This looping pattern is very common in nonblocking algorithms. A process reads some shared variable (in this case `Head`); executes some operations based on that value, the effects of which are not visible to other processes (in this case, modifying the freshly allocated node at H5); and finally uses a synchronisation primitive to modify the shared variable, but only if the value of the variable has not changed since the earlier read (in this case, using the SC on line H6). If the modification fails, the process returns to the start of the loop, and tries again.

We turn now to the *pop* operation. A process *p* executing a *pop* operation enters a loop

in which it tries to remove a node from the top of the stack. *p* repeatedly reads the current value of `Head` using LL (P2) and checks if the value read is `null` (P3). If so, the stack was empty when *p* executed line P2, so *p* returns an indication that the stack was empty (P4). If `Head` was not `null`, *p* reads the `next` field of the node (P5) and then uses SC to attempt to set `Head` to the `next` value (P6), thus removing the node. If this succeeds, *p* exits the loop and returns the value contained in the node just removed.

Recycling Memory

We turn now to the issue of recycling memory. This is a difficult issue in nonblocking algorithm design. In the case of the Treiber stack, a popping process cannot simply free a node to the system after removing it from the stack, as would be possible in a sequential implementation, or one based on mutual exclusion. To see why, suppose that we replace line P8 with the following code:

```
P8:  val := head->val;
P9:  free(head);
P10: return val;
```

The resulting stack implementation frees nodes after removing them from the stack. Now, consider the following execution.

- A process *p* invokes *pop* when the stack is not empty. It loads `Head` (which is not `null`) and is delayed.
- Another process *q* invokes *pop*. It executes all of the *pop* code, including P9 and P10. Note that *q*'s `head` variable is the same as *p*'s.
- Process *p* now continues its execution, attempting the read of `head->next` at line P5. However, *q* has freed this node. Therefore, this read is illegal in many systems, and may cause an error.

The fundamental problem is that it is difficult to determine when a process has a stale reference to a block of memory (that is, to a block of memory that might be freed by another process). Because of this issue, nonblocking algorithms are normally unable to free memory to the system without additional support. A garbage collector can be used to recycle storage, since a collector can determine when no references to a piece of memory exist. However, garbage collection can only be used in certain contexts: garbage collection may

be deemed inappropriate in the context of operating system software, or it may interfere with real-time requirements. Moreover, it seems very likely that algorithms that depend on garbage collection will not be useful in the implementation of a garbage collector. Finally, some programming languages (for example, C/C++) are not well suited to garbage collection: efficient garbage collection sometimes requires the cooperation of the non-garbage collection processes, as well as precise information about the types of variables.

One solution is to never free memory to the system. Rather than returning unused memory to the system, we place it on a freelist local to the process or application. An access to a node already placed on such a freelist will not cause an error. However, this solution prevents the amount of memory used by a data structure from falling, and may not be acceptable in contexts where the size of available memory is small relative to application requirements. A spike in the frequency of push operations may cause the total memory consumed by the stack to increase, and that consumption cannot fall for the lifetime of the stack. (One important application of the Treiber stack is as a freelist that is shared by processes [MS98a]. Stack nodes are used as memory buffers in the application data structure. In such a context, the Treiber stack does not itself need a freelist, because after a node has been removed from the stack, it will be used by the application.)

Other techniques that enable unused memory to be given back to the system exist, and are discussed in Chapters 5 and 6. The main point here is that recycling memory from nonblocking data structures is tricky, and simple solutions are not always applicable.

CAS and the ABA Problem

As noted in the previous section, Treiber's stack algorithm used the CAS synchronisation primitive, rather than LL/SC. A simple attempt at using CAS to implement a nonblocking stack is presented in Figure 1.6. The LL operations at lines H4 and P2 have been replaced by reads; the SC operations at lines H6 and P6, have been replaced by CAS operations. The idea is that the CAS operation provides a similar kind of conditional update as the SC operation. Therefore, it might seem that a successful CAS operation applied to `Head` by one process should only modify `Head` if no other process has done so since the earlier read.

This algorithm will work correctly if memory is never recycled, or if garbage collection is used. However, it is incorrect in a context where memory is recycled using a local freelist. To see why, suppose that we replace line P8 with the following code

```
P8: val := head->val;
```

```
void push(val v) {
H1.  node * nd, head;
H2.  nd := new node();
H3.  nd->val := v;
H4.  while(true) {
H5.    head := Head;
H6.    nd->next := head;
H7.    if (CAS(&Head, head, nd))
         break;
H8.  }
H9.  return;
}
```

```
val pop(val *out) {
P1.  node * head, next;
P2.  while (true) {
P3.    head := Head;
P4.    if (head = null)
P5.      return empty;
P6.    next := head->next;
P7.    if (CAS(&Head, head, next))
         break;
P8.  }
P9.  return head->val;
}
```

Figure 1.6: Stack algorithm using CAS. This algorithm does not explicitly recycle memory.

```
P9:  to_freelist(head);
P10: return val;
```

where `to_freelist` adds its argument onto a freelist. Now, consider the following execution, illustrated in Figure 1.7:

- Some process *p* invokes *pop* while the stack is not empty. It loads `Head` (which is non-`null`) and then `head->next` before being delayed. This situation is illustrated in Figure 1.7(i).
- Another process *q* invokes *pop* twice, removing the top two nodes (those marked `a` and `b` in the figure). The nodes removed from the stack during these operations are placed on a freelist.
- A process *r* invokes *push*, adding a node distinct from *p*'s `next` variable onto the stack, and then *q* invokes `push`, placing the node referenced by *p*'s `head` variable onto the stack. The resulting state is illustrated in Figure 1.7(ii).
- Process *p* now continues its operation, executing the CAS at line P8 of Figure 1.6. This CAS succeeds in modifying `Head` because process *q* set that variable to be equal to *p*'s `head` variable. This results in the situation illustrated in Figure 1.7(iii). The node marked `c` has been incorrectly removed from the stack, and that marked `b` has been incorrectly added.

The problem is that CAS does not guarantee to modify a location only if the value in the location has not changed since the location was last read. It only guarantees that the value is

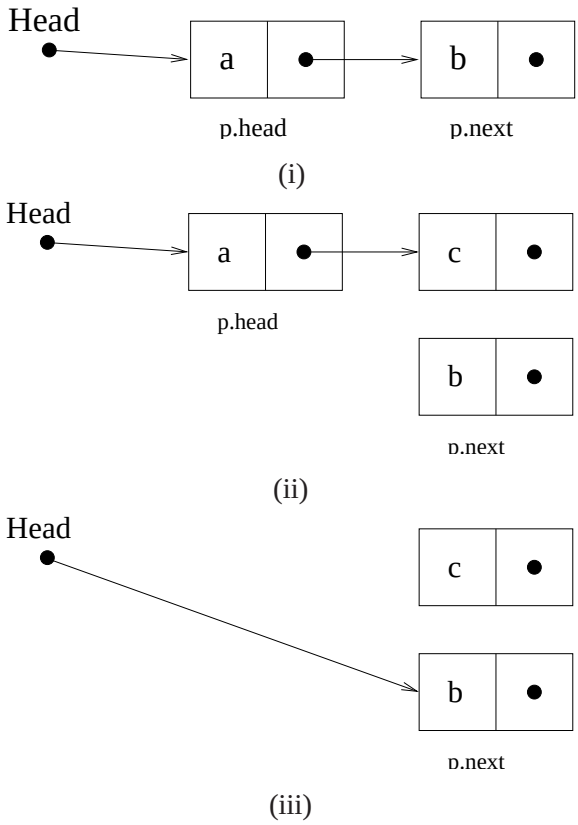


Figure 1.7: States of the stack implementation in Figure 1.6 leading to an error when recycling through a freelist.

the same as it was before. Note that the stack implementation that used LL/SC did not suffer from this problem because the semantics of LL/SC guarantee that the SC will fail if the value has changed. Also note that if memory was never recycled, the node marked `a` could never have been replaced on the stack. Similarly, if garbage collection was used the collector would delay recycling of the node marked `a` until after *p* had executed a (failing) CAS and reread the `Head` variable at line P2.

This issue with the use of the CAS operation is known as the *ABA problem* [PLJ94]. Recall that a typical way to use CAS (as exemplified in the stack algorithm) is to read a value, call it *A*, from a location (in our case `Head`), and to then use CAS to change the value at that location from *A* to a new value. The intent is often to ensure that the CAS only succeeds if the value in the location has not changed since it was read (an effect which is achieved directly by the LL/SC operations). However, the location might change to a different value *B* and back to *A* again between the read and the CAS, in which case the CAS can succeed. Typically, such a pattern will cause an algorithm to behave incorrectly.

In general, the ABA problem does not arise when garbage collection is used, so long as CAS is only used to modify pointer values, and no pointer can appear twice in the same location without first becoming unreachable and subsequently being reallocated.[4] This is the case with the Treiber stack, and numerous other nonblocking algorithms. However, as we argued earlier, garbage collection is not always applicable.

Figure 1.9 presents pseudocode for a version of the stack algorithm that does not suffer from the ABA problem, even when recycling nodes through a freelist, and does not depend on garbage collection (this is essentially the algorithm presented in the original paper [Tre86]). We introduce a new type `refint_t`, presented in Figure 1.8. Members of the `refint_t` type have both a pointer to a node and an integer, called a *version number*. There are systems where the CAS operation can atomically compare and modify both a pointer and an integer (for example, a 32-bit system with a 64-bit CAS). In such systems, an algorithm may use CAS to increment the version number of a `refint_t` every time the value of the pointer is changed. Assuming for a moment that the version number can take the value of any integer, if a location containing a `refint_t` has the same value at two points in time, then it had the same value throughout that interval.

This idea is applied in the stack by giving `Head` the type `refint_t`, and incrementing its version number at lines H6 and P6. Now, an execution like the one illustrated in Figure 1.7

[4] This situation can always be achieved by introducing a level of indirection between values and locations.

```
struct {node *ptr;
        int ver} refint_t
```

Figure 1.8: The `refint_t` type. If `x` has type `refint_t`, we use `x.ptr` to refer to the pointer member, and `x.ver` to refer to the integer.

cannot occur. The modifications to the version number that would be carried out by processes *q* and *r* during their operations would prevent *p*'s CAS from succeeding.

This version number technique, or one very like it, is used in a range of other nonblocking algorithms ([TSP92, Moi97, MS98a, LMS03a, HF03, JP03, DHLM04] provide examples). Note that so far we have pretended that the version number can increase without bound. However, in practice, version numbers can only represent a finite range of values, and the version number may wrap-around to a value that was previously in the location. But if sufficient bits are used to represent the version number, it can be made extremely unlikely that this wrap-around will cause a problem [Moi97].

However, the version-number technique requires that CAS be able to modify two adjacent values simultaneously: the pointer and the version number. This is impossible in many multiprocessor systems. If the system uses 64-bit pointers, but only provides a 64-bit CAS, the CAS cannot atomically modify both the pointer and the version number.

As discussed in Section 1.2 and in Chapters 5 and 6, this thesis presents techniques for overcoming the ABA problem, as well as enabling storage reclamation from nonblocking data structures.

1.1.4 Verification of Nonblocking Algorithms

As has already been mentioned, nonblocking algorithms are typically more complicated than their lock-based counterparts. This extra complexity often makes it difficult to determine whether an algorithm is correct. Indeed, several algorithms from the literature have been shown to be incorrect after publication (for example, [MP91, DFG+00, Val94, Val95, TZ01b]).

Because of this, careful researchers provide evidence that their novel algorithms are correct. This evidence typically takes one of two forms:

- Some kind of rigorous testing or model-checking is carried out (eg., [MS96b, Har01]).
- Some sort of manually constructed formal proof or semi-formal argument is presented,

```
struct node {
  val val; node *next
}

refint_t *Head;
initially Head = (null, 0);
```

```
void push(val v) {
H1. nd := new_node();
H2. nd->val := v;
H3. while(true) {
H4.   head := Head;
H5.   nd->next := head.ptr;
H6.   if (CAS(&Head, head,
           (nd,
            head.ver+1)))
        break;
H7. }
H8. return;
}
```

```
val pop(val *out) {
P1. while (true) {
P2.   head := Head;
P3.   if (head.ptr = null)
P4.     return empty;
P5.   next := head.ptr->next;
P6.   if (CAS(&Head, head,
           (next,
            head.ver+1)))
        break;
P7. }
P8. return head.ptr->val;
}
```

Figure 1.9: Types, initial state and pseudocode for the version number stack.

purporting to show that the algorithm is correct (eg., [DFG+00, JP03, MNSS05]).

Rigorous testing and model-checking can be used to find errors in many systems. Both approaches suffer from the drawback that typically they only examine *some* possible executions of a given algorithm, rather than guaranteeing correctness for all executions. (Any model checking algorithm can only enumerate finitely many states directly, in finite time. Some advanced model-checking techniques may be able to check all executions of an unbounded system, using a bounded approximation of the original system. We return to this issue in Chapter 2).

The other approach, manually constructing a formal or semi-formal argument, is perhaps more popular. A formal argument or proof has the advantage that it covers all possible executions of an algorithm. Unfortunately, published proofs are often long and difficult, or lacking in rigor. Both these conditions make the proofs less reliable. For example, [DFG+00, TZ01b] present incorrect algorithms, along with semi-formal correctness arguments.

[Gao05] has noted that many nonblocking algorithms seem to have the property that automatic verification is impossible and manual verification is inadequate. Some recent work [CDG05, Doh03, DGLM04, GGH05a, GGH05b] has attempted to find a middle ground, using *proof checking* and *machine-assisted theorem proving* to verify nonblocking algorithms. As discussed in the next section, the development of such techniques is one of the central concerns of this thesis.

1.2 Contributions and Overview of the Thesis

The contributions of this thesis can be divided into two categories. The first involves the verification of nonblocking algorithms. The second involves lifting the restrictions suffered by many nonblocking algorithms relating to memory reclamation and the availability of synchronisation primitives. Accordingly, this thesis is divided into two parts. Part I is concerned with the verification of nonblocking algorithms. Part II is concerned with memory reclamation techniques and synchronisation primitives.

1.2.1 Techniques for Verifying Nonblocking Algorithms

As discussed in Section 1.1.4, it is desirable to develop techniques for the verification of nonblocking algorithms that provide a greater level of assurance of correctness than the standard techniques currently used. Part I of this thesis describes and applies such techniques to

the verification of nonblocking algorithms. Much of this work is built from techniques first developed in [Doh03], which in turn are based on work originally developed for the verification of distributed systems [LV93, Lyn96]. The work reported in [Doh03] was an attempt to model and verify an algorithm known as the *Snark algorithm* [DFG+00], which is an implementation of a double-ended queue (a structure containing a sequence of values that supports both *add* and *remove* operations at both ends). That effort resulted in the discovery that the algorithm as originally published is incorrect. The verification work of this thesis extends that of [Doh03] by showing how algorithms of greater complexity can be verified, using a broader range of verification techniques.

Chapter 2 describes a technique in which both implementations and their specifications are formalised as *labelled transition systems* (LTS). This allows us to apply a powerful technique that uses *simulation relations*. A simulation relation is a relation between the states of an LTS representing an implementation and an LTS representing a specification, the existence of which guarantees that every observable behaviour of the implementation is allowed by the specification.

The verification work presented in this thesis has three important features. First, we verify algorithms that use dynamically allocated memory and present a useful technique for describing the properties of this memory within a simulation relation. This technique is applied to verifications in Chapters 3 and 6.

Second, we use a certain kind of simulation relation called a *backward simulation*. Backward simulations are needed very infrequently in most verification contexts, but are more often necessary in the verification of nonblocking algorithms (algorithms from [Blo88, Fra03, DDG+04, MNSS05] would all require backward simulations if verified using simulation relations). Because of this and the fact that backward simulations appear to be, in some sense, trickier than the simulations that are typically required, we believe that this work contains useful insights into the verification of nonblocking algorithms. Verifications presented in Chapters 3 and 4 use backward simulations.

Third, all the verifications presented in this thesis have been proof-checked using the PVS proof assistant [PVS]. This gives them a greater degree of reliability than proofs that are constructed manually.

The specific verifications presented in this thesis are as follows. Chapter 3 describes the verification of a variant of a nonblocking queue algorithm first presented in [MS96b]. This verification is the simplest of those presented in the thesis, and so serves as an introduction to the verification techniques. Additionally, during the verification process, we discovered a

useful optimisation of this algorithm, which is also presented in Chapter 3. Chapter 4 presents a subtle verification using backward simulation. This forms the most interesting part of the verification of a corrected version of the Snark algorithm mentioned in Section 1.2.1. The verification functions as an extended example of the application of backward simulation to nonblocking algorithms.

As we describe in the next section, Chapter 6 in Part II presents an implementation of the LL/SC operation pair. We apply the techniques of Part I to the verification of this algorithm. This verification is large, reflecting the complexity of the algorithm, but is a straightforward application of the techniques presented in Chapter 3. Its main purpose is to provide evidence for the correctness of the LL/SC implementation, and to show that our techniques can be applied to complicated algorithms.

1.2.2 Novel Nonblocking Algorithms

We have described two important limitations that restrict the application of nonblocking algorithms in modern computer systems.

- Many nonblocking algorithms depend on garbage collection to reliably release memory back to the system. (The presence of a garbage collector is assumed in [DFG+00, LMS03a, HLM02a, HHL+06]. Many more examples exist.)
- Many nonblocking algorithms require the LL/SC operations, or a CAS operation that can compare-and-swap both a pointer and an adjacent version number, in order to overcome the ABA problem. Such operations are not available on many systems.

We encountered both these issues in Section 1.1.3. Significant research has been conducted into schemes that enable memory reclamation from nonblocking data structures, various LL/SC implementations, and alternative solutions to the ABA problem [Val94, AM95, Moi97, Gre99, Moi00, DMMm01, HLM02b, JP03, Mic04, Jay05].[5] Part II of this thesis presents our contributions to this effort. In Chapter 5 we present a novel nonblocking *reference-counting* technique that enables memory reclamation from nonblocking data structures. This technique has certain advantages over prior proposals, relating to its overall resource consumption. However, this technique requires a CAS operation that can compare-and-swap both a pointer and a counter. As with version-numbering, this reference counting technique cannot be applied on all systems.

[5] A correction to an error in [Val94] is presented in [MS95].

The main result described in Chapter 6 overcomes this problem. We present an implementation of the LL/SC operation pair that may be applied to arbitrarily sized data, that requires only a CAS operation that can atomically modify a pointer value. This implementation can be used with the reference-counting technique to enable nonblocking memory reclamation from many dynamic sized nonblocking data structures. Further, the LL/SC implementation itself can be used in any algorithm that requires the LL/SC operations with their full semantics, or in algorithms that require a CAS or LL/SC operations applicable to both a pointer and other data, such as a version number. Thus, such algorithms can be used even in systems where LL/SC operations, or a CAS operation that can modify multiple values, would otherwise be unavailable.

Similar claims could be made for previous approaches to memory reclamation and solutions to the ABA problem. However, existing solutions have one of two major drawbacks:

- They depend on very unusual properties of the underlying system: either an exotic synchronisation primitive such as DCAS (as in [DMMm01]) or memory blocks not changing layout after reclamation (as in [Val94, Gre99]).
- They require that the maximum number of processes that will ever use the facility be known in advance (eg., [Her91, AM95, LMS03a, JP03, HLM02b, Mic04, LMS03a, JP03]).

These drawbacks are discussed in more detail in Chapters 5 and 6. However, it is worth noting here that it is common for nonblocking algorithms to require that the maximum number of processes be known in advance (eg., [Her91, AM95, LMS03a, JP03]). In fact, the results presented in Chapters 5 and 6 are the first nonblocking algorithms that allocate and release dynamic memory and do not depend on exotic synchronisation primitives or knowledge of the maximum number of processes that will access the data structure.

1.3 Notation

This section describes notation used in the thesis.

Logic, Sets and Functions

We use the standard logical connectives, listed here in order of increasing binding power: $\forall$ for ‘for all’; $\exists$ for ‘there exists’; $\Rightarrow$ for implies; $\vee$ for ‘or’; $\wedge$ for ‘and’; $\neg$ for ‘not’. These

binding conventions are the same as those used in PVS [COR+95]. The scope of bound variables extends to the end of the expression following the quantifier, and we use a dot notation to separate quantifier and predicate. Thus, in

$$\forall x \bullet P \Rightarrow Q(x)$$

x is bound over the predicate Q.

We use $\mathbb{N}$ to denote the natural numbers, $\mathbb{Z}$ to denote the integers and $bool$ to denote the booleans $\{true, false\}$. We use ranges of the form $[i \ldots j]$ to denote the set of integers k such that $i \leq k$ and $k \leq j$. $S \times T$ is the Cartesian product of sets S and T. The projections π_1 and π_2 return the first and second members of these products, respectively. Expressions of the form

$$\prod_{s \in S} e$$

where e is some set expression that may involve s, denotes the product of the sets e across the index set S. For products like this, we use the projections π_s for each $s \in S$.

For complex products, we often use mnemonic access names with a dot syntax. For example, for some tuple $t \in X \times (Y \times Z)$ we might stipulate that $t.x = \pi_1(t)$, $t.y = \pi_1(\pi_2(t))$ and $t.z = \pi_2(\pi_2(t))$.

Given a relation $R : S \times T$ and $s \in S$, $R[s]$ is the relational image of s onto T:

$$R[s] = \{t \in T \mid R(s,t)\}$$

We often need to modify the value of a function at a certain point: given a function $f : S \to T$, $s \in S$ and $t \in T$, let $f \oplus \{s \mapsto t\}$ be the function just like f, but with $f(s) = t$, i.e., for every $s' \in S$:

$$f \oplus \{s \mapsto t\}(s') = \begin{cases} f(s') & \text{if } s' \neq s \\ t & \text{if } s' = s \end{cases}$$

Finally, given a function f, let **dom** f be the domain of f.

Sequences

We make substantial use of sequences. We view sequences as functions over some (possibly infinite) prefix of $\mathbb{N}$ (so sequences are indexed from zero). When a sequence is a function over a finite prefix of $\mathbb{N}$, we say that the sequence is *finite*. Otherwise, we say that it is *infinite*.

We use $length(\alpha)$ to denote the length of the finite sequence α.[6] When $length(\alpha) = 0$, we say that α is empty. By *the empty sequence*, we mean the unique sequence $\langle\rangle$ such that $length(\langle\rangle) = 0$. Given two (finite or infinite) sequences α and β we say that α and β have the *same length* if and only if α and β are both infinite, or $length(\alpha) = length(\beta)$. Sometimes we need to quantify over the domain of a sequence, excluding its last element if it is finite. Thus, given a sequence α, let **dom**$^{-}\alpha$ be **dom** α when **dom** $\alpha = \mathbb{N}$ (i.e., α is infinite), and all but the greatest element of **dom** α when α is finite. Given some set *S*, let S^* be the set of finite sequences whose values are elements of *S*, and let S^∞ be the set of (finite or infinite) sequences whose values are elements of *S*.

[6] Precisely, when α is finite, $length(\alpha)$ is the size of the graph of α.

Part I

The Verification of Nonblocking Algorithms

Chapter 2

Verification

This chapter describes a formal methodology for verifying concurrent algorithms using transition systems. This approach is based on the work of Lynch *et al.* [LT87, LV93, Lyn96], and developed from previous work in the verification of nonblocking algorithms [Doh03, CDG05].

In Section 2.2, we define *linearisability* [HW90], the notion of correctness that we apply to nonblocking algorithms in this thesis. Linearisability is a correctness condition for concurrent implementations of objects (such as stacks and queues) that have a sequential specification. As mentioned in the introduction, we use transition systems called *I/O automata* [LT87] to model the specifications and implementations that we use in this work. I/O automata are described in Section 2.3.

Transition systems, such as I/O automata, are a natural choice for modelling, specifying and verifying concurrent systems. Section 2.1 outlines the reasons for this, and describes some of the advantages of the I/O automaton model. We also discuss how our use of I/O automata relates to the goal of constructing proofs of correctness that are mechanically checkable.

Section 2.4 defines *simulation relations*. A simulation relation is a relation between the states of two automata, the existence of which guarantees that one automaton implements the other automaton. Section 2.5 defines some notation for describing I/O automata. Section 2.6 shows how to construct simple specification automata that are known to have the desired correctness property, linearisability.

2.1 Transition Systems and Verification

Transition systems are frequently used to provide mathematical models of concurrent systems (such as nonblocking algorithms) ([CM88, Sha93, Lyn96, AHR00, Lam94] provide examples in different settings). Briefly, transition systems are structures with a set of *states* (sometimes called its *state space*), a set of *initial* states, and a *transition relation* between states. The use of transition systems is appealing when the algorithm being verified has a notion of *state*, as with a shared-memory nonblocking algorithm.

Labelled transition systems (LTS) are transition systems where each transition has a *label*. Labels are used to distinguish between *internal* transitions (modelling steps in a computation) that are "invisible" and those that are externally visible (modelling invocations and responses of operations).

Specifying an LTS amounts to specifying the properties of its externally observable behaviour: that is, the sequences of external labels that it can produce. LTSs themselves can be viewed as specifications of external behaviours. Thus, we can use an *abstract* LTS as a specification of a *concrete* LTS, that represents the behaviour of an implementation. This is the approach used in this thesis.

The size of a transition system's state space partly determines the difficulty of verifying the system's properties. As we discuss in Section 2.1.2, if a system has a "small" finite state space, then many important questions about the behaviour of the system can be answered automatically. On the other hand, if the system has infinitely many states, verifying its properties can be very challenging. In this thesis, we wish to verify systems that have an unbounded number of processes, sharing an unbounded amount of dynamically allocated memory. Thus, the systems of interest to us have infinite state spaces.

We use *simulation relations* [LV93] to show that an algorithm meets its specification. Simulation relations are relations over the states of two LTSs. The existence of a simulation relation from one LTS to another guarantees that the observable behaviour of the first is shared by the second.

Simulation relations have a very useful property: they reduce reasoning about all possible behaviours of the LTS to reasoning about the individual transitions. In this respect they are akin to proofs relying on invariants, which reduce reasoning about all possible states of an LTS to reasoning about transitions. This *locality* of proof obligations makes reasoning about a large class of possibilities tractable.

2.1.1 Mechanical Assistance for Verification

There are two main kinds of mechanical assistance available for the formal verification of transition systems: model checking and theorem proving. We discuss each in turn.

2.1.2 Model Checking

Model checking [CE82, CES86, QS82] is a verification technique based on generating a representation of the reachable states of a transition system. This representation must allow us to determine mechanically whether some state fails to satisfy some given property. Numerous model checkers are available (e.g., [Spi, SMV, dSp, YML99, Mur]). Modern model checkers can explore large finite state spaces. This makes them capable of automatically verifying properties of a broad range of finite systems. Moreover, it is possible to model check finite instances of systems with unbounded or infinite sets of reachable states. For example, an instance of a concurrent algorithm that uses shared memory can be verified automatically using model checking, so long as the instance in question uses a small, fixed amount of memory, and has a small number of processes. Indeed, the algorithms discussed in Chapters 3 and 6 were model checked (using the model checker Spin [Hol97]) during their development or verification. Moreover, bugs were found in the early versions of the the LL/SC algorithm of Chapter 6. However, checking a finite instance of an algorithm is a long way from providing a general verification, so other techniques must be examined.

Model checkers can only generate a finite representation of a set of reachable states. Therefore, if we wish to verify infinite systems, we must find some way to represent the infinite set of reachable states finitely. Such finite representations are known as *abstractions*. Substantial attention has been given to developing ways to construct abstractions (e.g. [DD02, GS97, BCDR04, MYRS05], but there are many more examples). Some of this work has been directed towards the verification of systems involving concurrent access to shared memory (e.g., [Yah01, WS02, ARR$^+$07a]). We discuss some of these contributions in detail in Chapter 3. In general, obtaining a precise finite representation of the infinite state space of such systems is a very difficult problem. Many techniques generate an abstraction that is an *over-approximation* of the system in question. That is, the abstraction may represent more states than are reachable by the system, or generate a representation of behaviours that do not belong to the system. Often, such over-approximations cannot be used to verify that a system has a property of interest, even when the system does have the property.

2.1.3 Theorem Proving

One of the advantages of using rigorous mathematical models and specifications is that proof obligations can be submitted to a mechanical theorem prover. A mechanical theorem prover is an application capable, at least, of checking proofs of theorems expressed in some kind of formal notation. Most provers have some ability to *construct* proofs, using heuristic-driven, automated proof search and decision procedures. Unlike model checkers, theorem provers can be readily used to verify properties of systems of unbounded size.

Like model checkers, there are several theorem provers available (for example [PVS, LP , isa, met]). Most provide an input language based on mathematical logic and some mechanical proof automation. The verifications presented in this thesis have been checked using the PVS proof assistant [COR+95]. PVS is widely used in academia and industry, provides an easily learned higher-order logic[1] with powerful constructs and is well supported by developers.

The use of a mechanical theorem prover offers several advantages over the construction of proofs by hand. Automated proof search relieves the human of much of the responsibility for carrying out tedious, mechanical reasoning. The PVS system can carry out simple quantifier instantiation and propositional reasoning automatically, as well as applying lemmas based on reasonable heuristics. PVS also has sophisticated decision procedures for equational logic and pure boolean expressions. In combination, these features mean that a user of the PVS system can submit most simple proof goals to the PVS prover, with good reason to hope that they can be proved without any human intervention.

Proofs are checked with mechanical precision. In the ideal case, steps in an argument are matched against the rules of the logic that the prover supports. However, the use of decision procedures in a theorem proving system complicates this issue somewhat: the mechanically checked proof may rely on the correctness of decision procedures that do not explicitly represent applications of proof rules. Still, in the PVS system, these decision procedures are implementations of well-understood algorithms. While these implementations may contain bugs, successfully checking a proof using PVS provides a high level of assurance that the proof is correct and complete.

The main difficulty in conducting a verification by proving theorems, using a proof assistant to check or construct the proofs, is the high level of human involvement. In many cases, model checkers are able to eliminate all, or almost all, need for human insight. The human theorem prover must express the correctness conditions of the system in question, state lem-

[1] Where quantification over functions is allowed.

mas and invariants that are necessary for the proof, and (at least) guide the prover through the process of constructing the proofs.

2.2 Correctness

In order to prove that an implementation of a data structure is correct, we must be able to state precisely the correctness conditions for that implementation: that is, we must be able to *specify* them. In this thesis, we focus on concurrent implementations of datatypes with a clear sequential specification. Stacks and queues are examples of such datatypes. In this setting, a natural way to specify the behaviour of a concurrent datatype is to transform a sequential specification of the datatype into a concurrent one. This is the approach taken by the dominant correctness condition for concurrent implementations of sequential data structures: *linearisability*.

In essence, *linearisability* [HW90] requires that there be some point between the invocation and response of each operation on a concurrent data structure, called a *linearisation point*, when the operation appears to all processes to take effect. The linearisation points form a sequence of operations on the object that must conform to the object's sequential specification. This correctness condition has become standard in the nonblocking algorithms literature. One of the reasons why linearisability has become popular is because it is a *local* property [HW90]: that is, a system of linearisable implementations is linearisable exactly when each implementation within that system is linearisable.

The remainder of this section formally defines linearisability. Section 2.2.1 defines a notion of *sequential datatype*, and Section 2.2.2 defines linearisability in terms of this definition.

2.2.1 Sequential Datatypes

We view a sequential datatype as a specification of a set of valid *behaviours*, where a behaviour is a sequence of operations of the datatype, and responses to those operations. What follows is a simple way to define datatypes formally, adapted from [Lyn96, Section 9.4]. Each datatype is equipped with a set of invocations and responses, that constitute the interface to the datatype. The behaviours of the datatype, which we call *traces* are sequences of pairs invocations and responses.

A datatype $\mathcal{D}$ is a tuple (D, D_0, I, R, u) where D is the set of *values* of the datatype;

$D_0 \subseteq D$ is the set of initial values; I is the set of invocations; R is the set of responses; and $u : D \times I \to D \times R$ is an *update function* that defines how the datatype responds to invocations. The update function u defines the effect of these operations on members of the datatype. We model a behaviour as a sequence of invocation/response pairs. For a datatype $\mathcal{D}$ with invocations I and responses R, define the *sequential alphabet* of $\mathcal{D}$ to be $alpha(\mathcal{D}) = I \times R$.

Definition 2.1 (Execution of datatype)
An *execution* of a datatype $\mathcal{D} = (D, D_0, I, R, u)$ is a sequence $e \in alpha(\mathcal{D})^*$ such that $e_0 \in D_0$, and for every $n \in \mathbf{dom}^- e, i \in I, r \in R$, $u(e_n, i) = (e_{n+1}, r)$.

We now define a notion of the externally observable behaviour of a datatype. A *trace* of a datatype is a sequence of pairs of invocations and responses that corresponds to some execution of the datatype.

Definition 2.2 (Trace of datatype)
A *trace* of a datatype $\mathcal{D}$ is a sequence $t \in alpha(\mathcal{D})^*$ such that there exists some execution e of $\mathcal{D}$ satisfying $\mathbf{dom}^- e = \mathbf{dom}\ t$, and for every $n \in \mathbf{dom}^- t$, $u(e_n, \pi_1(t_n)) = (e_{n+1}, \pi_2(t_n))$.

As an example of this specification style, consider the *stack* datatype. The stack contains elements of some non-empty set T. It is modelled as a sequence of elements from that set. Let the stack datatype be $\mathcal{S} = (D, D_0, I, R, u)$ where:

- $D = T^*$ is the set of sequences of elements from T.
- $D_0 = \{\langle\rangle\}$, the set containing just the empty sequence.
- $I = \{push\_inv(t) \mid t \in T\} \cup \{pop\_inv\}$ and

  $$R = \{push\_resp, empty\} \cup \{pop\_resp(t) \mid t \in T\}$$

  $push\_inv(t)$ represents an invocation of the push operation with the parameter t; pop_inv represents an invocation of the pop operation; $push\_resp$ signals that a push operation has been completed; $pop\_resp(t)$ represents the response to a pop_inv invocation, with the return value t; $empty$ signals that an attempted pop operation found the stack empty.
- The left side of the sequence is the top of the stack so that in response to a push, we want to concatenate the pushed value onto the left side of the sequence. For a pop, unless the stack is empty, we want to remove and return the leftmost value in the

sequence; if the stack is empty, we should do nothing to its state, but return $empty$ as a response. Hence, for any $d \in D$, $t \in T$, the update function u satisfies:

$$u(v, push\_inv(t)) = (\langle t \rangle \frown v, push\_resp)$$
$$u(\langle\rangle, pop\_inv) = (\langle\rangle, empty)$$
$$u(\langle t \rangle \frown v, pop\_inv) = (v, pop\_resp(t))$$

Stacks have traces like

$$\langle (push\_inv(t_1), push\_resp), (pop\_inv, pop\_resp(t_1)), \ldots \rangle$$

which has the following execution

$$\langle\ \langle\rangle, \langle t_1 \rangle, \langle\rangle \rangle$$

2.2.2 Linearisability

We now turn to the definition of linearisability. Linearisability is due originally to Herlihy and Wing [HW87, HW90], and has become a very common correctness condition for concurrent objects. The idea is to make it look to each process (and the observer) as though each operation on a concurrent implementation of a datatype occurs between the invocation and response of the operation, one at a time in an order consistent with the sequential specification of the datatype. The formal definition presented here is adapted from [HW87] and [Lyn96].

Linearisability depends on a notion of *history*. A history is a representation of a sequence of interactions between a set of processes and a concurrent implementation of a datatype, and corresponds to the notion of a *trace* of a datatype. In the definition of *trace* from the previous section, we represented each operation as an invocation/response pair. However, in a concurrent setting, each operation takes place over some interval, so the invocation and response of each operation may not be *adjacent* in any sense. Therefore, we model a concurrent operation as an interval demarcated by an invocation at the beginning and a response at the end. Along with each invocation or response, we record the process that is executing the operation. We need several definitions before we arrive at the definition of *history*.

Definition 2.3 (Concurrent alphabet)
Given a datatype $\mathcal{D} = (D, D_0, I, R, u)$ and a set $PROC$ (whose members are called *processes*), the *concurrent alphabet of* $\mathcal{D}$ *for* $PROC$, written $alpha(\mathcal{D}, PROC)$ is the set $(I \cup R) \times PROC$.

Normally we write elements of the concurrent alphabet as an invocation or response subscripted by a process, so that $(inv, p) \in alpha(\mathcal{D}, PROC)$ becomes inv_p.

We are only interested in sequences over the concurrent alphabet of a datatype that could be generated by a system in which after making an invocation on an instance of the datatype, no process makes another invocation before receiving a response. We call such sequences *well-formed*. We first define *process subhistory*, which, for a given process *p* is the sequence of invocations and responses performed by *p*; then we define *well-formedness* and *history*.

Definition 2.4 (Process subhistory)
Given a datatype $\mathcal{D}$, process set $PROC$, and sequence $s \in alpha(\mathcal{D}, PROC)^*$, the *process subhistory for $p \in PROC$ in s*, written $s \mid p$, is the sequence of invocations and responses in *s* that are indexed by *p*.

Definition 2.5 (Well-formedness)
Given a datatype $\mathcal{D}$ and process set $PROC$, a sequence $s \in alpha(\mathcal{D}, PROC)^*$ is *well-formed* if for every $p \in PROC$, $s \mid p$ begins with an invocation, and for every $n \in \mathbf{dom}^-(s \mid p)$, if $(s \mid p)_{n+1}$ is a response, then $(s \mid p)_n$ is an invocation.

Definition 2.6 (History)
Given a datatype $\mathcal{D}$ and process set $PROC$, a *history of $\mathcal{D}$ and $PROC$* is a well-formed sequence $h \in alpha(\mathcal{D}, PROC)^*$.

We define an operation in a history *h* to be a triple $(n, inv_p, resp_p)$ where *p* is a process, $h_n = inv_p$ and $resp_p$ is the next *p*-indexed response after the invocation inv_p in the history.[2] Some invocations may not have matching responses. These invocations are called *pending*.

Definition 2.7 (Operation, pending invocation, complete history)
Given a datatype $\mathcal{D}$ with invocations *I* and responses *R*, process set $PROC$, and execution history *h* of $\mathcal{D}$ and $PROC$, an *operation in h* is a triple $(n, inv_p, resp_p)$ with $h_n = inv_p$, $inv \in I$ and $resp \in R$ such that there is some *k* satisfying $(h \mid p)_k = inv_p$ and $(h \mid p)_{k+1} = resp_p$. A *pending invocation in h* is a pair (n, inv_p) where $inv_p \in I \times PROC$ such that $h_n = inv_p$ is the last element of $h \mid p$. The sequence $complete(h)$ is *h* with all pending invocations removed.

Now, a history *h* induces a natural partial order over its operations, denoted $<_h$.

[2] The first component of an operation is used to distinguish it from other operations in the history that have the same invocation and response.

Definition 2.8 (Irreflexive partial order of a history, $<_h$)
Given a datatype $\mathcal{D}$, process set $PROC$ and execution history h of $\mathcal{D}$ and $PROC$, $<_h$ is the irreflexive partial order over the operations of h defined by

> $(m, inv_p, resp_p) <_h (n, inv'_q, resp'_q)$ if and only if there exists some k, such that $m < k < n$ and $h_k = resp_p$.

We will extend the partial orders of histories to total orders, and then use these total orders to construct traces of datatypes. This construction relies on some simple observations about total orders of operations. First, observe that an irreflexive total order $<$ over a set of operations S, such that $<$ has a least element or S is empty, induces a sequence of operations from S. If $<$ has a least element, this sequence is constructed by laying out the operations of S in the order determined by $<$; if S is empty, then the sequence is $\langle\rangle$. Second, observe that, given a history h of datatype $\mathcal{D}$, such that h has some operation, any total order over the operations of that history has a (not necessarily unique) least element if it contains the irreflexive partial order $<_h$. (The operation $(n, inv_p, resp_p)$ such that $resp_p$ is the first response in h is a least element.) Thus, given a history h, any total order (empty or not) over the operations of h that contains $<_h$ induces a sequence of operations. Finally, this sequence of operations induces a sequence of invocations and responses of $\mathcal{D}$ constructed by laying out in order the pairs made up of the invocations and responses of each operation in the sequence, with the process index removed.

We now define linearisability.

Definition 2.9 (Linearisability of histories)
A history h of datatype $\mathcal{D}$ and set $PROC$ is *linearisable* if it can be extended to a history h' by appending elements of $alpha(\mathcal{D}, PROC)$, such that there exists an irreflexive total order $<$ over the operations of $complete(h')$ satisfying the following conditions:

1. The partial order $<_{h'}$ is contained in the total order $<$. That is, for every pair of operations $\mathcal{O}_1, \mathcal{O}_2$ in h', $\mathcal{O}_1 <_{h'} \mathcal{O}_2$ implies $\mathcal{O}_1 < \mathcal{O}_2$.

2. The sequence of invocations and responses induced by $<$ is a trace of $\mathcal{D}$.

There are two sets of decisions which must be made to show that a given history h can be linearised: the choice of the extension h' and the construction of the total order $<$. An example should illuminate how these choices should be made. With reference to the stack

datatype $\mathcal{S}$ introduced in the previous section, consider the following history[3]:

$$h = \langle push\_inv_p(t_1), push\_inv_q(t_2), pop\_inv_r, push\_resp_q, \\ pop\_resp_r(t_2), pop\_inv_r, pop\_resp_r(t_1)\rangle$$

Figure 2.1 illustrates this history.

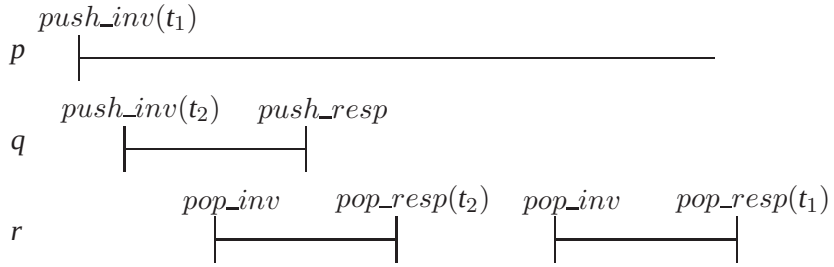


Figure 2.1: The operations of the example trace.

There are three operations in *h*:

$$\mathcal{O}_1 = (1, push_q(t_2), push\_resp_q)$$
$$\mathcal{O}_2 = (2, pop_r, pop\_resp_r(t_2))$$
$$\mathcal{O}_3 = (5, pop_r, pop\_resp_r(t_1))$$

Also, there is one pending invocation: $(0, push_p(t_1))$. Note that the response of $\mathcal{O}_3$ returns the value t_1. This value can only have been placed on the stack by process *p* during its push operation, so we cannot construct a trace of the stack datatype from the operations $\mathcal{O}_1$, $\mathcal{O}_2$ and $\mathcal{O}_3$, however they are ordered. Hence, we cannot construct a total order on the operations of *h* to satisfy Definition 2.9. We need to extend *h* to a history h' such that *p*'s pending invocation becomes an operation. In that case, h' will have a fourth operation, *p*'s push. So define

$$h' = \langle push_p(t_1), push_q(t_2), pop_r, push\_resp_q, \\ pop\_resp_r(t_2), pop_r, pop\_resp_r(t_1), push\_resp_p\rangle$$

and let $\mathcal{O}_4 = (0, push_p(t_1), push\_resp_p)$. Note that $\mathcal{O}_4$ is an operation of h'.

Because there are no pending invocations in h', $complete(h') = h'$. All we need to do now is construct the order $<$ to satisfy clause (1) of Definition 2.9. We do this by choosing

[3]Strictly speaking, we should write the process-indexed invocations of push operations in the form $push\_inv(t)_p$, and similarly for other invocations or responses that have arguments. The form used here seems more natural, and we use it throughout this thesis.

a *linearisation point*, for each operation in h'. This is a point in the interval between the invocation and response of each operation where we can think of the operation as 'taking effect'. Once a linearisation point has been assigned to each operation, we let $\mathcal{O}_i < \mathcal{O}_j$ if the linearisation point of $\mathcal{O}_i$ occurs before the linearisation point of $\mathcal{O}_j$. Thus, the order of linearisation points induces a total order on the operations of a history.

Since the response of $\mathcal{O}_2$ returns the value t_2 we need to place $\mathcal{O}_1$'s linearisation point before that of $\mathcal{O}_2$ (because t_2 needs to be in the stack for $\mathcal{O}_2$ to be able to return the value). Also, $\mathcal{O}_3$ returns t_1, the value pushed by the pending invocation $push_p(t_1)$, so we should choose a linearisation point for $\mathcal{O}_4$ before that of $\mathcal{O}_3$. Therefore, we must choose linearisation points so that the following conditions are satisfied

$$\mathcal{O}_1 < \mathcal{O}_2,\ \mathcal{O}_4 < \mathcal{O}_3$$

Figure 2.2 illustrates one possibility for a set linearisation points consistent with these constraints.

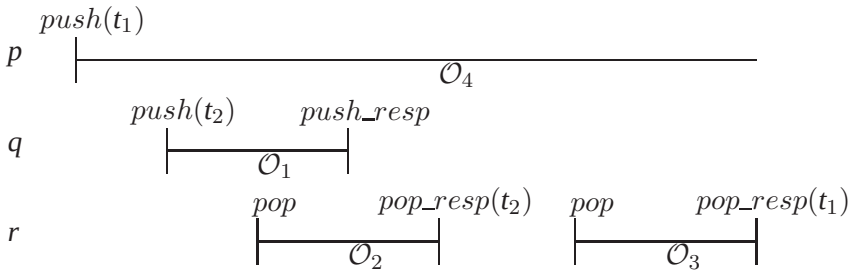


Figure 2.2: The operations of the example trace.

This set of linearisation points induces the following order on the operations of h':

$$\mathcal{O}_1 < \mathcal{O}_2 < \mathcal{O}_4 < \mathcal{O}_3$$

This order induces the following trace of the stack datatype:

$$\langle (push(t_2), push\_resp), (pop, pop\_resp(t_2)), \\ (push(t_1), push\_resp), (pop, pop\_resp(t_1)) \rangle$$

This is the trace of the following execution of $\mathcal{S}$:

$$\langle\ \langle\rangle, \langle t_2 \rangle, \langle\rangle, \langle t_1 \rangle, \langle\rangle\ \rangle$$

There is another possible choice for the linearisation point chosen for the pending push, $\mathcal{O}_4$: we could have stipulated that it occurred before the linearisation point of $\mathcal{O}_1$ and still

obtained a valid linearisation. There is often a substantial degree of freedom in choosing the linearisation points for a history.

Note that choosing linearisation points between the invocations and responses of each operation guaranteed that the resulting order contained $<_h$ (so we satisfied clause (2) of Definition 2.9).

Note that we have defined linearisability only for individual histories. We need to extend this definition to cover concurrent implementations of datatypes. We model concurrent objects, both implementations and specifications, as *I/O automata* [SAGG$^+$93], which are described in the next section. In this thesis, an I/O automaton is a labelled transition system whose observable behaviour is defined to be a set of histories. We say that an I/O automaton is linearisable if and only if every member of its set of histories is a linearisable history.

2.3 I/O Automata

The following definitions are adapted from definitions found in [LT87, SAGG$^+$93, Lyn96].

Definition 2.10 (I/O Automaton)
An *I/O automaton* is a tuple $(\mathit{external}, \mathit{internal}, \mathit{states}, \mathit{start}, \mathit{trans})$, where *external* is a nonempty set of *external actions*; *internal* is a set of *internal actions* such that $\mathit{external} \cap \mathit{internal} = \varnothing$; *states* is a set of states (sometimes called the *state space* of the automaton); $\mathit{start} \subseteq \mathit{states}$ is a nonempty set of start states; and $\mathit{trans} \subseteq \mathit{states} \times \mathit{acts} \times \mathit{states}$ is the transition relation, where $\mathit{acts} = \mathit{external} \cup \mathit{internal}$.

The definition of I/O automata given in [Lyn96] separates the set *external* into sets $Input$ and $Output$. As described in Section 2.6, we have no need of this distinction.

We define some helpful notation to describe I/O automata. Given an I/O automaton $A = (\mathit{external}, \mathit{internal}, \mathit{states}, \mathit{start}, trans)$, let $\mathit{external}_A = \mathit{external}$, $\mathit{internal}_A = \mathit{internal}$, $\mathit{states}_A = \mathit{states}$, $\mathit{start}_A = \mathit{start}$ and $\mathit{trans}_A = \mathit{trans}$. Also, let $\mathit{acts}_A = \mathit{external}_A \cup \mathit{internal}_A$. When $(s, a, s') \in \mathit{trans}_A$ we write $s \xrightarrow{a}_A s'$, or $s \xrightarrow{a} s'$ when no confusion is possible. If $s \xrightarrow{a}_A s'$ we may refer to s as the *pre-state* of the transition, and s' as the *post-state*.

We define a notion of of *trace* for I/O automata in a way similar to our definition for datatypes. Just as with datatypes, the set of traces of an automaton constitutes its *behaviour*.

We begin with two preliminary definitions: *execution fragments*, *executions*, and *move* .

Definition 2.11 (Execution fragment)
An *execution fragment* of an I/O automaton A is a sequence $\alpha \in states_A^*$ such that for all $n \in \mathbf{dom}^- \alpha$, there exists some $a \in acts_A$ such that $\alpha_n \xrightarrow{a} \alpha_{n+1}$.

An *execution* of an automaton is an execution fragment that begins with a start state.

Definition 2.12 (Execution)
An *execution* of an I/O automaton A is an execution fragment $\alpha \in states_A^*$ such that $\alpha_0 \in start_A$.

The set of executions of an automaton A is denoted $execs_A$.

Definition 2.13 (Move)
An I/O automaton A *moves* from $s \in states_A$ to $s' \in states_A$ via $\mu \in acts_A^*$, written $s \stackrel{\mu}{\Longrightarrow}_A s'$, iff $s = s'$ and $\mu = \langle\rangle$, or there exists some execution fragment α of A satisfying $\mathbf{dom}^- \alpha = \mathbf{dom}\ \mu$, the first element of α is s, the last s', and such that for all $n \in \mathbf{dom}^- \alpha$, $\alpha_n \xrightarrow{\mu_n} \alpha_{n+1}$.

Now we are ready to define the *traces* of an automaton. Traces constitute the observable behaviour of an automaton, in an analogous way to the traces of a datatype. However, unlike with datatype traces, automaton traces depend on the classification of actions into internal and external: only the external actions are observable in a behaviour. This allows us to model situations in which concurrent implementations make changes to state that are not observable externally. For any automaton A and sequence $\mu \in acts_A^*$, let $trace_A(\mu)$ be the sequence of external actions of A occurring in μ.

Definition 2.14 (Trace)
A sequence $\mu \in external_A^*$ is a trace of automaton A iff there exists some $\nu \in acts_A^*$ such that $trace_A(\nu) = \mu$, and there exists some $s \in start_A$, $s' \in states_A$ such that $s \stackrel{\nu}{\Longrightarrow}_A s'$.

The set of all traces of an automaton is denoted $traces_A$, and constitutes the observable behaviour of the automaton A. We define a relation between automata called *trace inclusion*, denoted $\leq_T$, as follows:

Definition 2.15 (Trace inclusion, Finite trace inclusion)
For any I/O automata A and B, $A \leq_T B$ iff $traces_A \subseteq traces_B$, and $A \leq_{T^*} B$ iff for every $\mu \in traces$, if μ is finite, then $\mu \in traces_B$.

For any automata A and B, $traces_A \leq_T traces_B$, then any behaviour exhibited by A can also be exhibited by B. Therefore, if B is correct with respect to some specification defined

in terms of traces and $A \leq_T B$, then B is correct with respect to that specification as well. Likewise, if $A \leq_{T*} B$, then A is correct with respect to the safety properties specified by B. In this thesis, we focus on the verification of safety properties.

Note that trace inclusion (finite or not) is a *pre-order*: that is, it is reflexive and transitive. Because trace inclusion is a pre-order, we can reason hierarchically. Given a specification automaton A and implementation automaton C, we can show that $C \leq_T A$ if we can find an intermediate automaton I such that $C \leq_T I$ and $I \leq_T A$. This is often a very useful strategy, which we employ in Chapters 3 and 4.

During our verifications, we construct proofs that $A \leq_T B$ or $A \leq_{T^*} B$, for given automata A and B. B provides the specification and A models the implementation. We call our specification automata *abstract automata*, and our implementation automata as *concrete automata*.

2.4 Verifying Trace Inclusion

This section describes formal techniques for proving that the traces of one automaton are included within the traces of another. Our approach is built around the use of *simulation relations*: relations between the states of automata that satisfy certain properties, the existence of which guarantees trace inclusion between the automata. We first define *reachable states* and *invariants*, then we discuss simulation relations.

One subset of the state-space of an automaton is particularly important: the *reachable* states of the automaton. This is the set of states that can appear in an execution of the automaton: i.e., the least set containing the start states and closed under the transition relation.

Definition 2.16 (Reachable states)
For an I/O automaton A, the set of *reachable states*, denoted $reach_A$ is the least set satisfying

1. $start_A \subseteq reach_A$.
2. For all $s, s' \in states_A$, if $s \in reach_A$ and $s \xrightarrow{a} s'$ for some $a \in acts_A$, then $s' \in reach_A$.

An *invariant* of an automaton is a superset of the reachable states of the automaton. Thus, in order to prove that some property P is an invariant, we must show that Clauses 1 and 2 above hold for P. We use invariants in our verifications, but simulation relations are much more important to this thesis.

A *simulation relation* is a relation over the states of two automata with certain properties, the existence of which guarantees that every trace of one automaton is a trace of the other. The existence of a simulation relation between abstract and concrete automata guarantees that the traces of the concrete automaton are also traces of the abstract automaton. This is because a simulation relation allows the construction of an execution of the abstract automaton given an execution of the concrete automaton, such that the abstract execution has the same trace as the concrete execution. One way to think about a simulation relation is that it specifies the sense in which the states of the concrete automaton *represent* the states of the abstract automaton. The simulation relations used in the following chapters will help to illuminate this idea.

There are several different kinds of simulation relation, differing from one another in their range of applicability and complexity. [LV93] provides a good survey of the classes of simulation relations available. In this section we define two kinds of simulations relation: *forward simulation* and *backward simulation*. Different notions of forward simulation are used in many verification contexts (for example [HHS86, CM88, WD96, Abr96, dREB98]) and the technique is well understood. Although backward simulation exists in several formalisms, it is not often applied. However, it is required in two of the verifications presented in this thesis.

2.4.1 Forward Simulation

The following definition is adapted from [SAGG$^+$93, Lyn96].

Definition 2.17 (Forward Simulation)
Given automata A and C such that $external_A = external_C$, a *forward simulation* R from C to A is a relation over $states_C \times states_A$ satisfying:

1. For all $s_C \in start_C$, there is some $s_A \in start_A$ such that $R(s_C, s_A)$.
2. For all $s_C, s'_C \in reach_C$, and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$, then for all s_A such that $R(s_C, s_A)$, there is some $s'_A \in states_A$ and execution fragment β of A such that $R(s'_C, s'_A)$, $s_A \overset{\beta}{\Longrightarrow} s'_A$ and $trace_A(\beta) = trace_A(a)$. Note that β may be the empty execution fragment.

The automaton A in the above definition is the abstract automaton; the automaton C is the concrete automaton.

The existence of a forward simulation between A and C allows us to construct for any execution of C, an execution of A with the same trace. We do this by an induction on the

length of executions of *A* with the hypothesis that: (i) for each execution α of *C* with given length, there is some state of *A* related to the last state α; and (ii), this abstract state can be reached by an abstract execution fragment β such that $trace(\alpha) = trace(\beta)$. 1 above gives us the base case and as the length of executions increases, the hypothesis is preserved by applying 2. These observations are the basis of the proof of the following soundness property:

Theorem 2.1 (Forward simulation implies trace inclusion)
If *R* is a forward simulation from *C* to *A*, in the sense of Definition 2.17, then $C \leq_T A$.

The definition of forward simulation enables us to "record" information about the history of the execution. This is achieved using existentially quantified variables within the simulation relation. We use this technique several times in this thesis.

Note that when using forward simulation, at each step in a concrete execution, we must be able to choose an abstract action or execution fragment to satisfy one of the conditions in Definition 2.17. Because we construct the abstract execution by induction over the concrete execution (beginning at the start of the concrete execution and moving forwards) this choice can only be based on the earlier states of the execution. That is, we can use only the history of the execution, not the future. Sometimes it is impossible to make this choice based only on the execution history (verifications in Chapters 3 and 4 provide examples). Backward simulations, described in the next section, overcome this limitation.

2.4.2 Backward Simulation

The following definition of backward simulation is adapted from [SAGG+93].

Definition 2.18 (Backward Simulation)
Given automata *A* and *C* such that $external_A = external_C$, a *backward simulation R* from *C* to *A* is a relation over $states_C$ and $states_A$ satisfying:

1. For all $s_C \in start_C$, and all s_A such that $R(s_C, s_A)$, $s_A \in start_A$.
2. For all $s_C \in reach_C, s'_C \in states_C$, and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$, then for all s'_A such that $R(s'_C, s'_A)$, there is some $s_A \in states_A$ and execution fragment β of *A* such that $R(s_C, s_A)$, $s_A \stackrel{\beta}{\Longrightarrow} s'_A$ and $trace_A(\beta) = trace_A(a)$. As before, β may be the empty execution fragment.
3. For all $s_C \in reach_C$, there exists some s_A such that $R(s_C, s_A)$.

There are three important differences between Definitions 2.17 and 2.18. First, Condition 1 of Definition 2.18 requires that every abstract state related to a concrete start state be an abstract start state; the corresponding condition in Definition 2.17 requires only that some related abstract pre-state exist. Second, Condition 2 of Definition 2.18 is dual to Condition 2 of Definition 2.17: for forward simulation, we begin with related pre-states and must produce related post-states; for backward simulation, we begin with related post-states and must produce related pre-states. Third, Condition 3 of Definition 2.18 has no analogue in the definition of forward simulation.

To see why these differences exist we must understand how the existence of a backward simulation allows the construction of an abstract execution with the same trace as a given concrete execution. We consider the argument for finite executions; the argument for infinite executions is more technical and can be found in [Lyn96]. Given an execution of C, Condition 3 allows us to choose an abstract state related to the final state of the concrete execution. Condition 2 allows us to construct an abstract execution *backwards* from this state, having the same trace as the given concrete execution. At the end of this process, we choose an abstract state that is related to the concrete start state. Condition 1 guarantees that this abstract state will be an abstract start state. These observations are the basis of the proof that the existence of a backward simulation between two automata implies finite trace inclusion.

Theorem 2.2 (Backward simulation implies finite trace inclusion)
If R is a backward simulation from C to A, in the sense of Definition 2.17, then $C \leq_{T^*} A$.

See [Lyn96] for a proof.

Note that the existence of a backward simulation between two automata only implies finite trace inclusion. The existence of a backward simulation guarantees (finite or infinite) trace inclusion iff the simulation is *image finite*. A relation $R : S \times T$ is image finite iff for every $s \in S$, the set

$$\{t \in T \mid R(s,t)\}$$

is finite.[4]

In this thesis, we verify only safety properties. For this reason, we do not concern ourselves with the image finiteness property. The backward simulation presented in Chapter 3

[4]Image finitness is required to construct infinite abstract traces from infinite concrete executions, using König's lemma [LV93].

is image finite, whereas the backward simulation in Chapter 4 is not. An image finite backward simulation could be constructed along very similar lines to the simulation presented in Chapter 4, but this is unnecessary for the verification of safety properties.

Note that, in contrast with forward simulation, when choosing an abstract action or execution fragment to satisfy 2 we can examine the future of the execution, but not the history. This means that backward simulation can be applied in situations where forward simulation cannot (and *vice-versa*). It turns out that applying both backwards and forwards simulation results in a complete proof method for trace inclusion.

Theorem 2.3
Given automata A and C, if $C \leq_T A$ then there exists some automaton B, such that there is a forward simulation R_F from C to B and an image finite backward simulation R_B from B to A.

See [LV93] for a proof. This technique of using both backward and forward simulation is used in Chapter 3.

In our verifications, when no forward simulation is possible, because it is impossible to choose a step of the specification automaton for a step of the concrete automaton, we say that the concrete automaton exhibits *prophetic linearisation*. This term is meant to suggest that we cannot find a linear order for some of the operations in an execution until after the operations have been completed. Occasionally we speak of *future dependent linearisation points*. These are steps of an algorithm that are sometimes linearisation points, depending on events that happen after the step in question. All algorithms that have future dependent linearisation points exhibit prophetic linearisation, and require backward simulation to verify using the methods presented in this thesis.

2.4.3 One step simulations

The simulation relations just presented are more general than is typically required. In particular, in the verifications presented in Chapters 3 and 6, the execution fragments used as witnesses for Condition 2 of Definitions 2.17 and 2.18 are only ever single actions or the empty sequence. Expressing the general definitions of the simulation relations in a formal logic (such as that of PVS), and reasoning about those definitions introduces needless complexity.

Therefore, we define *one-step* simulations, in which these conditions are replaced with simpler versions. The idea is as follows. When the concrete automaton takes an internal action, the abstract automaton must either take an internal action, or no action (ie., the abstract

prestate must be related to the concrete poststate). When the concrete automaton takes an external action, the abstract automaton must take the same external action.

Definition 2.19 (One step forward simulation)
Given automata A and C such that $external_A = external_C$, a *one-step forward simulation* R from C to A is a relation over $states_C$ and $states_A$ satisfying:

1. For all $s_C \in start_C$, there is some $s_A \in start_A$ such that $R(s_C, s_A)$.
2. For all $s_C \in reach_C$, $s'_C \in states_C$ and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$ and $a \in external_C$, then for all s_A such that $R(s_C, s_A)$, there is some $s'_A \in states_A$ such that $R(s'_C, s'_A)$ and $s_A \xrightarrow{a} s'_A$.
3. For all $s_C \in reach_C$, $s'_C \in states_C$ and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$ and $a \in internal_C$, then for all s_A such that $R(s_C, s_A)$, one of the following is satisfied:
   (a) there is some $s'_A \in states_A$ and action $b \in internal_A$ such that $R(s'_C, s'_A)$, $s_A \xrightarrow{b} s'_A$
   (b) $R(s'_C, s'_A)$.

Definition 2.20 (One step backward simulation)
Given automata A and C such that $external_A = external_C$, a *one-step backward simulation* R from C to A is a relation over $states_C$ and $states_A$ satisfying:

1. For all $s_C \in start_C$, and all s_A such that $R(s_C, s_A)$, $s_A \in start_A$.
2. For all $s_C \in reach_C$, $s'_C \in states_C$ and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$ and $a \in external_C$, then for all s'_A such that $R(s'_C, s'_A)$, there is some $s_A \in states_A$ such that $R(s_C, s_A)$ and $s_A \xrightarrow{a} s'_A$.
3. For all $s_C \in reach_C$, $s'_C \in states_C$ and $a \in acts_C$, if $s_C \xrightarrow{a} s'_C$ and $a \notin external_C$, then for all s_A such that $R(s'_C, s'_A)$, one of the following is satisfied:
   (a) there exists some $s_A \in states_A$ and action $b \in internal_A$ such that $R(s_C, s_A)$, $s_A \xrightarrow{b} s'_A$
   (b) $R(s_C, s_A)$.
4. For all $s_C \in reach_C$, there exists some s_A such that $R(s_C, s_A)$.

2.5 Describing I/O Automata

It is useful to have some notation to describe the states and transition relations of I/O automata. The notation we describe here is modelled closely on the IOA language which is used for describing I/O automata [GLV01, GL00]. The notation allows us to easily describe components of the state space of an automaton, and describe its transition relation.

This section presents a simple I/O automaton that is used to illustrate this notation and provides an example of the modelling style used in this thesis.

Our example is an automaton *A*, that models a stack containing elements in *T* (as defined in Section 2.2.1), concurrently accessed by some set $PROC$ of processes. Its external actions are the concurrent alphabet of the stack datatype defined in Section 2.2.1. Specifically

$$\begin{aligned} \mathit{external}_A = (&\{push\_inv_p(t) \mid t \in T\} \cup \\ &\{pop\_inv, push\_resp, pop\_resp_p\}) \times PROC \end{aligned}$$

Its internal actions label transitions that represent each process actually executing an operation (ie., the linearisation points of the operations), so we have

$$\mathit{internal}_A = \{do\_push, do\_pop\} \times PROC$$

Now, define a set of *program counters* $COUNTER = \{idle\} \cup alpha(\mathcal{D})$. The states of *A* are pairs whose first component is a stack value, and whose second is a tuple of $COUNTER$ values indexed by elements of $PROC$. Letting $D = T^*$ be the set of values of the stack datatype

$$\mathit{states}_A = D \times \Pi_p COUNTER$$

The component $\Pi_p COUNTER$ associates with each process a program counter value that is used to record whether the process is executing an operation and if so, what point in that operation it is up to.

Typically, the set of states of an I/O automaton is a cartesian product, so it is useful to introduce *state variables* to access each element of the state of an automaton. These state variables are just access names for the state type of the automaton. We introduce the state variables pc_p for each $p \in PROC$ and $stack$ where, for any $s \in \mathit{states}_A$, $s.stack = \pi_1(s)$ and, $s.pc_p = \pi_p(\pi_2(s))$. Using this notation, we can define the set of start states of *A*.

$$\mathit{start}_A = \{s \in \mathit{states}_A \mid s.stack = \langle\rangle \wedge \forall p \in PROC \bullet pc_p = idle\}$$

$push\_inv_p(t)$:
pre $pc_p = idle$
eff $pc_p := push(t)$

$pop\_inv_p(t)$:
pre $pc_p = idle$
eff $pc_p := pop$

$push\_resp_p$:
pre $pc_p = push\_resp$
eff $pc_p := idle$

$pop\_resp_p(t)$:
pre $pc_p = pop\_resp(t)$
eff $pc_p := idle$

$do\_push_p(t)$:
pre $pc_p = push(t)$
eff $stack := \pi_1(u(stack, push(t)))$,
$pc_p := \pi_2(u(stack, push(t)))$

do_pop_p :
pre $pc_p = pop$
eff $stack := \pi_1(u(stack, pop))$,
$pc_p := \pi_2(u(stack, pop))$

Figure 2.3: Transition relation of the *Stack* automaton. Recall that u is the update function for the $\mathcal{S}$ datatype.

Several of the automata presented in this thesis have process-indexed variables: these variables always represent the local state of each process, so sometimes we refer to them as *local* variables. We also refer to un-indexed variables as *shared* variables.

We now define the transition relation of A. To do this, we will associate each action with a *precondition* and an *effect* that together specify the transitions labelled by that action. Figure 2.3 presents this association for the stack automaton.

The precondition of each action acts as a guard for the action. The precondition constrains the values taken by state variables in pre-states of transitions labelled by the action. The effect of each action is a set of *parallel assignments*, where the post-state value of the variable on the left-hand side is taken to be the value of the right-hand side expression in the pre-state. Variables not mentioned on the left-hand side of any assignment keep the same value. For example, the precondition and effect associated with the action do_pop_p entail that

$$\begin{aligned} s \xrightarrow{do\_pop_p} s' \Leftrightarrow \quad & s.pc_p = pop \wedge s'.pc_p = \pi_2(u(s.stack, pop)) \ \wedge \\ & s'.stack = \pi_1(u(s.stack, pop)) \wedge \\ & \forall\, q \neq p \bullet s'.pc_q = s.pc_q \end{aligned}$$

Note that, given a pre-state and action there is only one possible post-state: every transi-

tion relation discussed in this thesis has this property. The parallel assignment notation used here is simpler and clearer than a more general relational notation that would be needed to specify systems having a transition relation where there could be more than one post-state for each pre-state and action.

2.6 Specification Automata

As mentioned in the introduction, the approach outlined here uses I/O automata to model both the specifications and implementations of the algorithms that we verify. This section describes how we construct I/O automata to act as specification automata in our verifications. We show how to mechanically construct, from a given datatype, an automaton whose traces are exactly the linearisable traces of that datatype. An automaton constructed by this method is called the *canonical automaton* for that datatype. The construction presented here is based on that presented in [Lyn96].

The construction is very simple. In fact, the stack automaton presented in the previous section is the canonical automaton for the stack datatype. The canonical automata described here model a system of processes executing operations on a shared instance of the given datatype. Initially, no process is executing an operation (we say that every process is "idle"); during the execution, each process repeatedly chooses an operation to invoke, executes the steps of the operation and after producing a response action, returns to its "idle" state.

Fix a datatype $\mathcal{D}$ with values D, initial value d_0, invocations I, responses R and update function u, and a set of processes $PROC$. Let A be the canonical automaton for $\mathcal{D}$ and $PROC$. The actions of A are defined as follows:

$$\mathit{external}_A = alpha(\mathcal{D}, PROC)$$
$$\mathit{internal}_A = \{do\_inv \mid inv \in I\} \times PROC$$

The labels do_inv must be distinct from each other and distinct from everything in $alpha(\mathcal{D})$. The states of A are defined as follows:

$$\mathit{states}_A = D \times \Pi_p COUNTER$$

where $COUNTER = \{idle\} \cup alpha(\mathcal{D})$.

As with the example stack of Section 2.5, we introduce variables to access the components of a state s: let $s.d = \pi_1(s)$ and let $s.pc_p = \pi_p(\pi_2(s))$. The start states of the canonical automaton are as follows:

$$\mathit{start}_A = \{s \in \mathit{states}_A \mid s.d = d_0 \wedge \forall p \in PROC \bullet pc_p = idle\}$$

inv_p :	do_inv_p :	$resp_p$:
pre $pc_p = idle$	**pre** $pc_p = inv$	**pre** $pc_p = resp$
eff $pc_p := inv$	**eff** $d := \pi_1(u(d, inv))$, $pc_p := \pi_2(u(d, inv))$	**eff** $pc_p := idle$

Figure 2.4: Transition relation of the canonical automaton.

The transition relation of the canonical automaton *A* is presented in Figure 2.4. Each identifier inv ranges over the set *I* of invocations, and each $resp$ ranges over the set *R* of responses, so each such action presented in the figure should be thought of as representing a *set* of actions. Each operation is executed in three steps: a process *p* receives an invocation; it then applies that invocation to the shared variable representing the datatype; and finally, it completes the operation by taking a transition labelled by the response to that operation.

The construction of canonical automata presented here differs from the construction presented in [Lyn96, Section 13.2] in certain respects. The use of *process*-indexed invocations and responses differs from the indexing used in [Lyn96]. Lynch uses indices on invocations and responses, but there the interpretation is that the indices represent *ports*. This difference is partly attributable to the fact that [Lyn96] is concerned with distributed systems, whereas here the concern is multi-processor systems. However, there is a more substantive difference between the two constructions of canonical automata, related to the relaxation here of a condition on the I/O automata of [Lyn96] called *input-enabledness*. An input-enabled automaton has a set of external actions, called *input* actions that are enabled in every state. This means that input-enabled automata can *receive* every input from the external environment at all times. However, in a multi-processor system, each process can only invoke an operation when it is *not* in the middle of another operation on the same datatype. Therefore, the canonical automata defined here are *not* input enabled: invocations are enabled when the invoking process is $idle$. Because we do not need to represent a set of input actions that must be enabled, we do not separate *external* actions into $Input$ and $Output$ actions.

Our automata can be thought of as informal compositions of a shared object with its client processes. This approach provides a straightforward way to guarantee that the traces of our automata are actually histories rather than arbitrary sequences over the concurrent alphabet of the datatype. (Recall that histories have the property that each process subtrace starts with an

invocation, and that after an invocation, each process waits for a response before attempting another invocation).

2.6.1 Properties of Canonical Automata

This section presents results concerning linearisability of canonical automata, and explores theoretically their usefulness in the verification of implementations of datatypes. The results in this section are fairly straightforward and all have analogues in [Lyn96]. Their importance lies in showing soundness and the breadth of applicability of the proof method being developed.

For the rest of this section, fix a datatype $\mathcal{D} = (D, D_0, I, R, u)$ and a set of processes $PROC$. Let *A* be the canonical automaton for $\mathcal{D}$ and $PROC$, constructed as in the previous section.

Linearisability of the Canonical Automaton

We now outline the proof that every trace of *A* is a history of $\mathcal{D}$ and $PROC$ and that every trace is linearisable to $\mathcal{D}$.

Note that $external_A = alpha(\mathcal{D}, PROC)$. Therefore, in order to show that every trace of *A* is a history, we only need to show that *A* produces only histories over $alpha(\mathcal{D}, PROC)$. This is true by virtue of the preconditions on each transition.

Lemma 2.1 (The canonical automaton's traces are histories)
All the traces of *C* are histories of $\mathcal{D}$ and $PROC$.

Proof: Consider some trace μ of *C* and process *p*:

- Note that for each $s \in start(C)$, $s.pc_p = idle$ and for each $a \in acts_A$ where *a* is indexed by some $q \neq p$, if $s \xrightarrow{a} s'$ then $s.pc_p = s'.pc_p$. Therefore, the first *p*-indexed action in μ must be an invocation, since the precondition of every other *p*-indexed action requires that $s.pc_p \neq idle$.

- Assume there is an occurrence of a *p*-indexed invocation in μ. Each state *s* appearing in the execution which produced μ after this occurrence will have $s.pc_p \neq idle$ until an occurrence of a *p*-indexed response. Hence if there is an action following the *p*-indexed invocation in μ it must be a response.

- A similar consideration shows that any action following a response in $\mu \mid p$ is an invocation.□

We now outline the proof that every trace of A is linearisable (cf. [Lyn96], Theorem 13.3).

Lemma 2.2 (Linearisability of canonical automaton)
A is linearisable to $\mathcal{D}$.

[Lyn96] presents a proof of this theorem for a slightly different canonical automaton, but the proof carries directly to the automata discussed here. The basic motivation is that an order for the operations in any execution can be constructed according to the order of do actions in that execution. That is, the do actions act as linearisation points for the operations. Since the transitions labelled by do actions are just applications of the update function of the datatype being implemented to a instance of that datatype, this order induces a valid sequential execution.

Completeness of the Canonical Automaton

There is an important question remaining about the canonical automata constructed in this section. Can we guarantee that A has *every* linearisable history in its set of traces? This is a very desirable property to have: if it holds and we have some automaton A meant to implement $\mathcal{D}$ we know that if we are unable to show $C \leq_T A$, then either we are not clever enough or our implementation contains a bug. We do not have to find some other way to specify linearisable histories. The following theorem formalises a sense in which the canonical automaton is *complete* (cf. [Lyn96], Theorem 13.5).

Theorem 2.4 (Completeness of the canonical automaton)
All histories that are linearisable with respect to $\mathcal{D}$ are traces of A.

Again, [Lyn96] provides a proof. Briefly, for every history h linearisable with respect to $\mathcal{D}$, there is a total order over the operations of h, witnessing its linearisability. An execution of C can be constructed, containing invocation and response actions in the order given by the history, with internal do actions in the order given by the linearisation points.

2.7 Concluding Remarks

This chapter presents the theoretical preliminaries of the verification techniques used in this thesis. We note here that every verification presented in this thesis has been proof checked using the PVS proof assistant. This provides a high degree of assurance that our proofs are correct. We do not present the PVS versions of our proofs in this thesis, or discuss the techniques used in our PVS development in any detail.

Chapter 3

Verifying a Nonblocking Queue Algorithm

This chapter describes a verification of a lock-free queue algorithm that is a variant of the practical and widely-used algorithm of Michael and Scott [MS96b, MS98a]. In fact, we verify a slightly optimised version of the algorithm. This optimisation was discovered during the early stages of the verification process. This optimisation is minor and does not constitute a significant change in the underlying algorithm, so henceforth we refer to the optimised version as the *M&S queue*.

This verification is the simplest presented in this thesis, and thus serves as an introduction to the techniques used in the other verifications. The verification of the M&S queue requires both a forward and a backward simulation, so this chapter explores the use of both techniques. Also, this algorithm uses dynamic memory, and so this verification provides an example of the way we model a dynamic heap and relate heap objects to the abstract datatype.

This chapter presents work first reported in [DGLM04]. At that time, the work presented here was the first complete formal verification of a version of the M&S queue. [YS03] presented an earlier verification using a model-checking technique, but that work does not describe a complete verification. [AC05] presents a formal verification of a queue algorithm based on the Michael and Scott queue, using a technique based on refinement and formal proof. We discuss both of these contributions in Section 3.6.

Interestingly, none of this related work directly addresses the issue of prophetic linearisation. The authors of [AC05] verify a modified M&S queue that does not exhibit prophetic linearisation. The authors of [YS03] do not directly consider the issue of whether the M&S

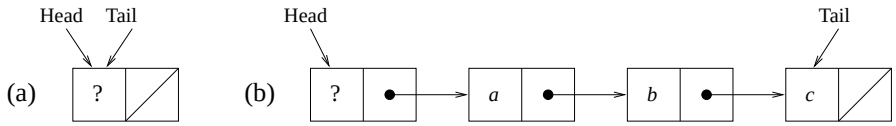


Figure 3.1: Basic queue representation

```
struct refint_t {
  node *ptr;
  int ver
}

struct node {
  value val;
  refint_t next
}
```

```
struct queue{
  refint_t Head, Tail;
}

initialise(queue * Q){
  dummy := new_node();
  dummy->next := null;
  Q->Head := (0, dummy);
  Q->Tail := (0, dummy);
}
```

Figure 3.2: Declarations and initialisation.

queue meets some behavioural specification of a concurrent queue, focussing rather on verifying that the M&S queue has certain invariants. The present verification exploits one of the main advantages of the I/O automaton framework: that it enables direct and formal treatment of prophetic linearisation, using backward simulation.

Section 3.1 presents the M&S queue. Section 3.2 presents the abstract and concrete automata. Section 3.3 presents the backward simulation. Section 3.4 presents the forward simulation. In Section 3.5 we describe the most important parts of the proof. Section 3.6 presents a comparison with related work, and we conclude the Chapter in Section 3.7.

3.1 The Queue Implementation

The M&S queue implements a queue as a linked list of nodes, each having a `val` and a `next` field, along with `Head` and `Tail` pointers. `Head` points to the first node in the list, which is a dummy node; the remaining nodes contain the values in the queue. When no operation is in progress, `Tail` points to the last node in the list. Figure 3.1 shows an empty queue and a queue containing values *a*, *b* and *c*. The declarations and initialisation are shown in Figure 3.2. Pseudocode for the `enqueue` and `dequeue` operations is given in Figures 3.3 and 3.4.

```
void enqueue(queue *Q, value v){
E1. nd := new_node();
E2. nd->val := v;
E3. nd->next.ptr := null;
E4. while (true){
E5.   tail := Q->Tail;
E6.   next := tail.ptr->next;
E7.   if (tail = Q->Tail){
E8.     if (next.ptr = null){
E9.       if (CAS(&tail.ptr->next,
                  next,
                  (nd, next.ver+1))){
E10.        break;
E11.      }
E12.    }else{
E13.      CAS(&Q->Tail,
              tail,
              (next.ptr, tail.ver+1));
E14.    }
E15.  }
E16.}
E17.CAS(&Q->Tail,
         tail,
         (nd,tail.ver+1))
}
```

Figure 3.3: Pseudocode for the enqueue operation.

```
bool dequeue(queue *Q, value *pv){
D1. while (true){
D2.   head := Q->Head;
D3.   next := head->next;
D4.   if (head = Q->Head){
D5.    if (next.ptr = null){
D6.      return false;
D7.    } else {
D8.      *pv := next.ptr->val;
D9.      if (CAS(&Q->Head,
                 head,
                 (next.ptr, head.ver+1))){
D10.       tail := Q->Tail;
D11.       if (head.ptr = tail.ptr){
D12.         CAS(&Q->Tail,
                 tail,
                 (next.ptr, tail.ver+1))
           }
           break;
D13.     }
D14.   }
D15. }
D16.}
D17.free_node(head.ptr);
}
```

Figure 3.4: Pseudocode for the dequeue operation.

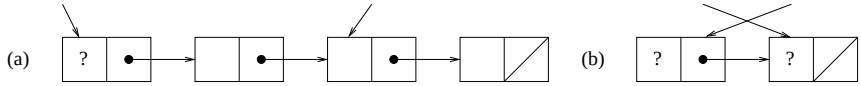


Figure 3.5: Queue representation variations

Shared locations containing pointers (i.e., `Head` and `Tail` variables and `next` fields) are usually updated using CAS operations. The one exception is in the initialisation of a new node (line E3), where a store is sufficient because no other process can access a node while it is being initialised. These shared locations contain a *version number* as well as a pointer. This version number is incremented atomically every time the location is written. As discussed in Section 1.1.3, this use of version numbers provides a very strong probabalistic guarantee that ABA cannot occur. Henceforth, we assume that version numbers are unbounded.

A process p executing an `enqueue` operation acquires and initialises a new node (E1–E3), and appends the new node to the list by repeatedly determining the last node in the list, i.e., the node whose `next.ptr` field is `null` (E5–E8, E13), and attempting to make its `next.ptr` field point to the new node (E9). Then p attempts to make `Tail` point to this node (E17).[1] Between p appending its new node and `Tail` being updated, `Tail` *lags* behind the last node in the list. Examples of this situation are presented in Figure 3.5. In Figure 3.5(a), the queue contains three elements. In Figure 3.5(b), the queue is empty, because `Head` points to a node with a `null next` pointer.

We cannot determine the last node in the list by just reading `Tail`, because another enqueuing process q may cause `Tail` to lag. Since p cannot wait for q to update `Tail` (that would compromise lock-freedom), p attempts to "help" q by doing the update (E13). Thus, `Tail` can lag behind the end of the list by at most one node.

Also, another process may change `Tail` after p reads it at E5, but before p dereferences (its local copy of) the pointer at E6. To ensure that the value read at E6 is valid, p checks at E7 that `Tail` has not changed since p executed E5. If the test at E8 shows that the node accessed at E6 had no successor at that time, then we know that the node was the last node in the list at that time. Similarly, a successful CAS at E9 guarantees that the `next` field of that node is unchanged in the interval between p's executions of E6 and E9.

We turn now to a description of the `dequeue` operation, presented in Figure 3.4. In this informal description of the M& S queue, we provide a C-style signature for the `dequeue`

[1]The CAS at E17 can be deleted without affecting the correctness of the algorithm. However, without this CAS, `Tail` would not point to the last node of the list in all quiescent states.

operation. Rather than `dequeue` returning some *null* value, `dequeue` returns a boolean value that is `false` if and only if *p* found the queue empty. A parameter `value * pv` is used as a pointer to a location that holds the value that was dequeued, if *p* found the queue nonempty. This is the convention used in earlier presentations of the algorithm [MS96b, MS98a].

A process *p* executing a `dequeue` operation checks whether the dummy node (pointed to by `Head`) has a successor (D2–D5). If not, then the queue was empty when *p* executed D3, so the operation returns `false` (D6). As in the `enqueue` operation, `Head` is read twice to ensure that the node accessed at D3 was the dummy node at that time.

If the dummy node has a successor, then *p* reads the value in the successor node (D8), expecting that this node is the first non-dummy node in the list. Process *p* must read the value now because concurrent operations may modify the value field of *p*'s `next` node after it is removed from the list. After reading the value, *p* attempts to swing `Head` to point to the node whose value *p* read at D8 (D9). If the attempt succeeds, that node is the new dummy node; its value is removed from the queue by the successful CAS. If the attempt fails, *p* retries the operation from the beginning.

Once *p* has successfully executed the CAS at D9, it remains to allow the old dummy node to be reused. For the reasons discussed in Section 1.1.3, this node cannot be freed to the system because another process may be about to access it. Instead, it is placed on a *freelist*, using the `free_node` operation (D17). The `new_node` operation (E1) returns a node from the freelist, if one is available; otherwise, it allocates and returns a new node. In a typical system, the freelist could be implemented using the Treiber stack, which is described in Chapter 1.

Before passing the old dummy node to `free_node`, a dequeuing process checks for the special case shown in Figure 3.5(b), where the `Head` and `Tail` are "crossed", because `Tail` is lagging and points to the old dummy node (D10-D11). In this case, it attempts to update `Tail` (D12) before putting the old dummy node on the freelist.

Our algorithm differs from Michael and Scott's original algorithm [MS96b, MS98a] in that we test whether `Tail` points to the dummy node only *after* `Head` has been updated, so a dequeuing process reads `Tail` only once. The `dequeue` in the original algorithm performs this test before checking whether the `next` pointer in the dummy node is `null`, so it reads `Tail` every time a dequeuing process loops. In the modified algorithm presented here, processes only perform this read and test once for each deque operation. Under high load, when operations retry frequently, this change will reduce the number of accesses to

shared memory.

3.2 Modelling the Queue Specification and Implementation

We now describe the specification and implementation automata for our verification of the M&S queue. Section 3.2 presents a formal definition of the queue datatype. The specification automaton, denoted by *AbsAut*, is the canonical automaton for the queue datatype, and is presented in Section 3.2.1. (The general construction of a canonical automaton is described in Section 2.6). The implementation automaton, denoted by *ConcAut*, models the M&S queue algorithm directly, and is presented in Section 3.2.2.

The Queue Datatype

A queue contains a sequence of objects from some set (called here *V*) and provides an *enqueue* operation, which adds a value to one end of the sequence, and a *dequeue* operation, which removes a value from the opposite end of the sequence.

We define the queue datatype using a set $\mathcal{Q}$, whose elements are the queues themselves, along with functions enq and deq, modelling respectively enqueue and dequeue operations. A queue $Q \in \mathcal{Q}$ is a triple $(Q.seq, Q.Head, Q.Tail)$, where $Q.seq$ is a sequence of values,[2] and $Q.Head$ and $Q.Tail$ are naturals satisfying the constraint that $Q.Tail \leq Q.Head$. $Q.Head$ and $Q.Tail$ delimit the range corresponding to queue elements: the queue consists of the integers $Q.seq(Q.Head+1)$ through to $Q.seq(Q.Tail)$, inclusive. A queue Q is empty, written $empty(Q)$, iff $Q.Head = Q.Tail$. Initially, $Q.Head = Q.Tail = 0$.

The function enq, modelling the enqueue operation, takes as arguments a queue value Q and a value $v \in V$ to be enqueued and returns a new queue containing the value:

$$enq(Q, v) = (Q.seq \oplus \{Q.Tail + 1 \mapsto v\}, Q.Head, Q.Tail + 1)$$

The function deq, modelling the dequeue operation, takes as arguments a queue Q and returns a pair consisting of a new queue (the old queue with the first element removed), and a *return value* in $V_\perp = V \cup \{null\}$ where $null$ is some value not in V. A $null$ return value indicates that the queue is empty and so no value from V is available.

$$deq(Q) = \begin{cases} ((Q.seq, Q.Head + 1, Q.Tail), & \\ \qquad Q.seq(Q.Head + 1)) & \text{if } \neg empty(Q) \\ (Q, null) & \text{otherwise} \end{cases}$$

[2] That is, a function from naturals to values as described in Section 1.3.

Note that the functions enq and deq preserve the constraint that $Q.Head \leq Q.Tail$, where Q is the new queue value returned by either function.

Finally, the queue datatype (D, D_0, I, R, u) is defined as follows:

$$
\begin{aligned}
&D = \mathcal{Q} \\
&D_0 = \{Q \mid empty(Q)\} \\
&I = \{enq\_inv(v) \mid v \in V\} \cup \{deq\_inv\} \\
&R = \{enq\_resp\} \cup \{deq\_resp(r) \mid r \in V_\perp\} \\
&u(Q, inv) = \begin{cases}
(enq(Q, v), enq\_resp) & \text{if } inv = enq\_inv(v) \\
 & \quad \text{for some } v \in V \\
(\pi_1(deq(Q)), & \\
\quad deq\_resp(\pi_2(deq(Q)))) & \text{otherwise}
\end{cases}
\end{aligned}
$$

3.2.1 The Abstract Automaton

AbsAut has a shared variable Q, which holds the abstract queue. The do steps of *AbsAut* apply the enq and deq functions defined in the previous section directly, rather than using the queue's update function u, which simplifies the notation. Each process p, has a variable pc_p which ranges over the program counter values of the canonical queue automaton. The set of initial states of *AbsAut* is defined as follows.

$$start_{AbsAut} = \{ab \mid empty(ab.Q) \wedge \forall\, p \bullet ab.pc_p = idle\}$$

The transition relation of *AbsAut* is presented in Figure 3.6.

3.2.2 The Concrete Automaton

The concrete automaton *ConcAut* models the queue implementation described in Section 3.1. The M&S queue uses a shared heap that contains the dynamically allocated nodes used in the queue data structure. We first describe our model of the heap. We then describe the actions and states of the concrete automaton.

The Heap Model

We model a heap in which every object is a node with two fields val and $next$. Each of these fields contains a pointer/version-number pair. This is a simplification designed to reduce unnecessary complexity in the model. In the M&S algorithm, `val` fields are not equipped

$enq\_inv_p(v)$:	do_enq_p :	enq_resp_p :
pre $pc_p = idle$	**pre** $pc_p = enq(v)$	**pre** $pc_p = enq\_resp$
eff $pc_p := enq(v)$	**eff** $pc_p := enq\_resp$	**eff** $pc_p := idle$
	$Q := enq(Q, v)$	
deq_inv_p :	do_deq_p :	$deq\_resp_p(r)$:
pre $pc_p = idle$	**pre** $pc_p = deq$	**pre** $pc_p = deq\_resp(r)$
eff $pc_p := deq$	**eff** $pc_p := deq\_resp(\pi_2(deq(Q)))$	**eff** $pc_p := idle$
	$Q := \pi_1(deq(Q))$	

Figure 3.6: The transition relation of *AbsAut*. The variable *p* ranges over processes, *v* ranges over values from *V*, and *r* ranges over $V_\perp$. Recall that $\pi_1(deq(Q))$ is the queue returned by the function *deq*, and $\pi_2(deq(Q))$ is the value.

with version numbers. However, the value of the version number of any val field is never mentioned in the definition of the transition relation of the concrete automaton. Therefore, the presence of version numbers in the val fields of our model makes no difference to its behaviour.

We write $POINTER$ for the set of pointers, $HEAP$ for the set of heaps, and $FIELD$ for the set of field names (either val or $next$). A heap $h \in HEAP$ is a pair $(h.eval, h.unalloc)$: the function $h.eval : POINTER \times FIELD \to POINTER \times \mathbb{N}$ takes a pointer to a node and a field, and returns the pointer value and version number associated with that field of that node in *h*; and $h.unalloc$ is the set of pointers that have not yet been allocated in *h* (so $h.unalloc$ models the system freelist). Given some value $x : POINTER \times \mathbb{N}$, let $x.ptr = \pi_1(x)$ and $x.ver = \pi_2(x)$.

An assignment $pt{\to}fd := (pt', i)$, which updates field *fd* in the node pointed to by *pt*, is modelled using a function $update : HEAP \times POINTER \times FIELD \times POINTER \times \mathbb{N} \to HEAP$ defined by:

$$update(h, pt, fd, pt', i) = (h.eval \oplus \{(pt, fd) \mapsto (pt', i)\}, h.unalloc)$$

Allocation of a new node is modelled with the function $new : HEAP \to HEAP \times$

$POINTER$ satisfying the following properties:

$$\begin{aligned}
&\mathit{new}(h) = (h', \mathit{null}) \Rightarrow h.unalloc = \varnothing \wedge h' = h \\
&\mathit{new}(h) = (h', p) \wedge p \neq null \Rightarrow \\
&\qquad p \in h.unalloc \wedge h'.eval = h.eval \wedge h'.unalloc = h.unalloc \setminus \{p\}
\end{aligned}$$

Together, these properties guarantee that *new* returns a $null$ pointer exactly when it is applied to a heap with an empty $unalloc$ set. When *new* is applied to a heap with nonempty $unalloc$ set, it returns a pointer from that set, and a heap with that pointer removed from the set of unallocated pointers.

Michael and Scott do not specify what happens if `enqueue` is unable to allocate a new node. A practical implementation might return from the *enqueue* operation with an error code, or raise an exception. However, this would require that the specification automaton *AbsAut* be able to represent an "out of memory" error using some response action. However, it is difficult to say when it is correct for a specification to return this kind of error, which originates in the execution context of an implementation. We choose to ignore this issue. In our model, if *new* returns $null$, then *ConcAut* loops until space becomes available.

Note that in the heap model presented here, a process can dereference a pointer (by applying $eval$ or by applying *update*) even when that pointer is `null` or in the set $unalloc$. In a real system, this behaviour could cause an error. However, no memory is ever freed in the M&S algorithm. Further, the only local variables that can be `null` are the $next_p$ variables, but every dereference of such a variable is preceded by a test that the variable is not `null`. So for simplicity, the heap model presented here ignores this issue. That is, we assume that the heap function $eval$ is defined for all pointers and fields. However, we do prove that the M&S queue has the property that only non-`null` and allocated variables are ever dereferenced: Section 3.4 describes how we prove this property. In more complex verifications, it would be preferable to have a heap model that represented the situations in which dereferencing a pointer would be illegal. Chapter 6, which describes the verification of a complex algorithm where deallocation does occur, presents an extended heap model in which accesses to unallocated or `null` pointers cause an error flag to be set. Once the error flag has been set, the effect of all heap operations is undefined. Using that model, the proof obligations of forward simulation require us prove that this error flag is never set during any execution of the relevant automaton.

The Concrete Automaton

Our concrete automaton must capture the assumed atomicity of the read, write and allocation operations. We wish to model a situation in which each process can execute read, write and allocation operations without interleavings with other processes. This is easily achieved using I/O automata: each such atomic operation is modelled using one internal action. For example, *ConcAut* has an internal action e_1_p modelling a process *p* executing line E1 of `enqueue`, allocating a new node from the freelist. An action e_2_p models a process *p* executing line E2, writing a value into the `val` field of its new node.

CAS operations conditionally modify the heap. We split these CAS operations into two internal actions, one action modelling a successful CAS that modifies the heap; the other modelling an unsuccessful CAS that leaves the heap unchanged. For example, *ConcAut* has internal actions $d\_9\_yes_p$ and $d\_9\_no_p$ modelling *p* executing D9 when the CAS is successful and unsuccessful, respectively. The precondition of each action is used to select which of these actions a process should take: the precondition of the action modelling a successful CAS implies that the value in the location being CASed is equal to the given expected value; and the precondition of the action modelling an unsuccessful CAS implies that the value in the location is not the expected value.

Other conditionals (`if` statements), are modelled similarly to the CAS operations. They are split into two internal actions, one modelling the case where the condition succeeds, the other modelling the case where the condition fails. For example, *ConcAut* has internal actions $d\_4\_yes_p$ and $d\_4\_no_p$ modelling *p* executing D4, when the condition is respectively true or false.

Thus for each line of code in Figures 3.3 and 3.4, *ConcAut* has either one or two internal actions for each process. Also, because it is meant to implement a queue, *ConcAut* has the same external actions as *AbsAut*.

Each process *p* has a local "program counter" variable pc_p, ranging over a type that contains one value for each line of code containing a read, write, conditional or CAS, (for example, there is a counter value e_1 corresponding to line E1), and special values $idle$, enq_resp and deq_resp that play the same roles as in *AbsAut*. That is, when $pc_p = idle$, process *p* is not executing any operation on the queue; when $pc_p = enq\_resp$, *p* is about to return from an enqueue operation; and when $pc_p = deq\_resp(r)$ for some $r \in V_\perp$, *p* is about to return from a dequeue operation with value *r*.

ConcAut has variables $h \in HEAP$, $Head, Tail \in POINTER \times \mathbb{N}$, and $freelist \subseteq$

$$\{cs \mid cs.Head = cs.Tail \wedge cs.Head \neq null \wedge cs.Head.ver = 0 \wedge$$
$$cs.Head \xrightarrow{cs} next.ptr = null \wedge \neg cs.free?(cs.Head.ptr) \wedge$$
$$(\forall p \bullet cs.pc_p = idle) \wedge$$
$$\neg cs.free?(null) \wedge cs.Head \xrightarrow{cs} val = v_0 \wedge$$
$$cs.freelist \cap cs.h.unalloc = \varnothing\}$$

Figure 3.7: The initial states of *ConcAut*.

$POINTER$, which model the heap, `Head`, `Tail` and the freelist.[3] For each process p, there are variables $head_p, tail_p, next_p \in POINTER \times \mathbb{N}$, and $node_p \in POINTER$, which model the local variables in the code, and a local variable $result_p \in POINTER$ to hold the value that p returns from the dequeue operation.[4]

The initial states for *ConcAut* are presented in Figure 3.7 and the transition relation is presented in Figures 3.8, 3.9 and 3.10.

Figure 3.7 uses the notation $pt \xrightarrow{cs} fd$ to mean $cs.h.eval(pt, fd)$ and $cs.free?(pt)$ to mean $pt \in cs.unalloc \cup cs.freelist$. This notation is used in the remainder of this chapter and a similar notation is used in Chapter 6.

3.2.3 The Intermediate Automaton

As discussed in Chapter 2, simulation proofs can often be done using a *forward simulation*, in which the abstract execution is constructed by starting at the beginning of the concrete execution and working forwards.

However, forward simulation is not sufficient to prove that *ConcAut* implements *AbsAut*. The only point during a `dequeue` operation at which the queue is guaranteed to be empty is when the operation executes D3, loading `null` into `next`. A forward simulation would need to determine at this point whether the operation will return `null`. This is not possible, however, since the operation will retry if `Head` is changed between the operation's execution of D2 and D4.

We use two examples to explain this. First we describe a situation in which a dequeuing process returns empty, but where execution of neither D4 nor D5 can be used as a linearisation

[3] Recall that the M&S queue uses a freelist to recycle nodes without releasing the memory back to the system freelist. $freelist$ models the M&S queue freelist; $h.unalloc$ models the system freelist.

[4] In the pseudo-code of Figure 3.4, this value is returned in a location referenced by an input parameter.

$enq\_inv_p(v)$:
pre $pc_p = idle$
eff $pc_p := \text{e\_1}(v)$

e_2_p :
pre $pc_p = \text{e\_2}(v)$
eff $node_p{\rightarrow}val := v$,
$pc_p := \text{e\_3}$

e_3_p :
pre $pc_p = \text{e\_2}(v)$
eff $node_p{\rightarrow}next.ptr$
$:= null$,
$pc_p := \text{e\_5}$

enq_resp_p :
pre $pc_p = enq\_resp$
eff $pc_p := idle$

e_5_p :
pre $pc_p = \text{e\_5}$
eff $tail_p := Tail$,
$pc_p := \text{e\_6}$

e_1_p :
pre $pc_p = \text{e\_1}(v)$
eff $h := \pi_1(new\_nd())$,
$freelist :=$
$\pi_2(new\_nd())$,
$node_p :=$
$\pi_3(new\_nd())$,
$pc_p :=$
$\pi_3(new\_nd()) = null\ ?$
$\text{e\_1}(v)\ :\ \text{e\_2}(v)$

e_6_p :
pre $pc_p = \text{e\_6}$
eff $next_p := tail_p{\rightarrow}next$,
$pc_p := \text{e\_7}$

$e\_7\_yes_p$:
pre $pc_p = \text{e\_7} \wedge$
$tail_p = Tail$
eff $pc_p := \text{e\_8}$

$e\_7\_no_p$:
pre $pc_p = \text{e\_7} \wedge$
$tail_p \neq Tail$
eff $pc_p := \text{e\_5}$

$e\_8\_yes_p$:
pre $pc_p = \text{e\_8} \wedge$
$next_p.ptr = null$
eff $pc_p := \text{e\_9}$

$e\_8\_no_p$:
pre $pc_p = \text{e\_8} \wedge$
$next_p.ptr \neq null$
eff $pc_p := \text{e\_13}$

$e\_9\_yes_p$:
pre $pc_p = \text{e\_9} \wedge$
$tail_p.ptr{\rightarrow}next$
$= next_p$
eff $tail_p{\rightarrow}next :=$
$(node_p,$
$next_p.ver + 1)$,
$pc_p := \text{e\_17}$

Figure 3.8: Enqueue transitions of *ConcAut* (continued in next figure).

$e\_9\_no_p$:
pre $pc_p = \text{e\_9} \wedge$
$tail_p.ptr \rightarrow next \neq next_p$
eff $pc_p := \text{e\_5}$

$e\_13\_yes_p$:
pre $pc_p = \text{e\_13} \wedge$
$tail_p = Tail$
eff $Tail :=$
$(node_p, tail_p.ver + 1)$,
$pc_p := \text{e\_5}$

$e\_13\_no_p$:
pre $pc_p = \text{e\_13} \wedge$
$tail_p \neq Tail$
eff $pc_p := \text{e\_5}$

$e\_17\_yes_p$:
pre $pc_p = \text{e\_17} \wedge$
$tail_p = Tail$
eff $Tail :=$
$(node_p, tail_p.ver + 1)$,
$pc_p := enq\_resp$

$e\_17\_no_p$:
pre $pc_p = \text{e\_17} \wedge$
$tail_p \neq Tail$
eff $pc_p := enq\_resp$

Figure 3.9: Enqueue transitions of *ConcAut*.

point for the operation.

- Process *p* begins execution of a `dequeue` operation when the queue is empty. *p* executes lines D1-D3, loading `null` into its `next` variable at D3.
- Another process *q* executes a complete `enqueue` operation. The queue is no longer empty.
- Process *p* executes lines D4-D5. Because *p*'s `next` variable is `null` and the `Head` has not changed since *p*'s operation began, the tests at both D4 and D5 succeed. Hence *p* returns *false* at D5.

Because process *q* completed an enqueue operation the queue is no longer empty when *p* executes D4 and D5. Therefore, neither D4 nor D5 can be used as a linearisation point for *p*'s operation. The only point at which the queue is empty is when *p* executed D3, loading `null` into its `next` field.

Unfortunately, we cannot *a*lways choose D3 as a linearisation point when a process loads `null` into its `next` field. To see why, consider the following execution.

deq_inv_p :
pre $pc_p = idle$
eff $pc_p := deq$

$deq\_resp_p(r)$:
pre $pc_p = deq\_resp(r)$
eff $pc_p := idle$

d_2_p :
pre $pc_p =$ d_2
eff $head_p := Head$,
$pc_p :=$ d_3

d_3_p :
pre $pc_p =$ d_3
eff $next_p :=$
$head_p.ptr \rightarrow next$,
$pc_p :=$ d_4

$d\_4\_yes_p$:
pre $pc_p =$ d_4 $\wedge$
$head_p = Head$
eff $pc_p :=$ d_5

$d\_4\_no_p$:
pre $pc_p =$ d_4 $\wedge$
$head_p \neq Head$
eff $pc_p :=$ d_2

$d\_5\_yes_p$:
pre $pc_p =$ d_5 $\wedge$
$next_p.ptr = null$
eff $result_p := null$,
$pc_p := deq\_resp$

$d\_5\_no_p$:
pre $pc_p =$ d_5 $\wedge$
$next_p.ptr \neq null$
eff $pc_p :=$ d_8

d_8_p :
pre $pc_p =$ d_8
eff $result_p :=$
$next_p.\rightarrow val$,
$pc_p :=$ d_9

$d\_9\_yes_p$:
pre $pc_p =$ d_9 $\wedge$
$head_p = Head$
eff $Head :=$
$(next_p.ptr,$
$head_p.ver + 1)$,
$pc_p :=$ d_10

$d\_9\_no_p$:
pre $pc_p =$ d_9 $\wedge$
$head_p \neq Head$
eff $pc_p :=$ d_2

d_10_p :
pre $pc_p =$ d_10
eff $tail_p := Tail$,
$pc_p :=$ d_11

$d\_11\_yes_p$:
pre $pc_p =$ d_11 $\wedge$
$head_p.ptr =$
$tail_p.ptr$
eff $pc_p :=$ d_12

$d\_11\_no_p$:
pre $pc_p =$ d_11 $\wedge$
$head_p.ptr \neq$
$tail_p.ptr$
eff $pc_p :=$ d_17

$d\_12\_yes_p$:
pre $pc_p =$ d_12 $\wedge$
$tail_p = Tail$
eff $Tail :=$
$(next_p.ptr,$
$tail_p.ver + 1)$,
$pc_p :=$ d_17

$d\_12\_no_p$:
pre $pc_p =$ d_12 $\wedge$
$tail_p \neq Tail$
eff $pc_p :=$ d_17

d_17_p :
pre $pc_p =$ d_17
eff $freelist :=$
$freelist \cup \{head_p\}$,
$pc_p := deq\_resp$

Figure 3.10: Dequeue transitions of *ConcAut*.

- As before, process *p* executes lines D1-D3 when the queue is empty.
- As before, another process *q* executes a complete `enqueue` operation.
- Now, another process *r* executes a complete `dequeue` operation, followed by a complete `enqueue` operation. The queue now contains one element. Further, the execution of *r*'s `dequeue`has modified `Head`.
- Process *p* executes the test at line D4 which fails because of *r*'s `dequeue`. So *p* loops back to the top of the `while` loop. *p* completes its `dequeue` operation by removing the last value that *r* enqueued.

Therefore, we need to use a *backward simulation*, showing how to construct an abstract execution by working from the last step of a (finite) concrete execution back to the beginning. Because we are working backwards, we can distinguish between the two kinds of executions exemplified above, and correctly choose linearisation points for dequeue operations that return empty.

Since only this one aspect requires backward simulation, we define an intermediate automaton *IntAut*, which captures the behaviour of the implementation that defies forward simulation, namely the handling of $dequeue$ on an empty queue, and is otherwise identical to *AbsAut*. We then prove a backward simulation from *IntAut* to *AbsAut* (see Section 3.3), and a forward simulation from *ConcAut* to *IntAut* (see Section 3.4).

The intermediate automaton *IntAut* is identical to the abstract automaton, except that in *IntAut*, a process executing a $dequeue$ operation may "observe" whether or not the queue is empty at any time before it decides what value to return. In addition to the queue and counter variables that are in *AbsAut*, each state of *IntAut* has a variable $empty\_ok_p$, to record whether *p* has observed an empty queue during the current $dequeue$ operation. The initial states and transition relation of *IntAut* are presented in Figures 3.11 and 3.12 respectively.

$$\{ab \mid empty(ab.Q) \wedge \forall p \bullet ab.pc_p = idle\}$$

Figure 3.11: The initial states of *IntAut*. Note that these states are defined in precisely the same way as the initial states of *AbsAut*.

$enq\_inv_p(v)$:
pre $pc_p = idle$
eff $pc_p := enq(v)$

enq_do_p :
pre $pc_p = enq(v)$
eff $pc_p := enq\_resp$
$Q := enq(Q, v)$

enq_resp_p :
pre $pc_p = enq\_resp$
eff $pc_p := idle$

deq_inv_p :
pre $pc_p = idle$
eff $pc_p := deq$
$empty\_ok_p :=$ *false*

obs_empty_p :
pre $pc_p = deq$
eff $empty\_ok_p :=$ $empty(deq)$

$deq\_nonempty_p$:
pre $pc_p = deq \wedge \neg empty(deq)$
eff $pc_p :=$ $deq\_resp(\pi_2(deq(Q)))$
$Q := \pi_1(deq(Q))$

deq_empty_p :
pre $pc_p = deq \wedge empty\_ok$
eff $pc_p :=$ $deq\_resp(null)$

$deq\_resp_p(r)$:
pre $pc_p = deq\_resp(r)$
eff $pc_p :=$ $idle$

Figure 3.12: The transition relation of *IntAut*.

IntAut has the same external actions as *AbsAut*, and the same internal action do_enq_p; the only difference for these transitions is that deq_inv_p sets $empty\_ok_p$ to *false*. *IntAut* has a new internal action $observe\_empty_p$ that sets $empty\_ok_p$ to record whether or not the queue Q is empty, which p may perform whenever its program counter value is deq. Also, in place of the do_deq_p action in *AbsAut*, *IntAut* has two actions, deq_empty_p and $deq\_nonempty_p$, allowing these cases to be treated separately. The $deq\_nonempty_p$ action is the same as the abstract automaton's do_deq_p action except that its precondition additionally requires that the queue is nonempty. The deq_empty_p action simply changes p's program counter from deq to $deq\_resp(null)$. The precondition for this action requires that $empty\_ok_p$ is true, indicating that p has observed that the queue was empty at some point during its execution. The point when this observation action takes place is the linearisation point for the operation.

Splitting $dequeue$ operations that return $null$ into one or more observations that the

queue is empty, followed by a decision to return $null$ based on the knowledge that we have observed the queue to be empty at some point during the operation, makes it possible to prove a forward simulation from the concrete automaton to the intermediate one, as we show in Section 3.4. In the forward simulation, we match steps of the concrete automaton where a process reads $null$ from $Head$ with the $observe\_empty$ action of the same process.

3.3 The Backward Simulation

In this section we define a relation BSR (see Figure 3.13), and show that it is a backward simulation from *IntAut* to *AbsAut*. Given states *as* of *AbsAut* and *is* of *IntAut*, the third conjunct of BSR requires that the queues represented by the two states are the same. The first two conjuncts require that each process is roughly speaking "at the same stage" of the same operation in both states, or is not executing any operation in either state. For example, if *p* is idle in *is* (i.e., $is.pc_p = idle$) then *p* is also idle in *as*. The first conjunct (*basic_ok*) covers the simple cases; the second conjunct (*dequeuer_ok*) covers the only interesting case, in which a process can be at slightly different stages in the two automata because $dequeue$ operations can take place over two or more steps. Specifically, if in *is*, *p* has invoked $dequeue$ but has not yet executed either deq_empty_p or $deq\_nonempty_p$ (i.e., $is.pc_p = deq$), then in *as*, either pc_p is also deq, or $pc_p = deq\_resp(null)$, indicating that *p* has already executed deq_empty_p. In the latter case, $is.empty\_ok_p$ must also be true, showing that *p* has observed that the queue was empty at some point during its $dequeue$ operation. In a situation where $as.pc_p = deq\_resp(null)$ but $is.pc_p = deq$, the dequeue operation of process *p* has been linearised earlier in the execution.

We turn now to the proof that BSR is a backward simulation from *IntAut* to *AbsAut*. For convenience, we state the proof obligations for *one-step* backward simulation, as applied to the automata *IntAut* and *AbsAut*.

1. For all $is \in start(IntAut)$ and all *as* such that $BSR(is, as)$, $as \in start(AbsAut)$.

2. For all $is \in reach IntAut$, $is' \in states(IntAut)$, $a \in external(IntAut)$, if $is \xrightarrow{a} is'$, then for all *as*′ such that $BSR(is', as')$, there is some *as* such that $BSR(is, as)$ and $as \xrightarrow{a} as'$.

3. For all $is \in reach(IntAut)$, $is' \in states(IntAut)$, $a \in internal(IntAut)$, then for all *as*′ such that $BSR(is', as')$, one of the following is satisfied:

$$
\begin{aligned}
&BSR(as, is) \hat{=} \\
&\quad basic\_ok(as, is) \wedge \\
&\quad dequeuer\_ok(as, is) \wedge \\
&\quad is.Q = as.Q \\
&basic\_ok(is, as) \hat{=} \\
&\quad \forall\, p \bullet is.pc_p \neq deq \Rightarrow is.pc_p = as.pc_p \\
&dequeuer\_ok(as, is) \hat{=} \\
&\quad \forall\, p \bullet is.pc_p = deq \Rightarrow \\
&\qquad\qquad (as.pc_p = deq \vee \\
&\qquad\qquad (as.pc_p = deq\_resp(null) \wedge is.empty\_ok_p))
\end{aligned}
$$

Figure 3.13: The backward simulation relation BSR

(a) there exists some *as* and action $b \in internal(AbsAut)$ such that $BSR(is, as)$ and $as \xrightarrow{b} as'$.

(b) $BSR(is, as')$.

4. For all $is \in reach(IntAut)$, there exists some *as* such that $BSR(is, as)$.

Conditions 1 and 4 are trivial, because related states of *IntAut* and *AbsAut* are almost identical, so we treat them very briefly. The first condition can be seen by observing that, for any $is \in start_{IntAut}$, $is.Q$ is empty and all the $p \in PROC$ are $idle$. Thus for any related *as*, $as.Q$ is empty and all $p \in PROC$ are idle. For the second condition, observe that given *is* (reachable or not), we can construct an *asa* such that $is.Q = as.Q$ and for all processes *p*, $is.pc_p = as.pc_p$. It is easy to see that then we have $BSR(as, is)$.

Conditions 2 and 3 are more complicated. We define a *step-correspondence* function [RR00], that determines the abstract action to choose, given an intermediate action and abstract poststate. We use a step-correspondence function in the verifications described in Section 3.4 as well as Chapters 4 and 6, so the simple function used here serves as an introduction to the technique.

The step-correspondence function *s* takes as arguments an intermediate action *a* and an abstract poststate as'. The function *s* returns either an abstract action or a special value $\perp$ that is not an abstract action. When $s(a, as')$ is an action, and *a* is external, $s(a, as')$ returns *a* as required by Condition 2 above. When $s(a, as')$ is an action but *a* is internal, $s(a, as')$

provides a witness for *b* in Condition 3a. Finally *s* is defined so that if $s(a, as') = \perp$ then $BSR(is, as')$. Thus, if $s(a, as') = \perp$, then we can fulfill Condition 3b. In this case, we say that $is \xrightarrow{a} is'$ is a *stutter* step. Formally, *s* is defined as follows.

$$s(a, as') = \begin{cases} a & \text{if } a \in external_{IntAut} \\ do\_enq_p(v) & \text{if } a = do\_enq_p(v) \text{ for some } p \text{ and } v \\ do\_deq_p & \text{if } a = observe\_empty_p \text{ for some } p \text{ and} \\ & as'.pc_p = deq\_resp(null) \\ do\_deq_p & \text{if } a = deq\_nonempty_p \text{ for some } p \\ \perp & \text{otherwise} \end{cases}$$

For every intermediate action *a* except $observe\_empty$, deq_empty and $deq\_nonempty$, we choose the same action *a* for *AbsAut*. In the case of external actions, this choice is required by Condition 2. For $deq\_nonempty$, we choose do_deq; and for deq_empty, we choose to stutter. Recall that a $dequeue$ operation on an empty queue is linearised to a point at which it executes $observe\_empty$, and not when it executes deq_empty. We reflect this choice of linearisation point by choosing do_deq for exactly one execution of $observe\_empty$ within that operation. We guarantee that we only choose do_deq once by examining that abstract *poststate* to check whether the process has yet executed its do_deq operation. It has done so exactly when its abstract program counter is $deq\_resp(null)$.

We also define a *prestate function*, *t* that generates the abstract prestate. Just as the step-correspondence function *s*, *t* takes as arguments an intermediate action *a* and an abstract state *as*′. When $s(a, as') \neq \perp$, $t(a, as')$ returns the abstract pre-state *as* required by Conditions 2 or 3a. When $s(a, as') = \perp$, $t(a, as') = as'$. Given *s* and *t*, we can combine Conditions 2, 3a and 3b into one proof obligation.

> For all $is \in reachIntAut$, $is' \in states_{IntAut}$, $a \in acts_{IntAut}$, if $is \xrightarrow{a} is'$, then for all *as*′ such that $BSR(is', as')$, $BSR(is, t(a, as'))$ and if $s(a, as') \neq \perp$, $t(a, as') \xrightarrow{s(a,as')} as'$.

It is generally easy to construct $t(a, as')$. In many cases, we simply replace the program counter of the process *p* whose action is being executed in the intermediate transition with the value required by the precondition of the abstract action. The only nontrivial case arises for the do_enq action, because to construct the program counter before the action, we must determine what value the enqueue operation is enqueuing. This is achieved by taking the value from the queue position that is updated by the do_enq action.

We now outline the proof that for all intermediate states *is*, *is′*, and abstract states *as′* such that $BSR(is', as')$ and $is \xrightarrow{a} is'$ where $a = observe\_empty_p$ for some *p*, $BSR(is, t(a, as'))$, and further that $t(a, as') \xrightarrow{s(a,as')} as'$, discharging the proof obligation for $observe\_empty$ actions. We omit the cases where *a* is some other action because they are straightforward and tedious.

First, consider the case where $as'.pc_p = deq\_resp(null)$. By *dequeuer_ok* of BSR, we have $is'.empty\_ok_p$. Because $observe\_empty_p$ sets $empty\_ok_p$ to *true* if and only if the queue is empty in state *is*, and does not modify the queue, it follows that the queue is empty in state *is′*, and therefore by BSR, the queue is empty in state *as′*. Therefore, the abstract prestate $t(a, as')$ is just like *as′*, except that $t(a, as').pc_p = deq$. Hence $is.Q = is'.Q = as'.Q = t(a, as').Q$. Furthermore, for each process $q \neq p$, $is.pc_q = is'.pc_q$ and $t(a, as').pc_q = as'.pc_q$, so the predicates $basic\_ok$ and $dequeuer_ok$ are preserved when applied to the process *q*. Finally, $is.pc_p = t(a, as').pc_p = deq$ so $dequeuer\_ok$ is true when applied to *p*, as is $basic\_ok$.

It is usually straightforward to prove $t(a, as') \xrightarrow{s(a,as')} as'$, since the construction of $t(a, as')$ ensures that the precondition for $s(a, as')$ holds and applying the effect of $s(a, as')$ to $t(a, as')$ yields *as′*. It is slightly trickier in our case, where the intermediate transition is an $observe\_empty$ action. Not every execution of $observe\_empty$ corresponds to a linearisation point for a $dequeue$ operation that returns $null$ (*IntAut* can execute $observe\_empty$ multiple times within a single $dequeue$ operation, while in *AbsAut* there is exactly one do_deq action per $dequeue$ operation). Therefore, for each $dequeue$ operation that returns $null$, we must choose do_deq for exactly one occurrence of $observe\_empty$, and choose the empty action sequence for the others.

We can only linearise a $dequeue$ operation by process *p* to an execution of the $observe\_empty_p$ action if the $dequeue$ operation returns $null$. This is true if $as'.pc_p$ is $deq\_resp(null)$, in which case we can infer that $empty\_ok_p$ in *is′* is *true*, from the *dequeuer_ok* conjunct of BSR. Because $observe\_empty_p$ sets $empty\_ok_p$ to *true* if and only if the queue is empty in state *is*, and does not modify the queue, it follows that the queue is empty in state *is′*, and therefore by BSR, the queue is empty in state *as′*. Therefore, we can construct the state *as* with an empty queue, which is needed to show that $as \xrightarrow{do\_deq_p} as'$ is a transition of the abstract automaton. Thus, we show that we can choose do_deq_p when *a* is $observe\_empty_p$ and $as'.pc_p$ is $deq\_resp(null)$. In all other cases, we choose the empty sequence for the abstract automaton when *a* is $observe\_empty_p$.

$$
\begin{aligned}
&\mathit{rel}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
&\qquad \mathit{enqueue\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \mathit{dequeue\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \\
&\qquad \mathit{obj\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \mathit{nds\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \\
&\qquad \mathit{distinctness\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \mathit{procs\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \\
&\qquad \mathit{injective\_ok}(\mathit{is}, \mathit{cs}, f) \wedge \mathit{access\_safety\_ok}(\mathit{is}, \mathit{cs}, f)
\end{aligned}
$$

Figure 3.14: The *rel* predicate

3.4 The Forward Simulation

In this section we describe a relation *FSR*, which is a forward simulation from *ConcAut* to *IntAut*. Because the concrete and intermediate automata are very different, the simulation relation and the proof are both substantially more complicated than the relation and proof described in the previous section.

The forward simulation relation over concrete state *cs* and intermediate state *is* is

$$\mathit{FSR}(\mathit{cs}, \mathit{is}) \mathrel{\hat{=}} \exists f : \mathit{rel}(\mathit{is}, \mathit{cs}, f)$$

where f is a function from naturals to pointers which we refer to as the *representation function*. We explain the purpose of f below, but briefly, it is used to constrain the structure of the nodes inside the heap of *ConcAut*, and relate that structure to the queue of *IntAut*. Figure 3.14 defines *rel*. The subpredicates of *rel* are defined later in this section.

The most important part of *rel* is the predicate *obj_ok* (Figure 3.15), which expresses the relationship between the concrete data structure, represented by nodes and pointers in *ConcAut*, and the queue variable of *IntAut*. To express this relationship, *obj_ok* uses the representation function f as follows. Recall that each state is of *IntAut* contains a queue variable Q, represented by a sequence and $Head$ and $Tail$ variables indicating which indexes are relevant in the current queue state. If $\mathit{obj\_ok}(\mathit{is}, \mathit{cs}, f)$ holds, then f indicates which node corresponds to each relevant position in $\mathit{is}.Q.seq$. That is, for each $i \in [\mathit{is}.Q.Head + 1 \ldots \mathit{is}.Q.Tail]$, $f(i)$ is the queue node in *cs* containing the value $\mathit{is}.Q.seq(i)$, and $f(\mathit{is}.Q.Head)$ indicates which queue node in *cs* is the dummy node pointed to by $\mathit{cs}.Head.ptr$. Moreover, for each $i \in [\mathit{is}.Q.Head + 1 \ldots \mathit{is}.Q.Tail - 1]$, $f(i+1)$ is the node pointed to by the next field of $f(i)$ (i.e., $f(i) \xrightarrow{cs} next).ptr = f(i+1)$), so that the order of values contained in the nodes of *cs* matches the order of values in the sequence $Q.\,\mathrm{seq}$ of the intermediate automaton.

$$
\begin{array}{ll}
\mathit{obj\_ok}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} & \\
\quad f(\mathit{is}.Q.Head) = \mathit{cs}.Head.ptr \wedge & (1) \\
\quad f(\mathit{is}.Q.Tail) \xrightarrow{\mathit{cs}} next.ptr = null \wedge & (2) \\
\quad (f(\mathit{is}.Q.Tail) = \mathit{cs}.Tail.ptr \vee & (3a) \\
\qquad (f(\mathit{is}.Q.Tail) = \mathit{cs}.Tail.ptr \xrightarrow{\mathit{cs}} next.ptr \wedge & \\
\qquad \neg \mathit{cs}.\mathit{free}(\mathit{cs}.Tail.ptr) \wedge & \\
\qquad \mathit{cs}.Tail.ptr \neq null)) & (3b) \\
\quad \wedge & \\
\quad \forall\, i : \mathbb{N} \bullet \mathit{is}.Q.Head \leq i \leq \mathit{is}.Q.Tail \Rightarrow & \\
\qquad (i \neq \mathit{is}.Q.Tail \Rightarrow (f(i) \xrightarrow{\mathit{cs}} next).ptr = f(i+1)) \wedge & (4a) \\
\qquad \mathit{is}.Q.seq(i) = (f(i) \xrightarrow{\mathit{cs}} \mathit{val}).ptr \wedge & (4b) \\
\qquad \neg \mathit{cs}.\mathit{free}(f(i)) \wedge & (4c) \\
\qquad f(i) \neq null & (4d)
\end{array}
$$

Figure 3.15: The *obj_ok* predicate

Conjunct 1 of *obj_ok* asserts that $f(\mathit{is}.Q.Head)$ is the dummy node. Conjunct 2 states that the last node in the queue has a $null\ next$ pointer. Conjunct 3 captures the fact that $Tail$ can "lag" behind the real tail of the queue: either $Tail$ is accurate (3a), or $\mathit{cs}.Tail.ptr$ points to the next-to-last node in the queue, and that in such a situation $\mathit{cs}.Tail.\mathit{ptr}$ is both allocated and non-*null*.[5] (3b). Conjunct 4 expresses properties of the nodes in the concrete queue: the pointer value of the $next$ field of each queue node points to the node corresponding to the next index (4a); the value in each queue node is the value in the corresponding position in $\mathit{is}.Q.seq$ (4b); none of the queue nodes is unallocated or in the freelist (4c); and none of the queue nodes is $null$ (4d). (Conjuncts 4c and 4d, together with Conjunct 3a allow us to prove that $\mathit{cs}.Tail.\mathit{ptr}$ is valid when the tail is not lagging.)

In order to show that *obj_ok* is preserved across transitions, we need to specify a new representation function for the poststate of each transition. Our choice for the new function is motivated by our choice of step-correspondence. We discuss the step-correspondence more completely in Section 3.5, but we note here that transitions of the form $e\_9\_yes_p$ (during which a new node is added onto the queue) are each matched with a transition labelled by

[5] It is easy to infer this information from the other clauses of *obj_ok* in cases where $Tail$ is not lagging.

$$
\begin{array}{l}
\mathit{enqueue\_ok}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad (\mathit{cs}.\mathit{pc}_p = \mathit{idle} \Rightarrow \mathit{is}.\mathit{pc}_p = \mathit{idle}) \wedge \\
\qquad\quad (\mathit{pc\_e\_1\_9}(\mathit{cs}, p) \vee \mathit{cs}.\mathit{pc}_p = \mathrm{e\_13} \Rightarrow \\
\qquad\qquad\quad \mathit{is}.\mathit{pc}_p = \mathit{enqueuing}(\mathit{cs}.val_p)) \wedge \\
\qquad\quad (\mathit{cs}.\mathit{pc}_p = \mathrm{e\_17} \vee \mathit{cs}.\mathit{pc}_p = enq\_resp \Rightarrow \mathit{is}.\mathit{pc}_p = enq\_resp) \\
dequeue\_ok(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad (\mathit{cs}.pc_p = \mathrm{d\_5} \wedge \mathit{cs}.next_p.ptr = null \Rightarrow \mathit{is}.empty\_ok_p) \wedge \\
\qquad\quad (\mathit{pc\_d\_2\_9}(\mathit{cs}, p) \Rightarrow \mathit{is}.pc_p = dequeuing) \\
\qquad\quad (\mathit{pc\_d\_10\_17}(\mathit{cs}, p) \vee \mathit{cs}.pc_p = deq\_resp \Rightarrow \\
\qquad\qquad\quad \mathit{is}.pc_p = deq\_resp(\mathit{cs}.result_p))
\end{array}
$$

Figure 3.16: The *enqueue_ok* and *dequeue_ok* predicates. A predicate of the form $\mathit{pc\_e\_m\_n}(\mathit{cs}, p)$, where m, n are integers, holds when $\mathit{cs}.\mathit{pc}_p = \mathrm{e\_}i$ for some $i \in [m \ldots n]$.

do_enq_p in the intermediate automaton (during which a new value is added onto the sequence of the intermediate automaton). The motivation for this is simple: enqueue operations "appear to take effect" during $\mathit{e\_9\_yes}_p$ and do_enq_p transitions of the respective automata, so these transitions are both linearisation points.

For a representation function f, concrete action a, concrete state cs and intermediate state is, we use the new representation function f', where

$$
f' = \begin{cases} f \oplus \{\mathit{is}.Q.Tail + 1 \mapsto \mathit{cs}.\mathit{node}_p\} & \text{if } a = e\_9\_yes \\ f & \text{otherwise} \end{cases}
$$

That is, for transitions labelled by actions of the form $\mathit{e\_9\_yes}_p$ (during which a new node is added onto the queue), we modify the representation function so that $f(\mathit{is}.Q.Tail + 1)$ is the new node added onto the queue. This is because after the transition from is labelled by do_enq_p with poststate is', $is'.QTail = \mathit{is}.Q.Tail + 1$, so that the new index is matched to the new node. In every other case we use the old representation function.

Predicates *enqueue_ok* and *dequeue_ok* (Figure 3.16) play the same role as *basic_ok* and *dequeuer_ok* in the backward simulation. That is, they assert that each process is "at the same stage" of the same operation in both states, or is not executing any operation in either state.

The other subpredicates of *rel* capture properties needed to support the proofs that these predicates are preserved across various transitions. We describe each in turn, giving an explanation of each predicate's meaning; a brief description of how we show it is preserved

$$
\begin{aligned}
&distinctness\_ok(is, cs, f) \triangleq \\
&\quad distinctness1\_ok(is, cs, f) \wedge \\
&\quad distinctness2\_ok(is, cs, f)
\end{aligned}
$$

Figure 3.17: The *distinctness_ok* predicate.

across transitions; and an outline of how it is used in the proof.

The *distinctness_ok* (Figures 3.17 and 3.18) predicate says that nodes are not aliased in ways that would render the algorithm incorrect. The properties asserted in this predicate are critical to proving that other properties of various nodes are preserved by transitions that modify the heap, or that return nodes to the freelist. The subpredicate $distinctness1\_ok$ ensures that neither $node_p$ nor $head_p$ are part of the queue representation, during intervals where they might be modified or, in the case of $head_p$, added onto the freelist. This allows us to prove that *obj_ok* is preserved across transitions where these nodes are modified or freed.

The subpredicate $distinctness2\_ok$ states that various local pointer variables are not aliased, either between the local variables of one process, or between local variables of different processes. We describe each conjunct in turn.

- Conjunct 3.3 asserts, for distinct processes, that the $node$ variables of each process are not aliased, once the the new node has been allocated, and until it is enqueued.

- Conjunct 3.4 allows us to prove that the $node$ and $head.ptr$ expressions are not aliased when the associated nodes might be modified or placed back on the freelist. The node referenced by $head.ptr$ is only returned to the freelist once the dequeuing process has successfully removed it from the queue. Furthermore, because a node might be removed from the queue, placed on the freelist, and then reallocated to an enqueuing operation, it *is* possible for the $node$ variable of an enqueuing process to point to the same node as the $head.ptr$ variable of a dequeuing process. However, this can only happen if the $head$ version number of the dequeuing process is out-of-date, in which case, we don't need to be able to prove that $head.ptr$ is not aliased by some $node$ variable. Therefore, we allow a $head.ptr$ to alias a node when $head.ver$ is out of date, and the dequeuing process has not yet executed its CAS.

- Conjunct 3.5 allows us to prove that $node$ is not aliased by any $tail.ver$ while either

$$
\begin{aligned}
&distinctness1\_ok(is, cs, f) \mathrel{\hat{=}} \\
&\quad \forall p, i \bullet (pc\_e\_2\_13(cs, p) \land is.Q.Head \leq i \leq is.Q.Tail \Rightarrow \\
&\qquad cs.node_p \neq f(i)) \qquad (3.1) \\
&\quad \land \\
&\quad (pc\_d\_10\_17(cs, p) \land is.Q.Head \leq i \leq is.Q.Tail \Rightarrow \\
&\qquad cs.head_p.ptr \neq f(i)) \qquad (3.2)
\end{aligned}
$$

$$
\begin{aligned}
&distinctness2\_ok(is, cs, f) \mathrel{\hat{=}} \\
&\quad \forall p, q \bullet (p \neq q \land pc\_e\_2\_13(cs, p) \land \\
&\qquad cs.node_p \neq null \land pc\_e\_2\_13(cs, q) \Rightarrow \\
&\qquad\quad cs.node_p \neq cs.node_q) \qquad (3.3) \\
&\qquad \land \\
&\qquad (pc\_e\_2\_13(cs, p) \land pc\_d\_3\_17(cs, q) \Rightarrow \\
&\qquad\quad cs.node_p \neq cs.head_q.ptr \lor \\
&\qquad\quad (pc\_d\_3\_9(cs, q) \land cs.head_p.ver < cs.Head.ver)) \qquad (3.4) \\
&\qquad \land \\
&\qquad (pc\_e\_2\_13(cs, p) \land pc\_e\_6\_17(cs, q) \Rightarrow \\
&\qquad\quad cs.node_p \neq cs.tail_q.ptr \lor \\
&\qquad\quad cs.tail_q.ver < cs.Tail.ver) \qquad (3.5) \\
&\qquad \land \\
&\qquad (p \neq q \land pc\_d\_3\_17(cs, p) \land pc\_d\_10\_17(cs, q) \Rightarrow \\
&\qquad\quad (pc\_d\_10\_17(cs, p) \land cs.head_p.ver < cs.Head.ver) \lor \\
&\qquad\quad cs.head_p.ptr \neq cs.head_q.ptr) \qquad (3.6)
\end{aligned}
$$

Figure 3.18: The predicates $distinctness1\_ok$ and $distinctness2\_ok$.

node may be modified. For similar reasons as those given in the description of Conjunct 2, it is possible for $node$ to alias some $tail$.*ptr*, so we allow *tail*.*ptr* to alias some $node$ when $tail$.*ver* is out of date.

- Conjunct 3.6 allows us to prove that, for distinct processes, the $head$.*ptr* variables of each process are not aliased when either might be returned to the freelist. Again, we allow aliasing to occur during intervals when it doesn't matter.

Proving that the *distinctness_ok* predicates are preserved is fairly straightforward. When a local variable is set to a new value (by an $eval$ or *new*), we need to be able to prove that the value being loaded is not the current value of the other variable in question. For example, when proving that Conjunct 1 of $distinctness1\_ok$ is preserved across transitions labelled by e_1_p (when *p* allocates a new node) we prove (using Conjunct 4 of *obj_ok*, and the definition of the *new* function) that the newly allocated node was not within the queue representation in the prestate of the transition, and thus is not in the representation in the poststate.

Note that the body of the definition of $distinctness2\_ok$ does not mention the intermediate automaton, so the properties asserted by $distinctness2\_ok$ are simple invariants. The simulation relation asserts several other invariants of the concrete automaton. This is because the proofs that these properties are preserved by the simulation relation depend ultimately on assertions about the representation function made in the *obj_ok* predicate. For example, the proof that Conjunct 3.4 of $distinctness2\_ok$ is preserved over transitions labelled by d_2_q (when *q* loads $Head$ into $head$), depends on the fact that $Head$.*ptr* is not aliased by any $node_p$ variable in the prestate. But proving that $Head$.*ptr* is never equal to some $node_p$ depends on the fact that $Head$.*ptr* is never in the freelist, which in turn depends on the fact that nodes within the queue representation are not in the freelist.

Certain invariants of the concrete automaton that currently appear within the simulation relation could be expressed independently, and proved to be invariant using the standard inductive technique (briefly described in Section 2.4 of Chapter 2). However, it is not always obvious which invariants can be proven independently of the properties asserted by *obj_ok*, and there is nothing to be gained by trying to work this out. The simplest approach, which we follow, is to include all these properties in the simulation relation.

The predicate *injective_ok* (Figure 3.19) asserts that the representation function is injective over the domain $[is.Q.Head \ldots is.Q.Tail]$. This ensures that each relevant index of *IntAut* is represented by only one queue node, and that modifications to one node do not falsify properties of nodes corresponding to other indexes. Furthermore, *injective_ok* allows

$$
\begin{array}{l}
\mathit{injective\_ok}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, i, j \bullet \quad \mathit{is}.Tail \leq i \leq \mathit{is}.Head \wedge \\
\qquad\qquad \mathit{is}.Tail \leq j \leq \mathit{is}.Head \wedge f(i) = f(j) \Rightarrow i = j
\end{array}
$$

Figure 3.19: The *injective_ok* predicate.

$$
\begin{array}{l}
\mathit{nds\_ok}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad (\mathit{pc\_e\_2\_13}(\mathit{cs}, p) \Rightarrow \\
\qquad\qquad \neg \mathit{cs}.\mathit{free?}(\mathit{cs}.node_p) \wedge \mathit{cs}.node_p \neq null) \wedge \\
\qquad (\mathit{pc\_e\_3\_13}(\mathit{cs}, p) \Rightarrow \mathit{cs}.node_p \xrightarrow{\mathit{cs}} val.ptr = \mathit{cs}.val_p) \wedge \\
\qquad (\mathit{pc\_e\_4\_13}(\mathit{cs}, p) \Rightarrow \mathit{cs}.node_p \xrightarrow{\mathit{cs}} next.ptr = null)
\end{array}
$$

Figure 3.20: The *nds_ok* predicate.

us to prove that when a node is removed from the queue, it is no longer in the range of the representation function.

It is trivial to prove that *injective_ok* is preserved across transitions that do not modify the representation function. Recall that there is only one class of transitions that modify the representation function: those labelled by $e\_9\_yes_p$, where the process p executes a successful CAS, adding its new node onto the end of the queue. Proving that *injective_ok* is preserved across these transitions is accomplished by using Conjunct 1 of $distinctness1\_ok$ to show that the new node was not in the range of the representation function in the prestate.

The predicate $\mathit{nds\_ok}(\mathit{is}, \mathit{cs}, f)$ (Figure 3.20) expresses properties of each $node_p$ variable in the interval starting when the node is allocated and ending when it is added onto the queue. This is the interval in which the fresh node is initialised. Each assignment to a newly allocated node in the M&S queue algorithm corresponds to a conjunct of *nds_ok* that specifies the value held in that field after the assignment. Showing that *nds_ok* is preserved across transitions that modify the heap or the freelist amounts to doing one of three things:

- In cases where the modification is an update of a newly allocated node, showing that the value being written has the appropriate properties.
- In cases where the modification is a write or CAS that is not one of the initialising writes executed by the process that allocated the node, using *distinctness_ok* to prove

$$
\begin{aligned}
&access\_safety\_ok(is, cs, f) \triangleq \\
&\quad \forall p \bullet (pc\_e\_6\_17(cs, p) \Rightarrow cs.tail_p \neq null \wedge && (3.7)\\
&\quad \neg cs.h.unalloc(cs.tail_p.ptr)) && (3.8)\\
&\quad \wedge && (3.9)\\
&\quad (pc\_d\_3\_17(cs, p) \Rightarrow cs.head_p \neq null \wedge && (3.10)\\
&\quad \neq cs.h.unalloc(cs.head_p.ptr)) && (3.11)\\
&\quad \wedge && (3.12)\\
&\quad (pc\_d\_4\_17(cs, p) \Rightarrow cs.next_p \neq null \wedge && (3.13)\\
&\quad \neq cs.h.unalloc(cs.next_p.ptr)) && (3.14)\\
&\quad \wedge && (3.15)\\
&\quad (cs.pc_p = \mathrm{d\_3} \Rightarrow (cs.head_p.ptr \xrightarrow{cs} next).ptr = null \vee && (3.16)\\
&\quad \neg cs.h.unalloc((cs.head_p.ptr \xrightarrow{cs} next).ptr)) && (3.17)\\
&\quad \wedge && (3.18)\\
&\quad \forall pt \bullet \neg(cs.freelist(pt) \wedge cs.h.unalloc(pt)) && (3.19)
\end{aligned}
$$

Figure 3.21: The *access_safety_ok* predicate.

that the node being modified is not the newly allocated node.

- In cases where the transition places a node on the freelist, proving tha the node being freed is not a newly allocated node, using $distinctness2\_ok$.

The *nds_ok* predicate is used to show preservation of *obj_ok* when an enqueuing process successfully executes its CAS on the `next` field of the tail node, adding it's new node into the queue. For example, Conjunct 2 asserts that when a process *p* attempts to add its new node onto the queue, the value field of that node is equal to the value which *p* is attempting to enqueue. This, in combination with Conjunct 2 of *enqueue_ok*, allows us to prove the crucial property that after the node has been successfully added, the last element in the queue of the intermediate automaton is the last element in the queue of nodes in the concrete automaton (a property asserted by Conjunct 4b of *obj_ok*).

The predicate *access_safety_ok* says that the implementation never dereferences $null$ or

accesses a node that is in $unalloc$, which is important for correct interaction with a memory allocator. *access_safety_ok* asserts, for each program counter value where a dereference can occur (through an invocation of either *update* or $eval$), that the pointer being dereferenced is valid (non-$null$ and allocated). This predicate is not used in the rest of the simulation relation. It is included simply to provide confidence that the M&S algorithm interacts correctly with the system memory allocator.

The *procs_ok* predicate expresses several properties of the local variables of each process, and the relationship between those local variables and the shared variables. Its subpredicates are numnerous and are presented in Figures 3.22, 3.23 and 3.24. The *procs_ok* predicate itself is the conjunction of each of the subpredicates defined in these figures. Many of the subpredicates of *procs_ok* are ad-hoc strengthenings of the simulation relation that were found to be necessary to make the proof go through. We describe only the most important subpredicates of *procs_ok*, including those that are used in the proofs described in Section 3.5.

- *procs_ok_7* says that if *p* is a dequeuing process that has executed D3 (loading the $next_p$ variable) but not yet executed D9 (the CAS), then if *p*'s $head_p$ is accurate, its $next_p$ is also accurate. This is important for proving that during the D9 CAS, the $Head$ variable is correctly modified to point to the next node in the queue.

- *procs_ok_9* says that if the test at D5 failed for some process *p*, then the $next_p$ variable is non-$null$.

- *procs_ok_15* says that if a process *p* is an enqueuing process about to execute the E9 CAS, then the pointer component of $next_p$ is $null$. This ensures that if the E9 CAs is successful, the modified node has a $null$ next value, and is thus the last node in the queue.

- *procs_ok_16* records the fact that for a process enqueueing *p*, the newly allocated node is distinct from the $tail_p$ node.

- *procs_ok_19* states an important property that establishes what the test at E7 guarantees. When a process *p* executes the E7 test, and that test succeeds, then either $next_p$ is not $null$ (in which case *p* will retry the loop), or the version number of $next_p$ is out of date (and thus the next E9 CAS is doomed to fail), or the $next_p$ variable is correct, and the $tail_p$ variable points to the node that is last in the queue (and contains the last value in the abstract queue). Thus, if it is still possible for *p* to successfully execute the E9 CAs

$$
\begin{array}{l}
\mathit{procs\_ok\_1}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad cs.pc_p = \mathrm{d\_4} \wedge cs.Head = cs.head_p \wedge \\
\qquad\qquad cs.next_p.ptr = null \Rightarrow is.empty\_ok_p \\
\mathit{procs\_ok\_2}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_d\_3\_9}(cs, p) \Rightarrow \\
\qquad\qquad cs.head_p.ver < cs.Head.ver \vee cs.head_p = cs.Head \\
\mathit{procs\_ok\_3}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_e\_6\_17}(cs, p) \vee \mathit{pc\_d\_11\_12}(cs, p) \Rightarrow \\
\qquad\qquad cs.tail_p.ver < cs.Tail.ver \vee cs.tail_p = cs.Tail \\
\mathit{procs\_ok\_4}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_e\_7\_9}(cs, p) \Rightarrow \\
\qquad\qquad cs.next_p.ver \leq (cs.tail_p.ptr \xrightarrow{cs} next).ver \\
\mathit{procs\_ok\_5}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_e\_8\_9}(cs, p) \wedge cs.next_p.ptr = null \Rightarrow \\
\qquad\qquad cs.next_p.ver < cs.tail_p.ptr \xrightarrow{cs} next.ver \vee \\
\qquad\qquad (cs.next_p = cs.tail_p.ptr \xrightarrow{cs} next \wedge \\
\qquad\qquad cs.tail_p = cs.Tail \wedge cs.tail_p.ptr = f(is.Q.Tail)) \\
\mathit{procs\_ok\_6}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad (\mathit{pc\_e\_7\_8}(cs, p) \vee cs.pc_p = \mathrm{e\_13}) \wedge cs.next_p.ptr = null \Rightarrow \\
\qquad\qquad cs.tail_p.ver < cs.Tail.ver \vee \\
\qquad\qquad (cs.tail_p = cs.Tail \wedge f(is.Q.Tail) = cs.next_p.ptr \wedge \\
\qquad\qquad cs.tail_p.ptr \neq cs.next_p.ptr) \\
\mathit{procs\_ok\_7}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_d\_4\_9}(cs, p) \wedge cs.head_p = cs.Head \wedge cs.next_p.ptr \neq null \Rightarrow \\
\qquad\qquad cs.next_p.ptr = (cs.Head.ptr \xrightarrow{cs} next).ptr \\
\mathit{procs\_ok\_8}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad cs.pc_p = \mathrm{d\_12} \Rightarrow cs.head_p.ptr = cs.tail_p.ptr
\end{array}
$$

Figure 3.22: Subpredicates of *procs_ok*.

$$
\begin{array}{l}
\mathit{procs\_ok\_9}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_d\_8\_12}(\mathit{cs}, p) \Rightarrow \mathit{cs}.next_p.ptr \neq null \\
\mathit{procs\_ok\_10}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{d\_9} \wedge \mathit{cs}.head_p = \mathit{cs}.Head \Rightarrow \\
\qquad\quad \mathit{cs}.result_p = (\mathit{cs}.next_p.ptr \xrightarrow{\mathit{cs}} val).ptr \\
\mathit{procs\_ok\_11}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{pc\_e\_2\_13}(\mathit{cs}, p) \Rightarrow \mathit{cs}.node_p \neq \mathit{cs}.Tail.ptr \\
\mathit{procs\_ok\_12}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{d\_17} \vee \\
\qquad\quad (\mathit{cs}.pc_p = \mathrm{d\_11} \wedge (\mathit{cs}.head_p.ptr = \mathit{cs}.tail_p.ptr \vee \\
\qquad\qquad \mathit{cs}.tail_p.ver < \mathit{cs}.Tail.ver)) \vee \\
\qquad\quad (\mathit{cs}.pc_p = \mathrm{d\_12} \wedge \mathit{cs}.tail_p.ver = \mathit{cs}.Tail.ver) \Rightarrow \\
\qquad\qquad \mathit{cs}.head_p.ptr \neq \mathit{cs}.Tail.ptr \\
\mathit{procs\_ok\_13}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{e\_6} \wedge (\mathit{cs}.tail_p.ptr \xrightarrow{\mathit{cs}} next).ptr = null \Rightarrow \\
\qquad\quad \mathit{cs}.tail_p.ver < \mathit{cs}.Tail.ver \vee \\
\qquad\quad (\mathit{cs}.Tail = \mathit{cs}.tail_p \wedge \mathit{cs}.tail_p.ptr = f(\mathit{is}.Q.Tail)) \\
\mathit{procs\_ok\_14}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{e\_13} \Rightarrow \mathit{cs}.next_p.ptr = null
\end{array}
$$

Figure 3.23: Subpredicates of *procs_ok*.

at this iteration through the loop, then p has obtained an accurate snapshot of $Tail$ and $Tail{\rightarrow}next$.

3.5 Verifying the Forward Simulation

The forward simulation relation defined here is a large and complicated assertion. The complete simulation proof is correspondingly long and detailed. We will not attempt to describe all of it. First, we outline the structure of the proof. Then, in Section 3.5.1, we presents a careful manual proof that the critical *obj_ok* property is preserved by transitions modelling successful CAS operations.

$$
\begin{array}{l}
\mathit{procs\_ok\_15}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ cs.pc_p = \mathrm{e\_9} \Rightarrow cs.next_p.ptr = null \\
\mathit{procs\_ok\_16}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ \mathit{pc\_e\_6\_13}(cs, p) \Rightarrow cs.node_p.ptr \neq cs.tail_p.ptr \\
\mathit{procs\_ok\_17}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ \mathit{pc\_d\_3\_17}(cs, p) \Rightarrow \\
\qquad\qquad (\mathit{pc\_d\_3\_9}(cs, p) \wedge cs.head_p.ver < cs.Head.ver) \vee \\
\qquad\qquad \neg cs.free?(cs, cs.head_p.ptr) \\
\mathit{procs\_ok\_18}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ cs.pc_p = \mathrm{e\_17} \Rightarrow \\
\qquad\qquad cs.tail_p.ver < cs.Tail.ver \vee \\
\qquad\qquad (cs.tail_p = cs.Tail \wedge f(is.Q.Tail) = cs.node_p \wedge \\
\qquad\qquad\quad cs.tail_p.ptr \neq f(is.Q.Tail)) \\
\mathit{procs\_ok\_19}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ cs.pc_p = \mathrm{e\_7} \wedge cs.next_p.ptr = null \Rightarrow \\
\qquad\qquad cs.tail_p.ver < cs.Tail.ver \vee \\
\qquad\qquad (cs.Tail = cs.tail_p \wedge cs.next_p.ver < (cs.Tail.ptr \overset{cs}{\rightarrow} next).ver) \vee \\
\qquad\qquad (cs.Tail = cs.tail_p \wedge cs.next_p = cs.Tail \overset{cs}{\rightarrow} next \wedge \\
\qquad\qquad\quad cs.tail_p.ptr = f(is.Q.Tail)) \\
\mathit{procs\_ok\_20}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ (cs.pc_p = \mathrm{d\_11} \vee cs.pc_p = \mathrm{d\_12}) \wedge cs.head_p.ptr = cs.tail_p.ptr \Rightarrow \\
\qquad\qquad cs.tail_p.ver < cs.Tail.ver \vee \\
\qquad\qquad (cs.tail_p = cs.Tail \wedge f(is.Q.Tail) = cs.next_p.ptr \wedge \\
\qquad\qquad\quad cs.head_p.ptr = cs.next_p.ptr) \\
\mathit{procs\_ok\_21}(is, cs, f) \mathrel{\hat{=}} \\
\quad \forall p \bullet \ \ cs.pc_p = \mathrm{d\_10} \wedge cs.head_p.ptr = cs.Tail.ptr \Rightarrow \\
\qquad\qquad f(is.Q.Tail) = cs.next_p.ptr \wedge cs.head_p.ptr \neq cs.next_p.ptr
\end{array}
$$

Figure 3.24:

$$\begin{array}{l}
\mathit{procs_ok_22}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{e_17} \Rightarrow \mathit{cs}.tail_p.ver < \mathit{cs}.Tail.ver \vee \neg \mathit{cs}.\mathit{free}?(\mathit{cs}.tail_p.ptr) \\
\mathit{procs_ok_23}(\mathit{is}, \mathit{cs}, f) \mathrel{\hat{=}} \\
\quad \forall\, p \bullet \quad \mathit{cs}.pc_p = \mathrm{e_6} \wedge (\mathit{cs}.tail_p.ptr \xrightarrow{\mathit{cs}} next).ptr \neq null \Rightarrow \\
\qquad\quad \mathit{cs}.tail_p.ver < \mathit{cs}.Tail.ver \vee \\
\qquad\quad (\mathit{cs}.Tail = \mathit{cs}.tail_p \wedge (\mathit{cs}.tail_p.ptr \xrightarrow{\mathit{cs}} next).ptr = f(\mathit{is}.Q.Tail))
\end{array}$$

Figure 3.25: Subpredicates of *procs_ok*.

As in the backward simulation proof, we use a step-correspondence function to determine the intermediate action sequence to choose given a particular transition of the concrete automaton. (Again, we always choose either a single action, or the empty action sequence.) As before, this function maps each external action to itself, and maps all internal actions to the empty action sequence, with the following exceptions: $e\_9\_yes_p$, which models a successful CAS at line E9, is mapped to do_enq_p; $d\_9\_yes_p$ is mapped to $deq\_nonempty_p$; d_3_p is mapped to $observe\_empty_p$; and $d\_5\_yes_p$ is mapped to deq_empty_p.

In contrast to the backward simulation, we do not need to specify a function to calculate the intermediate state, because this is uniquely determined by the intermediate pre-state and the action (if any) chosen because the poststate of each transition is uniquely determined by the action and prestate (i.e., the effect of each action is deterministic).

3.5.1 A Proof Fragment

We now present a careful manual proof that *obj_ok* is preserved across two classes of transitions: those that represent the execution of line E9 by some process, where the CAS is successful; and those that represent the execution of D9, also where the CAS is successful. This is intended to illustrate the use of the representation function, and the style of reasoning we use to verify algorithms that employ dynamic memory.

Successful E9 Transitions

Fix a concrete state *cs* and intermediate state *is* such that $\mathit{FSR}(\mathit{cs}, \mathit{is})$, with representation function f. Fix also a concrete transition $\mathit{cs} \xrightarrow{a} \mathit{cs}'$, where $a = e\_9\_yes_p$ for some p, and let is' and f' be respectively the intermediate state and function determined by the

step-correspondence and witness functions. That is, *is*′ satisfies *is* $\xrightarrow{b}$ *is*′ where $b = do\_enq_p(cs.val_p)$, and $f' = f \oplus \{is.Q.Tail + 1 \mapsto cs.node_p\}$. When we say that part of the simulation relation *holds in the pre-state* (respectively *holds in the post-state*), we mean that it is true for *cs*, *is* and *f* (respectively *cs*′, *is*′, *f*′).

We need to show two things. First, that if the precondition of $e\_9\_yes_p$ holds in the pre-state (i.e., if $cs.pc_p =$ e_9, $cs.next_p = cs.tail_p.ptr \xrightarrow{cs} next$ and $rel(is, cs, f)$) then the intermediate precondition of $do\_enq_p(cs.val_p)$ holds (i.e., $is.pc_p = enq(v)$ where $v = cs.val_p$). In other words, we need to show that *is*′ exists. Second, we need to show that if the concrete precondition and simulation relation hold, then *obj_ok*(is', cs', f').

The first obligation is a straightforward application of $enqueue\_ok$ to process *p* (see Figure 3.16 on page 72). The second obligation is much more complicated. We begin by making some observations about the transition:

$$cs.Tail.ptr = cs.tail_p.ptr = f(is.Q.Tail) \quad (1)$$

$$f'(is'.Q.Tail) = cs.node_p \quad (2)$$

Claim 1 is shown using *procs_ok*_15 to show that $cs.next_p.ptr = null$, and then using *procs_ok*_5 to show that $cs.Tail.ptr = cs.tail_p.ptr = f(is.Q.Tail)$. Claim 2 follows immediately from the construction of *f*′ and the effect of do_enq_p.

Conjunct 1 of *obj_ok* (see Figure 3.15 on page 71) is preserved because $is'.Q.Head = is.Q.Head$, but $is.Q.Head < is.Q.Tail + 1$ (recall that this is a constraint on the set from which *is*.*Q* is drawn). Therefore $is'.Q.Head \neq is.Q.Tail + 1$, so by construction of *f*′ and because Conjunct 1 of *obj_ok* holds in the pre-state, $f'(is'.Q.Head) = f(is.Q.Head) = cs.Head.ptr = cs'.Head.ptr$.

For Conjunct 2, by construction of *f*′ and the effect of do_enq_p, we have $f'(is'.Q.Tail) = f'(is.Q.Tail + 1) = cs.node_p$ Moreover, by Conjunct 3 of *nds_ok*, $cs.node_p \xrightarrow{cs} next.ptr = null$. By *procs_ok*_16, $cs.tail_p.ptr \neq cs.node_p$, so $cs.node_p \xrightarrow{cs'} next.ptr = null$, and thus

$$f'(is'.Q.Tail) \xrightarrow{cs'} next.ptr = cs.node_p \xrightarrow{cs'} next.ptr = null$$

We show that Conjunct 3b holds in the post-state, arguing each sub-conjunct in turn.

$$
\begin{array}{rcll}
f'(is'.Q.Tail) & = & cs.node_p & \text{by (ii) above} \\
 & = & cs.tail_p.ptr \xrightarrow{cs'} next.ptr & \text{by construction of } cs' \\
 & = & cs.Tail.ptr \xrightarrow{cs'} next.ptr & \text{by Claim 1 above} \\
 & = & cs'.Tail.ptr \xrightarrow{cs'} next.ptr & \text{because } cs'.Tail = cs.Tail \\
 & & & \\
cs'.free?(cs'.Tail.ptr) & = & cs.free?(cs'.Tail.ptr) & cs'.free? = cs.free? \\
 & = & cs.free?(cs.Tail.ptr) & cs'.Tail = cs.Tail \\
 & = & cs.free?(f(is.Q.Tail)) & \text{by (i) above} \\
 & = & false & \text{conjunct 4c with} \\
 & & & i = is.Q.Tail
\end{array}
$$

Now by Claim 1, $cs.Tail.ptr = f(is.Q.Tail)$, so by Conjunct 4d applied to $is.Q.Tail$, $cs.Tail.ptr \neq null$. Therefore, $cs'.Tail.ptr \neq null$ by the effect of the e_9_yes transition, so the third conjunct is preserved. For the last conjunct of 3b we have

$$
\begin{array}{rcll}
f'(is'.Q.Tail) & = & cs.node_p & \text{by (ii) above} \\
 & \neq & cs.tail_p.ptr & \text{by } procs\_ok\_16 \\
 & = & cs.Tail.ptr & \text{by (i) above} \\
 & = & cs'.Tail.ptr &
\end{array}
$$

We prove Conjunct 4 by cases. For any i such that $is'.Q.Head \leq i \leq is'.Q.Tail$, either $i = is.Q.Tail+1$ or $is.Q.Head \leq i \leq is.Q.Tail$. We treat the case in which $i = is.Q.Tail+1$ first. $is.Q.Tail + 1 = is'.Q.Tail$ so there is nothing to prove for Conjunct 4a. For Conjunct 4b we have

$$
\begin{array}{rcll}
is'.Q.seq(i) & = & cs.val_p & \text{by effect of } do\_enq_p \\
 & & & \text{and } enqueue\_ok \\
 & = & cs.node_p \xrightarrow{cs} val.ptr & \text{by } nds\_ok \\
 & = & cs.node_p \xrightarrow{cs'} val.ptr & \text{by effect of } e\_9\_yes_p \\
 & = & f'(i) \xrightarrow{cs'} val.ptr & \text{by (ii) above}
\end{array}
$$

4c and 4d follow from *nds_ok* and (ii) above.

It remains to consider the case in which $is.Q.Head \leq i \leq is.Q.Tail$. For 4a, we further distinguish the cases in which $i = is.Q.Tail$ and $is.Q.Head \leq i < is.Q.Tail$. For the first case, we have

$$
\begin{array}{rcll}
f'(i) \xrightarrow{cs'} next.ptr & = & f(i) \xrightarrow{cs'} next.ptr & \text{because } i \neq is.Q.Tail + 1 \\
 & = & cs.tail_p.ptr \xrightarrow{cs'} next.ptr & \text{by (i) above} \\
 & = & cs.node_p & \text{by effect of } e\_9\_yes_p \\
 & = & f'(is'.Q.Tail) & \text{by (ii) above} \\
 & = & f'(i+1) & \text{by effect of } do\_enq_p
\end{array}
$$

If $is.Q.Head \leq i < is.Q.Tail$, (4a) follows directly if we can show that $f(i) \neq cs.tail_p.ptr$. This is because $i \neq is.Q.Tail$ and so (4a) holds for i in the pre-state and

$$\begin{array}{rcll}
(f(i)\xrightarrow{cs}next).ptr & \Rightarrow & (f(i)\xrightarrow{cs'}next).ptr & \\
= f(i+1) & & = f(i+1) & \text{given } f(i) \neq cs.tail_p.ptr \\
& \Rightarrow & (f'(i)\xrightarrow{cs'}next).ptr & i < is.Q.Tail \text{ so} \\
& & = f'(i+1) & f'(i) = f(i) \text{and} \\
& & & f'(i+1) = f(i+1)
\end{array}$$

But if $f(i) = cs.tail_p.ptr$ then by *injective_ok* and (i) above, we have $i = is.Q.Tail$, contradicting the hypothesis that $i < is.Q.Tail$.

(4b), (4c) and (4d) all follow for i from the fact that these conjuncts held in the pre-state and that because $i \neq is.Q.Tail + 1$, $is'.Q.seq(i) = is.Q.seq(i)$ and $f'(i) = f(i)$. Moreover, no val fields, nor *free?* are modified by the transition.

Successful D9 Transitions

We now present a careful manual proof that *obj_ok* is preserved across transitions that represent the execution of D9 by some process, where the CAS is successful. As before, fix a concrete state cs and intermediate state is such that $FSR(cs, is)$, with representation function f. Fix also a concrete transition $cs \xrightarrow{a} cs'$, where $a = e\_9\_yes_p$ for some p, and let is' and f' be respectively the intermediate state and function determined by the step-correspondence and witness functions. That is, is' satisfies $is \xrightarrow{b} is'$ where $b = deq\_nonempty_p$ and $f' = f$. We need to show that if the precondition of $d\_9\_yes_p$ holds in the pre-state and $rel(is, cs, f)$ then $obj\_ok(is', cs', f')$.

As before, we need to show that the intermediate precondition holds (presented in Figure 3.12 on page 65). The first part of that precondition, that $is.pc_p = deq$ is true by a simple application of $dequeue\_ok$ to p (see Figure 3.16 on page 72). The second, that $\neg(is.deq)$, is more complicated. By *procs_ok_9*, *procs_ok_7* and the precondition of $e\_9\_yes_p$, we have

$$cs.next_p.ptr \neq null \qquad \text{(i)}$$

$$cs.next_p.ptr = cs.Head.ptr\xrightarrow{cs}next.ptr \qquad \text{(ii)}$$

Assume for the sake of contradiction, that $(is.deq)$, ie., $is.Q.Head = is.Q.Tail$. Then , by Conjunct (2) of *obj_ok*, we have $f(is.Q.Head)\xrightarrow{cs}next.ptr = null$. Also, by Conjunct (1) of *obj_ok*, $f(is.Q.Head) = cs.Head.ptr$. So,

$$\begin{array}{rcll}
null & = & f(is.Q.Head)\xrightarrow{cs}next.ptr & \\
& = & cs.Head.ptr\xrightarrow{cs}next.ptr & \\
& = & cs.next_p.ptr & \text{By ii above.}
\end{array}$$

But this contradicts i above, so we have

$$is.Q.Head = is.Q.Tail \qquad \text{(iii)}$$

We now show that each conjunct of *obj_ok* holds in the poststate. First, Conjunct 1. Observe that by Conjunct 4a of *obj_ok*, and iii above, we have

$$f(is.Q.Head) \xrightarrow{cs} next.ptr = f(is.Q.Head + 1) \qquad \text{(iv)}$$

$f'(is'.Q.Head)$	$=$	$f(is.Q.Head + 1)$	Definition of f' and $deq\_nonempty$ transition
	$=$	$f(is.Q.Head) \xrightarrow{cs} next.ptr$	iv above
	$=$	$cs.Head \xrightarrow{cs} next.ptr$	Conjunct 1 of *obj_ok*
	$=$	$cs.next_p.ptr$	ii above
	$=$	$cs.Head.ptr$	Definition of $e\_9\_yes_p$ transition

It is easy to see that Conjuncts 2, 3a and 3b are all preserved. None of the variables of either automaton or fields that are mentioned in these conjuncts are modified by the concrete or intermediate transitions.

For Conjunct 4, fix an *i* such that $is'.Q.Head \leq i \leq is'.Q.Tail$. Because $is'.Q.Head = is.Q.Head + 1$ and $is'.Q.Tail = is.Q.Tail$, we may apply Conjunct 4 to *i* and obtain that 4a-4d all hold in the prestate. Observe that $f' = f$ and none of the variables or fields mentioned in 4a-4d are modified by the concrete or intermediate transitions. Therefore, 4a-4d must also hold in the poststate.

3.6 Related Work

There have been several variations on the M&S queue, designed to work in various contexts. Some are less general in the sense that they depend on unusual properties of the runtime environment to guarantee correctness ([Jav] depends on garbage collection, [Lee07] depends on properties of a realtime scheduler). Others allow non-linearisable behaviour [Lee07, FOL05].

The M&S queue has been used as a case study in previous work on the application of formal methods to concurrent algorithms [YS03, AC05, WS05, BAM06, BAM07]. The remainder of this section describes this work.

The authors of [YS03] present an automatic verification of certain properties of the M&S queue, using a model-checking technique. This technique using three-valued logic (where propositions can take the values *true*, *false* and *unknown*) to represent uncertainty. They call their technique *3VMC*, for *three-valued model checking*.

The 3VMC technique is capable of constructing abstractions of concrete systems with unbounded state and of using this abstraction to check invariants of the original system. Typically, the user defines predicates over the states of the concrete system that describe properties relevant to the verification. The abstraction technique then uses these predicates and others that are defined automatically to construct the abstraction. Like other kinds of model-checking, the technique is interesting because usually only a very limited form of interaction with a human is required to verify a given algorithm. Descriptions and applications of the 3VMC technique can be found in [Yah01, MYRS05].

[YS03] verifies certain properties of the M&S queue. These properties are taken from the paper presenting the original algorithm [MS96b] and are listed here.

1. The linked list of nodes is always connected.
2. Nodes are only inserted after the last node of the linked list.
3. Nodes are only deleted from the beginning of the linked list.
4. $Head$ always points to the first node in the linked list.
5. $Tail$ always points to a node in the linked list.

[YS03] presents a formalisation of these properties in the logic of the tool used for the verification.

The verification presented in this chapter proves that the M&S queue satisfies the behavioural properties appropriate to the queue datatype (as specified by the abstract automaton). The approach of [YS03] verifies important invariant properties of the M&S queue, but stops short of a full behavioural verification. No argument is presented, either in [MS96b] or [YS03] as to why these properties should be considered sufficient for the correctness of the queue implementation. There are several ways in which the M&S queue could satisfy these properties, but the queue be incorrect, nevertheless. This is a consequence of the fact that behavioural issues are simply not discussed. A *dequeue* may return `false`, even when the list is nonempty. Nodes might only be inserted at the end of the list, but it might be possible for an *enqueue* to complete without inserting a node. The linked-list may be connected but circular.

Apart from these issues, it is unclear whether the authors of [YS03] have actually verified these properties when the queue is accessed by an unbounded number of enqueuing and dequeuing processes. They report verifications showing that the properties are invariant when the queue is accessed concurrently by one enqueuing process and one dequeuing process; and by an unbounded number of concurrent enqueuers; and finally by an unbounded number of concurrent dequeuers. However, no verification of the properties under concurrent access by both unbounded enqueuers and some fixed (nonzero) number of dequeuers is reported, or vice-versa. Contrast this with our verification, which proves that the M&S queue is correct, for an unbounded number of concurrent enqueuers and dequeuers, relative to a behavioural specification of the queue's safety properties, given by linearisability and the canonical automaton.

Work presented in [ARR$^+$07b] applies 3VMC to tackle the problem of proving linearisability directly. They verify several implementations of concurrent data structures, including the Treiber stack and the M&S queue, using a technique they call *comparison under abstraction*. Roughly speaking, they run the concurrent implementation simultaneously with a sequential implementation that has a similar layout in the heap. At a putative linearisation point in the execution of the concurrent implementation, the corresponding operation is executed atomically on the sequential implementation. An isomorphism from the heap of

the concurrent implementation to that of the sequential implementation (with some bounded number of nodes not included in the domain of the isomorphism) is then used to infer that the operation is correctly linearised. The 3VMC technique is used to ensure that the verified algorithms are correct for an unbounded number of nodes.

This work has the advantage over [YS03] in that it attacks the question of behavioural correctness directly. Proving behavioural correctness of a concurrent data structure over an unbounded heap without human intervention is a significant achievement. However, the verification only works for a bounded number of concurrent operations. All their examples prove correctness of the implementation for between two and four concurrent operations. In the case of the M&S queue, their technique succeeds in verifying just two concurrent threads. The goal of the authors is to leverage the 3VMC technique to verifying data structures under an unbounded number of concurrent operations.

The authors of [ARR+07b] describe interesting limitations in their ability to assign linearisation points to operations, related to prophetic linearisation, amoung other issues. Each procedure implementing an operation is assigned a particular statement in the code that acts as the linearisation point for that operation. When this statement is executed, that particular operation on the sequential data structure is triggered. The linearisation point of each operation cannot be a statement executed in another operation. Further, the question of whether a particular occurrence of a statement in an execution is a linearisation point cannot be answered by looking into the future of the execution. All three of the verifications presented in this thesis feature linearisation points that either are in other operations or that depend on future knowledge. Therefore, these are significant restrictions.

Like us, the authors of [AC05] apply deductive techniques to the verification of the M&S queue. They formally derive a variant of the M&S queue, using a notation and methodology called *Event-B* [Abr03, ACM03], which is a version of the B Method [Abr96] that includes support for reasoning about concurrency. Event-B is a *refinement* based approach, where successive algorithms (called *refinements*) are constructed, beginning with a specification, and ending with an implementation. Each new refinement is shown to implement the previous one, using rules for the correctness-preserving transformation of one refinement into another. The authors of [AC05] use a proof assistant, called *Click ' Prove* to discharge the proof obligations that arise from the application of these rules.

Their work is similar to ours in several respects. The specification and proof is based on a formal notation; they construct their proofs using a mechanical theorem prover; and they prove a behavioural property of the algorithm: that its externally observable behaviour is indistinguishable from that of the specification.

However, there are two important differences. First, rather than using an abstract specification of a linearisable queue as we do, they use as their specification a model that is essentially the M&S queue as if all operations were executed atomically. That is, the model has a linked-list of nodes, `Head` and `Tail` variables that range over these nodes, and `enqueue` and `dequeue` operations that modify these variables and nodes, and execute without interleavings. It seems likely that they could have begun with a more abstract model built directly from a simple sequential specification, and thus that their use of an implementation

dependent specification does not reflect a fundamental limitation of the Event-B methodology. However, showing that a linked representation of a sequence of values is correct is very much a nontrivial task, and it would have been interesting to see how it could be achieved in the Event-B framework.

In any case, a more important difference is the way in which their work deals with prophetic linearisation. Rather than determining whether the queue is empty based solely on reading `Head`, the dequeue operation checks whether `Head` and `Tail` point to the same node, and if so, whether the node referenced by `Head` has a `null` next pointer. If both tests succeed, then the queue is empty when `Tail->next` is read. Because no further tests are required after the read, it can serve as a linearisation point, and this can be determined as such just by looking at the current state. Thus, they avoid dealing with the prophetic linearisation of both the original M&S queue, and our optimisation.

The algorithm they ultimately derive has a significant difference from the M&S queue in that version numbers are associated abstractly with queue nodes, rather than being associated with locations (locations do not contain pointers and version numbers). Because of this difference, it is not clear how to implement their algorithm directly on an actual machine. In their model, the CAS operation checks that the version number *of the node* has not changed when attempting a modification of the `Head` or `Tail` pointer. CAS can in reality, only check the number associated with the location being modified, not some node referred to from that location.

Recent work has attended to the question of applying reductions to the executions of nonblocking algorithms [WS05, Gro08], which has used the M&S queue as a case study. This work is based on the idea that the order pairs of read, write and CAS operations in an execution can often be reversed, without changing the observable behaviour of the execution.

[WS05] applies a static analysis technique to the problem of verifying the M&S queue. Their verification works in two phases. They prove manually that a version of the m&S queue algorithm is correct, under the assumption that certain blocks of code are always executed atomically. Then they use a static analysis technique to show that, for any state that can be reached by an execution of complete operations where these blocks are not executed atomically (but are executed to completion), there is some execution where these blocks are executed atomically that reaches the same state. The second phase is the primary contribution of the work, and it suggests that aspects of similar verifications relating to complicated interleavings can be completed automatically. The first phase of the verification effort could be completed by simulation.

Significant work has been done on applying the backward simulation technique to the verification of other algorithms and protocols. We defer detailed discussion of this work until Chapter 4. However, verifications presented in [SAGG+93] and [Smi97] have the same structure as the verification presented in this chapter. Like us, they use an intermediate automaton to capture the ”backwards” behaviour of the implementation, which admits a simple backwards simulation to the specification automaton. Forward simulation is then used to show that the concrete automaton implements the intermediate automaton. The verification presented in Chapter 4, which treats an algorithm known as *Snark* that implements a double-

ended queue object, departs from this pattern in that the backward simulation is substantially more complicated. This reflects the greater complication of the prophetic linearisation involved. In the M&S queue, the actions that are linearisation points given certain future events are linearisation points for the process that takes the action. Moreover, the operations that are linearised in this way do not modify the value of the queue. In contrast, the Snark algorithm has the property that an action of a process p can be a linearisation point of an operation of some other process $q \neq p$, and q's operation can modify the double-ended queue.

Relaxed Memory Models

Another attempt at automatic verification of the M&S queue (among other algorithms) is presented in [BAM06] and [BAM07]. Their technique is interesting, because it analyses the behaviour of the M&S queue in the context of *relaxed memory models*. Before describing their contributions, we briefly introduce the concept of relaxed memory models.

Shared-memory architectures often do not implement an abstraction where all memory operations appear to take effect to all processes at once. Rather, in many systems it is possible for processes to observe memory operations in different orders from one another. The architecture provides some guarantee about what kind of operation orders might be observed by a process running on the system. This guarantee is called the *memory model*, and memory models in which it is possible for processes to observe operations in different orders are called *relaxed memory models*, or just *relaxed models*. Implementations of shared-memory systems that provide relaxed models can benefit from important optimisations that greatly reduce memory-operation latency in common cases. However, they have the disadvantage that they exhibit behaviours not possible in more intuitive models, and thus present a significant problem for verification.

The work of [BAM06] and [BAM07] is based on applying decision procedures for satisfiability of boolean propositions (that is, formulae without predicates or quantification). [BAM07] describes an application called *CheckFence* that implements their technique. Given a sequential specification of a datatype and a set of operations on that datatype, called a *test*, CheckFence generates a boolean formula describing the possible behaviours of the datatype when the given operations are executed. These operations are only partially ordered by the test, and may execute in parallel. Also, given an algorithm (expressed in a subset of the C language) and a formal description of a memory model, CheckFence generates a boolean formula describing the possible executions of the algorithm, under the given memory model. Finally, CheckFence determines whether any of the algorithm's executions fail to meet the allowed behaviours. This amounts to checking whether the first formula can be false while the second formula is true: a boolean satisfiability problem.

The largest test reported in [BAM07] involved 12 operations executed by two concurrent processes and took several minutes (the test was carried on the M&S queue). One test on the Snark algorithm involving eight concurrent operations took about an hour. The tests are themselves quite small, involving 200 to 300 memory accesses at most. Graphs presented in [BAM07] show a near exponential increase in runtime and memory use as the number of

memory accesses increases. Therefore, scaling the technique up to larger test sizes may be difficult. Their approach is directed towards bounded testing, rather than full verification, and is thus orthogonal to our work.

3.7 Concluding Remarks

The techniques used in the construction of the forward simulation in this chapter are applied in Chapter 6 to the verification of a novel implementation of the LL/SC primitive. We review two important aspects of the forward simulation presented in this chapter that will reappear in that setting.

Dealing with the possibility of aliasing is critical to any verification of pointers. We must be able to show that updates to heap objects accessed from one pointer variable do not destroy properties of objects accessed from another variable. In the forward simulation presented in this chapter, aliasing is constrained by the *distinctness_ok* and *injective_ok* predicates, as well as some of the *procs_ok* predicates. Very similar techniques are used in the verification of Chapter 6.

Our heap model is idealised in the sense that we allow pointers to be dereferenced, even in cases where the pointer may be `null` or unallocated. (However, recall that we prove that no process ever deallocates such a pointer.) This simple approach is inadequate for the verification in Chapter 6, because explicit deallocations occur in that algorithm. The heap model in that verification is very similar to the one used here. However, the $eval$ and *update* functions cause the heap to enter an ”error state” when applied to an invalid pointer. This makes the proof more complicated. Chapter 6 describes how we deal with this additional complexity.

Chapter 4

Another Application of Backward Simulation

This chapter presents part of the verification of a nonblocking algorithm known as the *Snark algorithm* [DFG+00]. The Snark algorithm is a lock-free implementation of the double-ended queue datatype (henceforth *deque*) that uses the DCAS synchronisation primitive. A deque is like a stack or queue in that it provides operations on a sequence of values. However, a deque generalises both datatypes in that it provides operations that add values to or remove values from *either* end of the sequence.

During an earlier verification attempt [Doh03] it was discovered that the Snark algorithm as originally presented in [DFG+00], is incorrect. As we shall see, the corrected version, first described in [DDG+04], presents several challenges to verification. Principal among these challenges is the fact that the corrected version exhibits prophetic linearisation. This prophetic linearisation is interesting partly because an operation with a future-dependent linearisation point can have an effect that is visible to other operations. This is in contrast to the M&S queue, in which the only operations with future dependent linearisation points do not have a visible effect. As we shall see, a backward simulation used in the verification of the corrected algorithm must account for the fact that the sequence of values in a state of the implementation can be different from the sequence of values in a related state of the specification.

Because this chapter is concerned with the verification of the corrected version of the algorithm, we use the name *Snark algorithm* (or sometimes, just *Snark*) to refer to this corrected version. We describe the backward simulation proof between the specification automaton and an intermediate automaton that captures the Snark algorithm's "backwards" behaviour. The proof is significantly more complicated than other backward simulation proofs that we are aware of (such as [SAGG+93, DGLM04, CG05]), which we believe is a consequence of Snark's particular kind of prophetic linearisation. Backward simulations tend to be trickier to verify than forward simulations, but because they have been rarely necessary in practice, there seems to be a lack of substantial examples in the literature.

One motivation for our interest in backward simulation is the relatively large number of nonblocking algorithms that would require the use of backward simulation, if they were to be verified using simulation based techniques. ([MS98a, DDG+04] have already been discussed. Algorithms in [Blo88, LMS03b, Fra03, HHL+06] provide other examples.) Furthermore, there is a need to develop techniques for the verification of nonblocking algorithms that exhibit complicated patterns of prophetic linearisation. The elimination queue of [MNSS05] provides a good example.

A complete verification of the Snark algorithm would involve the definition of a forward simulation from a concrete automaton modelling the actual algorithm to the intermediate automaton, along with a proof that it is a forward simulation. We do not produce or prove such a forward simulation in this thesis. The techniques required to do so are essentially the same as were used in the proof of the forward simulation presented in Chapter 3, and the proof itself is very long and tedious. The novel and interesting aspects of the verification are the construction of the intermediate automaton, and the backward simulation.

Section 4.1 describes the Snark algorithm presented in [DDG+04], and explains why it exhibits prophetic linearisation. This is intended to motivate the backwards simulation that we describe in this chapter in detail. Section 4.2 describes the intermediate and abstract automata used in the verification. Section 4.3 presents the backward simulation and Section 4.4 describes important aspects of the proof that it is in fact a simulation relation.

4.1 DCAS and the Snark Algorithm

This section describes the Snark algorithm, as presented in [DDG+04]. The deque datatype encapsulates a sequence of values. A deque supports four operations: two operations `pushLeft` and `pushRight` that each add a value onto one end of the sequence, and `popLeft` and `popRight` that each remove and return a value from one end of the sequence. As the names suggest, the `pushLeft` operation adds a value to the end of the sequence from which `popLeft` removes a value, and similarly for `pushRight` and `popRight`.

Snark uses the *double-compare-and-swap* (DCAS) synchronisation primitive, a generalisation of CAS that operates on two independent locations. The DCAS primitive was first mentioned in Chapter 1, but we describe it again here for convenience. Figure 4.1 presents pseudocode describing the semantics of DCAS. The DCAS operation takes as arguments two locations, two expected values, and two new values. The two locations are independent, they do not need to be adjacent. The new values are written into the two locations if and only if *both* locations each contain the corresponding expected value.

DCAS has been used in the implementation of an experimental, nonblocking operating system kernel for the Motorola 68030 processor [MP91], which is one of the very few processors that supports DCAS. Later work produced DCAS-based techniques for the transformation of sequential data structures into functionally equivalent nonblocking data structures [Gre02], and lock-free reference counting [DMMm01] (which we discuss in Chapter 5).

Significant attention has been given to the development of nonblocking deque imple-

```
boolean DCAS(val *addr1, val *addr2,
             val old1, val old2,
             val new1, val new2) {
  atomically {
    if ((*addr1 = old1) &&
        (*addr2 = old2)) {
      *addr1 := new1;
      *addr2 := new2;
      return true;
    } else return false;
  }
}
```

Figure 4.1: Semantics of the DCAS operation.

mentations that use DCAS [ADF+00, DFG+00, MMm02, DDG+04]. Because the deque datatype provides push and pop operations at *both* ends of the sequence, it was thought [ADF+00, DDG+04] that implementing the deque datatype would provide a good test case for examining the utility of DCAS in the design of advanced data structures. The Snark algorithm improved on previous proposals by requiring fewer DCAS operations in the best case. However, the outcome of these experiments with deques, together with certain undesirable properties of other algorithms that use DCAS, suggest that the DCAS operation does not substantially extend the range of datatypes that admit simple and efficient implementations [DDG+04]. Partly for these reasons, interest in DCAS-based data structures has waned in recent years. However, as we shall see, the Snark algorithm provides an interesting case study in the verification of nonblocking algorithms.

4.1.1 The Algorithm

We turn now to a description of the Snark algorithm. The declarations and initial state for the Snark algorithm are presented in Figure 4.2. The Snark algorithm uses a doubly-linked list in which each node is connected to its neighbours through its `L` and `R` fields. The `V` field of a node contains a value. The Snark algorithm has two shared pointer variables, known as *hats*, called respectively `LeftHat` and `RightHat`. These variables are used to access either end of the doubly-linked list. Snark relies on a garbage collector to recycle unreachable storage.

Figure 4.3 illustrates a deque containing two elements. When the deque is not empty, `LeftHat` (resp. `RightHat`) points to the leftmost (resp. rightmost) node that contains an unpopped value. Snark uses sentinel nodes on either end of the deque to allow operations to detect whether the deque is empty. A value in the `V` field of a sentinel node is not part of the sequence of values contained in the deque. Observe that the inward pointers of the sentinels are self-pointers. We say that a node `nd` is *left-dead* (resp. *right-dead*) when `nd->L` (resp.

```
1. structure Node {
2.   Node *L;
3.   Node *R;
4.   val V;
5. }

Node *Dummy, *LeftHat,
     *RightHat;
```

```
initialise() {
1. Dummy := new Node();
2. Dummy->L := Dummy;
3. Dummy->R := Dummy;
4. LeftHat := Dummy;
5. RightHat := Dummy; }
```

Figure 4.2: The declarations and initial state for the Snark algorithm.

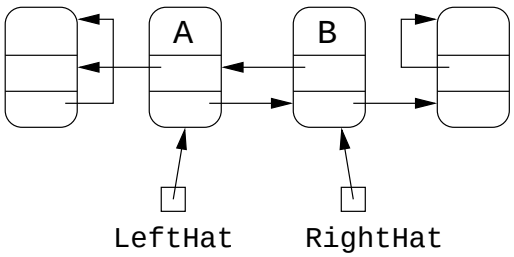
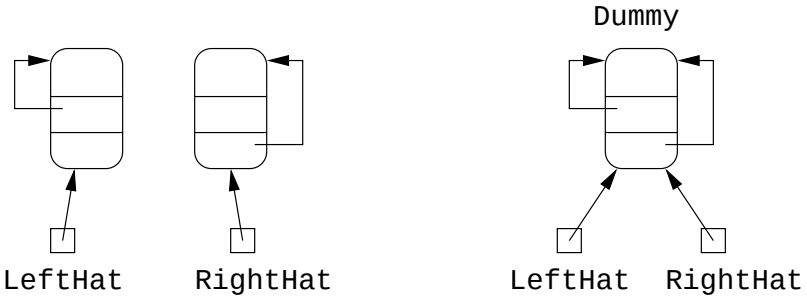


Figure 4.3: A deque containing two elements.



(a) (b)

Figure 4.4: Two empty deques. (a) Generic empty state. (b) Special case empty state using the `Dummy` node.

nd->R) is equal to nd. The following key properties together characterise the states of the Snark algorithm that represent nonempty deques.

1. LeftHat is not left-dead, ie.,
   LeftHat->L != LeftHat.
2. RightHat is not right-dead, ie.,
   RightHat->R != RightHat.
3. The node to the left of LeftHat is right-dead, ie.,
   LeftHat->L->R = LeftHat->L.
4. The node to the right of RightHat is left-dead, ie.,
   RightHat->R->L = RightHat->R.

These properties imply that if either LeftHat is left-dead, or RightHat is right-dead then the deque is empty. In fact, the Snark algorithm guarantees that if one hat points to a node with such a self-pointer, then so does the other. This implies that the empty deque can be represented by a variety of different configurations, all sharing the property that LeftHat is left-dead and RightHat is right-dead. Figure 4.4 illustrates two such states: (a) illustrates the generic case; (b) illustrates the construction of an empty deque using a pointer constant Dummy. Snark guarantees that in every state Dummy->L = Dummy->R = Dummy. The representation illustrated in Figure 4.4(b) is used as an initial state, and can be reached by removing values from the deque. Dummy is used during push operations whenever a left- or right-dead node is needed to maintain properties (3) and (4) above.

We now describe the pushRight operation; the pushLeft operation is symmetric. We first describe pushRight under the assumption that the deque is not empty during the operation. In that case, the operation adds a value onto the deque by doing the following:

1. The operation allocates a new node, the V field of the new node is set to the value being added to the deque, and the R field of the fresh node is set to Dummy.
2. RightHat is set to point to the new node, and the R field of the previously rightmost node (the previous value of RightHat) is set to point to the new node.

The R field of the new node is set to Dummy so that when the new node is added onto the deque, the right sentinel is left-dead.[1] The modifications to the V and R fields of the new node can be accomplished using ordinary writes, because the new node is not visible to any process except the process that allocated the node. As we describe shortly, the modification of the RightHat and R field of the rightmost node is accomplished atomically using a DCAS.

When the deque is empty, a pushRight operation sets both the L and R fields of the new node to Dummy. Then, a DCAS is used to set LeftHat and RightHat to point to the new node. This implies that when the node is added into the doubly-linked list, then both the left- and right-sentinels are right- and left-dead, respectively.

[1] Because other modifications to the deque may occur during the pushRight operation, it is not safe to simply read the R field of the rightmost node, and then set the R field of the new node to that value.

```
H1. rtype pushRight(val v) {
H2.   nd := new Node();
H3.   nd->R := Dummy;
H4.   nd->V := v;
H5.   while (true) {
H6.     rh := RightHat;
H7.     rhR := rh->R;
H8.     if (rhR = rh) {
H9.       nd->L := Dummy;
H10.      lh := LeftHat;
H11.      if (DCAS(&RightHat,
                   &LeftHat,
                   rh, lh,
                   nd, nd))
H12.        return "ok";
H13.    } else {
H14.      nd->L := rh;
H15.      if (DCAS(&RightHat,
                   &rh->R,
                   rh, rhR,
                   nd, nd))
H16.        return "ok";
H17.    }
H18.  }
H19.}
```

Figure 4.5: Pseudocode for the `pushRight` operation.

Figure 4.5 presents pseudocode for the `pushRight` operation. (Pseudocode for the symmetric `pushLeft` operation is presented in Figure 4.8 on page 102.) A process *p* executing `pushRight` allocates the new node and stores it in the variable `nd` . Then *p* sets `nd->R` to point to `Dummy` (H3). Next, *p* sets `nd->V` to the value that is being pushed (H4). Now, *p* loads the current value of `RightHat` into the local variable `rh` (H6). Recall that if `rh` points to a right-dead node, and `RightHat = rh`, then the deque is empty. The conditional at line H8 tests whether the deque may be empty, and if the test succeeds, *p* sets `nd->L` to `Dummy` (H9). After loading the current `LeftHat` (H10), *p* attempts to set both the `LeftHat` and `RightHat` to the new node using DCAS (H11). If this succeeds, the value `v` has been successfully added onto the deque. If the DCAS fails, it must be that some other process has updated the deque since *p* loaded either of the left- and right-hats. In this case, *p* retries the loop beginning at H5.

If the test at line H8 fails, then either the deque is not empty or `RightHat` has been modified since *p* loaded it into `rh`. Is either case, *p* attempts to splice the node onto the right

end of the deque. It sets `nd->L` to the value that it loaded from `RightHat` at line H6 (H14), and then attempts the DCAS at line H15. If this DCAS succeeds, it changes the `RightHat` variable to point to `nd` and sets `rh->R` (the rightwards field of the old `RightHat`) to `nd`. This adds the value `v` onto the deque. If the DCAS fails, *p* retries the loop.

We now describe the `popRight` operation; `popLeft` is symmetric. We first describe the `pushRight` operation under the assumption that the deque is not empty during the operation. In that case, the operation removes a value by making the rightmost node left-dead, and setting `RightHat` to point to the node immediately to the left of the rightmost node (i.e., setting `RightHat` to the previous value of `RightHat->L`). This is accomplished atomically using a DCAS. We say that the node which has been made left-dead by the DCAS has been *removed* from the doubly-linked list.

Absent any concurrent modification, the `popRight` operation can now return the value in the `V` field of the node that has just been removed. However, it is possible for two concurrent pop operations to both remove the same node from the doubly-linked list. Because of this, a pop operation must *secure* the node that it just removed, before the value can be returned. We describe at the end of this section how two processes can remove the same node, and how a process can secure the node.

Figure 4.6 presents pseudocode for the `popRight` operation. (Pseudocode for `popLeft` is presented in Figure 4.9 on page 103.) A process *p* executing `popRight` begins by loading `RightHat` into the local variable `rh` (P3), and `rh->L` into the local variable `rhl` (P4). Then, it tries to determine if the deque is currently empty. First it tests whether `rh->R` is right-dead (at line P5). If this test succeeds, it checks whether `RightHat` still has the same value as it did when *p* executed line P3 (P6). The Snark algorithm has the property that once a node contains a self-pointer in its `L` or `R` fields, it always has a self-pointer in that field (at least until the node is recycled by the garbage collector). Thus, because `rh` was right-dead when *p* executed line P5, `RightHat` is right-dead if `RightHat = rh` when *p* executes line P6. This implies that if the test at line P6 succeeds, then the deque is empty, so *p* returns an indication that it found the deque empty. If the test at line P6 fails, *p* retries the loop, by executing line P3 again.

If the test at line P5 fails, *p* attempts to remove a node from the right side of the doubly-linked list. Using a DCAS it attempts to change `RightHat` to point to `RightHat->L` and make `rh` (the old `RightHat`) left-dead (P8).

It is possible for two processes to successfully execute the DCAS at line P8 in such a way that the same node is removed from the doubly-linked list *twice*.[2] Because of this, each process is required to *secure* the node that it removes from the doubly-linked list, preventing another process from returning the value associated with that node. We now describe how this is achieved. The Snark algorithm has a special value `secured` that is never pushed onto the deque and can be used to mark when a node has been secured by a process. After process *p* removes a node from the list, *p* reads the node's `V` value (P9) and, using CAS, attempts to atomically replace the value in the node with the `secured` marker (P11), unless the value

[2] I.e, the same pointer value is used as the expected value of the hat variable in each DCAS operation.

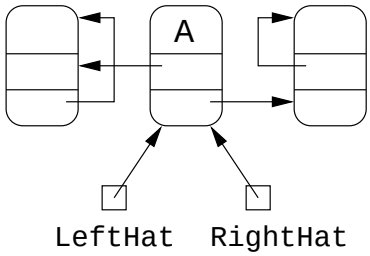
```
P1. val popRight() {
P2.  while (true) {
P3.   rh := RightHat;
P4.   rhl := rh->L;
P5.   if (rh->R = rh) {
P6.    if (RightHat = rh)
           return "empty";
P7.   } else {
P8.    if (DCAS (&RightHat,
                 &rh->L,
                 rh, rhl,
                 rhl, rh)) {
P9.     result := rh->V;
P10.    if (result != secured) {
P11.     if (CAS(&rh->V,
                 result,
                 secured)) {
P12.     rh->L := Dummy;
P13.     return result;
P14.    } else return "empty";
P15.   } else return "empty";
P16.  }
P17. }
P18.}
```

Figure 4.6: Pseudocode for the `popRight` operation.

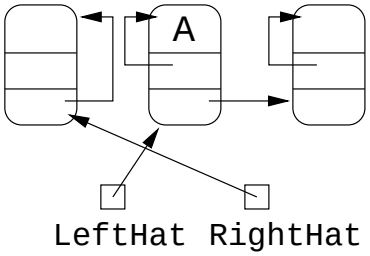
is already secured (tested at line P10). If the CAS is successful, *p* returns the value it read at line P9 (P13). If the `V` field already contains `secured` or if the CAS at line P12 fails, then some other process has already secured the node, and *p* returns `empty`. Because `secured` is a special value that is never pushed, only one process can succeed in its CAS on a given node (until the node is reclaimed), so the successful process can safely return the value in the node.

It may seem strange that a process returns `empty` when it finds some other process has secured the value of the node it removed from the list. However, it can be shown that if two processes remove the same node, then the deque is empty when the second successful DCAS is executed and that this DCAS occurs during both operations. Thus, failing processes can return `empty` without having to retry their entire operation, thereby avoiding the contention that would be caused by a retry.

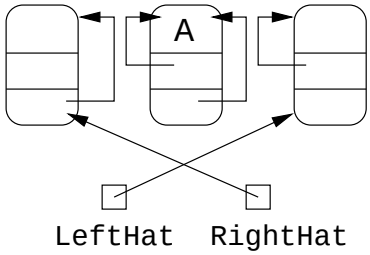
We now describe how it is possible for two processes to remove the same node from the doubly-linked list. This can occur when the `popRight` operation of one process overlaps with a `popLeft` operation of another. Figure 4.7 illustrates a sequence of deque states where two processes remove the node marked `A` from the doubly-linked list. The following example



(a)



(b)



(c)

Figure 4.7: A sequence of states in which the node containing A is removed from the doubly-linked list twice.

illustrates how this might occur.

- Process p invokes `popRight` when the deque contains one element (as illustrated by (a) of Figure 4.7). Process p loads `rh` and `rhl` and determines that the deque is not empty (lines P3-P5 of Figure 4.6). Then p is delayed.
- Likewise, another process $q \neq p$ invokes `popLeft` and executes lines P3-P5, finding the deque nonempty.
- Process p continues with its operation, using a DCAS to remove the node pointed to by its `rh` variable from the deque (line P8). The new deque state is illustrated in Figure 4.7(b).
- Likewise, process q executes the DCAS at line P8 of the `popLeft` routine. Because

```
H1. rtype pushLeft(val v) {
H2.   nd := new Node();
H3.   nd->L := Dummy;
H4.   nd->V := v;
H5.   while (true) {
H6.     lh := LeftHat;
H7.     lhL := lh->L;
H8.     if (lhL = lh) {
H9.       nd->R := Dummy;
H10.      rh := RightHat;
H11.      if (DCAS(&LeftHat,
                   &RightHat,
                   lh, rh,
                   nd, nd))
H12.        return "ok";
H13.    } else {
H14.      nd->R := lh;
H15.      if (DCAS(&LeftHat,
                   &lh->L,
                   lh, lhL,
                   nd, nd))
H16.        return "ok";
H17.    }
H18.  }
H19.}
```

Figure 4.8: Pseudocode for the `pushLeft` operation.

`LeftHat` and `lh->R` have not changed since *q* loaded these values, the DCAS is succesful. The new deque state is illustrated in Figure 4.7(c).

Now, one of *p* or *q* is guaranteed to read the value `A` from the node and successfully execute a CAS to change the node's `V` field to `secured`. The other will return `empty`. Note that the deque was empty when *q*'s DCAS was executed. The Snark algorithm has the property that whenever a node is removed twice, the deque was nonempty at the point when the first DCAS was executed, and empty at the point when the second DCAS was executed.

This "double remove" can occur under a broad range of conditions. For example, it is not necessary for there to be only one element in the deque when either of the pop operations read the hat variable. All that is necessary is that a `popLeft` and `popRight` operation respectively read the same pointer value from `LeftHat` and `RightHat`. This can happen when several push and pop operations occur between the reads of each operation.

The Snark algorithm provides an instance of prophetic linearisation because there is no way to determine whether a pop operation will return a value or `empty` until the execution

```
P1. val popLeft() {
P2.  while (true) {
P3.   lh := LeftHat;
P4.   lhr := lh->R;
P5.   if (lh->L = lh) {
P6.    if (LeftHat = lh)
           return "empty";
P7.   } else {
P8.    if (DCAS (&LeftHat,
                 &lh->R,
                 lh, lhr,
                 lhr, lh)) {
P9.     result := lh->V;
P10.    if (result != secured) {
P11.     if (CAS(&lh->V,
                 result,
                 secured)) {
P12.     lh->L := Dummy;
P13.     return result;
P14.    } else return "empty";
P15.   } else return "empty";
P16.  }
P17. }
P18.}
```

Figure 4.9: Pseudocode for the `popLeft` operation.

of the test at line P10 or the CAS at line P11. By the time these statements are executed, any number of deque operations may have been completed since the corresponding node was removed from the doubly-linked list. Therefore, we must choose a linearisation point for each pop operation before its CAS. We defer a detailed discussion of the linearisation points of the Snark algorithm until Section 4.2.4, when we describe the linearisation points for the intermediate automaton. The Snark algorithm's linearisation points can be inferred from the intermediate automaton's linearisation points, and a step correspondence that we describe in Section 4.2.5.

4.2 Modelling the Deque

This section describes the automata, an abstract, specification automaton *AbsAut*, and the intermediate automaton to be verified, *IntAut*. The specification automaton is the canonical automaton for the deque datatype (the general construction of a canonical automaton is described in Section 2.6; Section 4.2.1 contains the definition of the deque datatype). After

describing the abstract automaton in Section 4.2.2, we describe the intermediate automaton and discuss how it relates to the Snark algorithm in Section 4.2.3.

4.2.1 The Deque Datatype

As mentioned in the introduction to this chapter, a deque is like a stack or queue in that it contains a sequence of values from some set (called here *V*), but differs in that a deque provides insert (called here *push*) and remove (called here *pop*) operations on both ends of the sequence. We capture the sequential semantics of a deque using the following specification. A deque deq is a triple $(deq.seq, deq.left, deq.right)$ where $deq.seq$ is a function from integers to *V*, and $deq.left$ and $deq.right$ are integers, satisfying the constraint that $deq.left < deq.right$. The sequence of values contained in the deque deq is the sequence in $deq.seq$ from $deq.left$ to $deq.right$, not inclusive. A deque deq is empty, written $empty(deq)$, when $deq.left = deq.right - 1$ (equivalently, when $deq.left \geq deq.right - 1$).

Hitherto, we have stipulated that the deques provide four operations: one push operation for each side, and one pop operation for each side. This is the convention followed in [DFG+00, DDG+04]. However, we define the deque datatype with only two operations: a $push$ operation and a pop operation. Each operation has a parameter which indicates the side at which the operation adds or removes a value. This notational variation removes some redundancy from the model and the verification.

The following $push$ function models the deque push operations. It takes as arguments a deque value deq, a side $s \in \{left, right\} = SIDE$ and a value $v \in V$ to be pushed. It returns the deque that is the result of pushing *v* onto the appropriate side.

$$push(deq, s, v) = \begin{cases} (deq.seq \oplus \{deq.left \mapsto v\}, & \\ \qquad deq.left - 1, deq.right) & \text{if } s = left \\ (deq.seq \oplus \{deq.right \mapsto v\}, & \\ \qquad deq.left, deq.right + 1) & \text{otherwise} \end{cases}$$

The following pop function, which returns a new deque value as well as a *response value* in $V_\perp = V \cup \{\perp\}$ (where $\perp$ is a special value not in *V*), models pop operations. As with the queue model in Chapter 3, a $\perp$ return value indicates that the deque is empty.

$$pop(deq, s) = \begin{cases} (deq, \perp) & \text{if } empty(deq) \\ ((deq.seq, deq.left + 1, deq.right), & \\ \qquad deq.seq(deq.left + 1)) & \text{if } s = left \\ ((deq.seq, deq.left, deq.right - 1), & \\ \qquad deq.seq(deq.right - 1)) & \text{otherwise} \end{cases}$$

Let v_0 be any sequence, $v_0 : \mathbb{Z} \to V$. The deque datatype (D, d_0, I, R, u) is defined as follows:

$$
\begin{aligned}
D &= (\mathbb{Z} \to V) \times \mathbb{Z} \times \mathbb{Z} \\
d_0 &= (v_0, 0, 1) \\
I &= \{push\_inv(s, v) \mid v \in V,\ s \in SIDE\} \cup \{pop\_inv(s) \mid s \in SIDE\} \\
R &= \{push\_resp\} \cup \{pop\_resp(r) \mid r \in V_\perp\} \\
u(deq, inv) &= \begin{cases}
(push(deq, s, v), push\_resp) & \text{if } inv = push\_inv(s, v) \text{ for some } s, v \\
(\pi_1(pop(deq, s)), pop\_resp(\pi_2(pop(deq, s)))) & \text{otherwise}
\end{cases}
\end{aligned}
$$

4.2.2 The Abstract Automaton

AbsAut is the canonical automaton for the deque datatype as defined in Section 4.2.1. *AbsAut* has a shared variable deq, which holds the abstract deque value. As with the abstract queue automaton of Section 3.2.1, the do steps of *AbsAut* apply the the $push$ and pop functions defined in Section 4.2.1 directly, rather than using the update function *u*. *AbsAut* has a program-counter variable pc_p for each process *p*, that records which operation (if any) *p* is currently executing. The program counter variables range over the following set.

$$
\begin{aligned}
\{push\_inv(s, v) \mid s \in SIDE, v \in V\} \cup \{pop\_resp(v) \mid v \in V\} \cup \\
\{pop\_inv(s) \mid s \in SIDE\} \cup \{idle, push\_resp\}
\end{aligned}
$$

The set of initial states of *AbsAut* is presented in Figure 4.10; and the transition relation is presented in Figure 4.11.

$$\{ab \mid empty(ab.deq) \wedge \forall p \bullet pc_p = idle\}$$

Figure 4.10: The initial states of *AbsAut*.

4.2.3 The Intermediate Automaton

In this section, we describe the intermediate automaton *IntAut*. In Section 4.2.5, we explain how *IntAut* relates to the Snark algorithm. *IntAut* uses a set KEY, whose members are called *keys*, in its representation of a deque. Rather than having a state variable that is a deque containing values, *IntAut* has a state variable $kdeq$ that is a deque containing keys. That is, $kdeq$ has the same structure and operations as the deque datatype defined in Section 4.2.1 on page 104, but the values that it contains range over KEY, rather than V. *IntAut* maintains an

$push\_inv_p(s, v)$:
pre $pc_p = idle$
eff $pc_p := push\_inv(s, v)$

$push\_resp_p$:
pre $pc_p = push\_resp$
eff $pc_p := idle$

$do\_push_p(v)$:
pre $pc_p = push\_inv(s, v)$
eff $deq := push(deq, s, v)$,
$pc_p := push\_resp$

$pop\_inv_p(s)$:
pre $pc_p = idle$
eff $pc_p := pop\_inv(s)$

$pop\_resp_p(r)$:
pre $pc_p = pop\_resp(r)$
eff $pc_p := idle$

do_pop_p :
pre $pc_p = pop\_inv(s)$
eff $deq := \pi_1(pop(deq, s))$,
$pc_p := pop\_resp(\pi_2(pop(deq, s)))$

Figure 4.11: The transition relation of *AbsAut*, for $p \in PROC$, $v \in V$, $r \in V_\perp$, and $s \in SIDE$.

$push\_inv_p(s, v)$:
pre $pc_p = idle$
eff $pc_p := push\_inv(s, v)$

$push\_resp_p$:
pre $pc_p = push\_resp$
eff $pc_p := idle$

$do\_push_p(k)$:
pre $pc_p = push\_inv(s, v) \wedge$
$k \notin used$
eff $pc_p := push\_resp$,
$used := used \cup \{k\}$,
$keyed\_val :=$
$keyed\_val \oplus \{k \mapsto v\}$,
$kdeq :=$
$push(kdeq, s, k)$

Figure 4.12: The *push* actions of the automaton *IntAut*.

association between keys and values using another state variable $keyed\_val : KEY \to V$, so that $keyed\_val$ mapped across $kdeq.seq$ is a sequence of values in V.

Section 4.2.5 describes the relationship between *IntAut* and the Snark algorithm more fully, but here we remark that the set KEY models the set of pointers of the Snark algorithm, $kdeq$ models the doubly-linked list, and $keyed\_val$ models the V field of Snark's nodes. In *IntAut*, a pop operation that returns a value (i.e., that does not find the deque empty), first ensures that some key has been removed from $kdeq$ since the invocation of the pop operation (either by removing the key itself, or by observing the removal of a key by another process). These steps model the operation reading a hat variable in the Snark algorithm, and then successfully executing a DCAS on that hat. In *IntAut*, the pop operation later *secures* the key that has been removed from $kdeq$, which gives *p* alone the right to return the value associated with the key. This models a successful execution of the CAS operation in the Snark algorithm.

$pop\_inv_p(s)$:
pre $pc_p = idle$
eff $pc_p := \mathit{pop\_inv}(s)$
$key_p := \bot$

$pop\_nonempty_p$:
pre $pc_p = \mathit{pop\_inv}(s) \wedge$
$\neg empty(kdeq) \wedge$
$\mathit{key}_p = \pi_2(\mathit{pop}(kdeq, s))$
eff $pc_p := \mathit{deciding}(s)$,
$\mathit{val\_ok} := \mathit{val\_ok} \cup \{\mathit{key}_p\}$,
$\mathit{popped} := \mathit{popped} \cup \{\mathit{key}_p\}$,
$kdeq :=$
$\pi_1(\mathit{pop}(kdeq, s))$

$contend_p$:
pre $pc_p = \mathit{pop\_inv}(s) \wedge$
$empty(kdeq) \wedge$
$\mathit{popped}(\mathit{key}_p) \wedge$
$\mathit{key}_p \neq \bot$
eff $pc_p := \mathit{deciding}(s)$

$lose\_val_p$:
pre $pc_p = \mathit{deciding}(s) \wedge$
$key_p \notin val\_ok$
eff $pc_p :=$
$\mathit{pop\_resp}(\bot)$

$pop\_resp_p(r)$:
pre $pc_p = \mathit{pop\_resp}(r)$
eff $pc_p := idle$

$observe_p$:
pre $pc_p = \mathit{pop\_inv}(s) \wedge$
$\neg empty(kdeq)$
eff $\mathit{key}_p :=$
$\pi_2(\mathit{pop}(kdeq, s))$

pop_empty_p :
pre $pc_p = \mathit{pop\_inv}(s) \wedge$
$empty(kdeq)$
eff $pc_p :=$
$\mathit{pop\_resp}(\bot)$

$secure\_val_p$:
pre $pc_p = \mathit{deciding}(s) \wedge$
$\mathit{key}_p \in val\_ok$
eff $pc_p := \mathit{pop\_resp}(\mathit{keyed\_val}(\mathit{key}_p))$,
$val\_ok := val\_ok \setminus \{\mathit{key}_p\}$

Figure 4.13: The *pop* actions of the automaton *IntAut*.

We now describe the transitions of the intermediate automaton that relate to its implementation of the *push* operation. That is, transitions labelled by the external actions $push\_inv_p(s, v)$, $push\_resp_p$ and the internal action $do\_push_p(k)$ for each process *p*, side *s*, value *v*, and key *k*. Figure 4.12 presents the transition relation for transitions labelled by these actions. Note that the preconditions and effects associated with the $push\_inv$ and pop_inv actions are precisely the same as with *AbsAut*. The do_push transitions are more complicated. Note that the do_push transitions have an extra argument $k \in KEY$. The precondition of the do_push transition asserts that this key must not have been already used as an argument to a do_push action. This is expressed using a state variable $used \subseteq KEY$. When a key is used as an argument to a do_push action, it is added to $used$, and no key is ever removed from this set. The new key is associated with the value being pushed, via the $keyed\_val$ function, and the key is added to the appropriate side of $kdeq$. Finally, a $do\_push_p(k)$ transition sets the program counter of *p* to $push\_resp$, so that the next action *p* executes will be $push\_resp_p$.

We now describe the transitions of the intermediate automaton that relate to its implementation of the *pop* operation. As with *AbsAut*, actions of the form $pop\_inv_p(s)$ and $pop\_resp_p(r)$ represent respectively the invocations and responses of pop operations. *IntAut* also has the following internal actions, for each process *p*:

- $observe_p$, during which the process *p observes* a key at one end of $kdeq$. Later, *p* may remove this key from $kdeq$. There may be several $observe_p$ actions in each operation.
- $pop\_nonempty_p$, during which *p* removes from $kdeq$ the key that it last observed. This action may only occur when $kdeq$ is nonempty.
- pop_empty_p, after which *p* is guaranteed to return $\perp$ from the pop operation. This action may only occur when $kdeq$ is empty.
- Three further actions $contend_p$, $secure\_val_p$ and $lose\_val_p$. These actions are explained below, but they are used to regulate the steps by which a process observes that a key has been removed from $kdeq$, and then succeeds or fails in securing the right to return that key.

Figure 4.13 presents the transition relation for these actions.

Figure 4.14 presents a state diagram that illustrates the structure of an execution of the pop operation in *IntAut* by a process *p*. The identifiers in the boxes are program-counter values, and the labels on the arrows are actions. The annotations in angle brackets indicate preconditions. More precisely, an arrow from a box containing program-counter value *c*, to a box containing program-counter value *c′*, labelled with action a_p and annotation *S* means that *IntAut* has a transition, labelled by a_p with a precondition implying that in the prestate, $pc_p = c$ and *S* both hold, and an effect implying that in the poststate $pc_p = c'$. For the sake of clarity, the other effects of the transitions are not depicted.

The transitions of *IntAut* labelled by external actions are similar to those of *AbsAut*. A process *p* must be idle to take a pop_inv_p transition, and the program counter of *p* is set

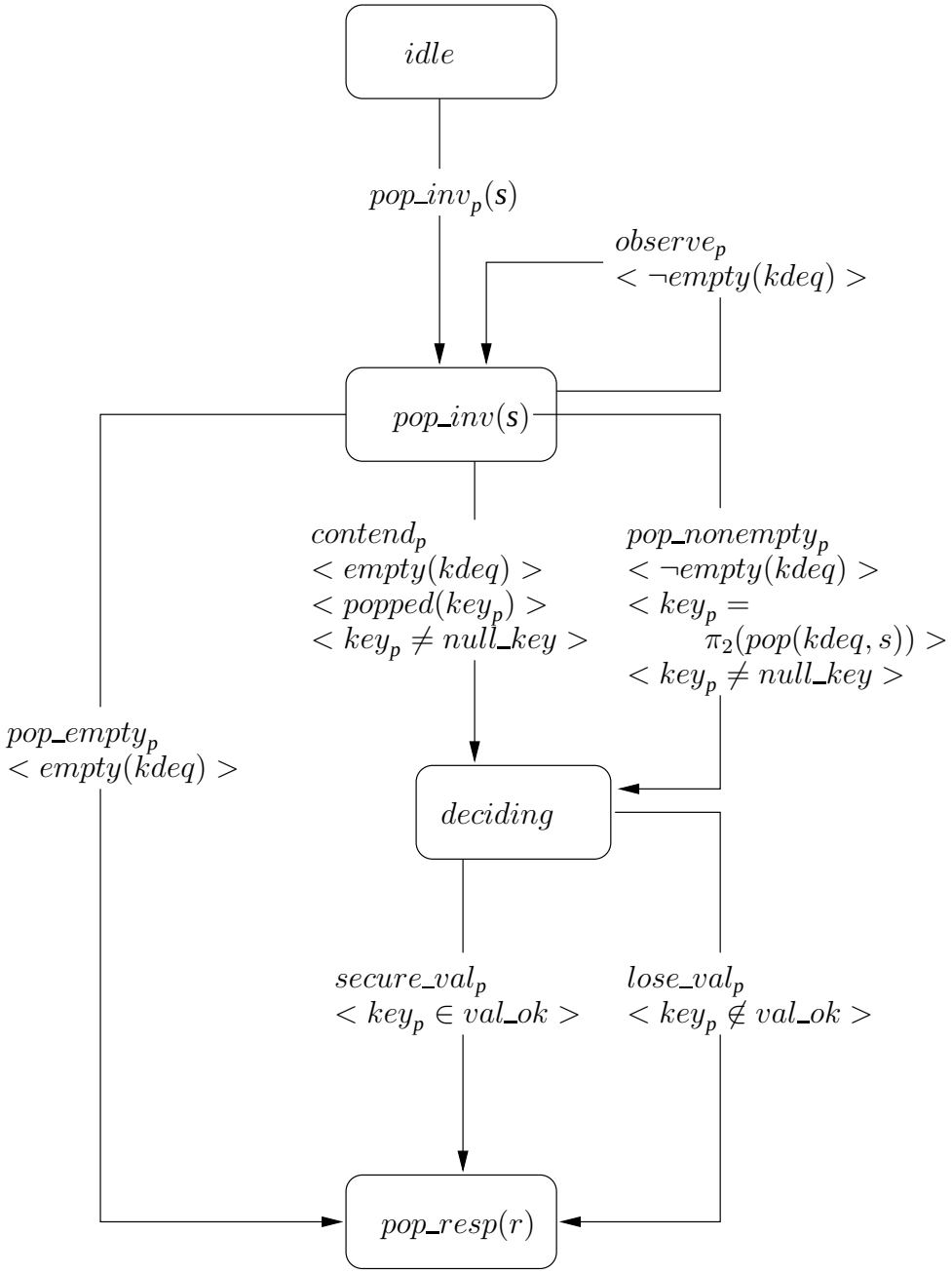


Figure 4.14: State diagram for the *pop* operation of *IntAut*.

to $pop\_inv(s)$, indicating that *p* has begun a pop operation on side *s*. A process *p* takes a $pop\_resp_p(r)$ transition when it has found some response value $r \in V_\perp$ to return from the pop operation, indicated by a program counter value of $pop\_resp(r)$. Afterwards, *p* becomes idle again.

While $pc_p = pop\_inv(s)$ (which holds just after the invocation of a pop operation), *p* can take one or more $observe_p$ actions, which record in another state variable $key_p : KEY_\perp$ the key at the end of $kdeq$ from which *p* is popping.[3] $kdeq$ must be nonempty during $observe_p$ actions, so that this key is guaranteed to exist. If $kdeq$ is empty, *p* may take a pop_empty_p action, which sets pc_p to $pop\_resp(\perp)$. Afterwards, during a pop_resp_p action, *p* returns an indication that it found the deque empty.

Once a process *p* has observed a key, *p* may take one of two further internal actions: $pop\_nonempty_p$ or $contend_p$. During a $pop\_nonempty_p$ action the key that *p* most recently observed is removed from $kdeq$, and *p*'s program counter is set to $deciding(s)$, which indicates that *p* is attempting to secure key_p while executing a pop operation on side *s*. *IntAut* has a state variable $val\_ok \subseteq KEY$ that records which keys have been popped but not yet secured by any process (the "value" of the key is "ok" because it can still be secured and returned by some process). The key that *p* observed is added to the set val_ok, indicating that the key can be secured. *IntAut* has a further state variable $popped \subseteq KEY$, which records the set of keys that have been removed from the deque during any $pop\_nonempty$ action. Accordingly, the key that *p* observed is added to this set.

By taking the $contend_p$ action, *p* becomes eligible to secure the key that it last observed, assuming that key has not yet been secured. The precondition of $contend_p$ implies that key_p has already been popped from $kdeq$ and that $kdeq$ is empty, and its effect simply sets *p*'s program counter to $deciding(s)$.

Once a process *p*'s program counter has become $deciding$ (via either $pop\_nonempty_p$ or $contend_p$), *p* takes either a $secure\_val_p$ action, or a $lose\_val_p$ action. The precondition of the $secure\_val_p$ action implies that $key_p \in val\_ok$, so no other process has yet secured the value. The program counter of *p* is set to $pop\_resp(keyed\_val(key_p))$, indicating that *p* will return the value associated with key_p. key_p is removed from the set val_ok, indicating that no other process can secure this key.

The process *p* takes the $lose\_val_p$ action if some other process secures the key last observed by *p*. Accordingly, the precondition of $lose\_val_p$ transitions implies that key_p is not in val_ok. The program counter of *p* is set to $pop\_resp(\perp)$, so that *p*'s next action will be a response indicating that *p* found the deque empty.

A state *is* is an initial state of *IntAut* if and only if the following conditions hold.

- $empty(is.kdeq)$,
- $is.used = is.popped = \varnothing$, so that no key is $used$ or $popped$,
- $is.val\_ok = \varnothing$ so that no key can be secured,

[3] Note that in Figure 4.13, the invocation of the *pop* function on $kdeq$ does not update the value of $kdeq$. There is no assignment to the $kdeq$ variable.

- For all processes *p*, $is.pc_p = idle$ and $is.key_p = \perp$, so that no operations are underway, and no key has been observed.

4.2.4 Linearisation Points of the Intermediate Automaton

We now describe linearisation points for operations of the intermediate automaton. The linearisation point for a push operation is straightforward: the $do\_push_p(k)$ step of each push operation is the linearisation point for the operation. This choice is forced on us, because after a $do\_push_p(k)$ transition, the value pushed may be immediately popped and returned by another process. Therefore, the value must be visible to other processes.

Pop operations that return $\perp$ after taking a pop_empty_p action are linearised at the pop_empty_p action. This is because $kdeq$ is empty at this point, and thus there are no values that can be returned. Finding linearisation points for other pop operations is much more difficult. Consider some process *p* that executes a pop operation in which the following actions occur (assuming $v \neq \perp$):

$$pop\_inv_p(s),\ observe_p,\ contend_p,\ secure\_val_p,\ pop\_resp_p(v)$$

Because *p* took a $contend_p$ action, we can show that there is some other process $q \neq p$ such that *q* executed a $pop\_nonempty_p$ action between the $observe_p$ and $contend_p$ actions, and such that $key_p = key_q$. When the $pop\_nonempty_q$ action took place, $kdeq$ was nonempty. It may seem tempting to choose this $pop\_nonempty$ action as the linearisation point of *p*'s operation. However, there is no guarantee that *q* is executing a pop operation on the same side as *p*'s operation, and thus no guarantee that the value *v* which *p* ultimately returns appears on the appropriate side of the deque, at that point. (The value *v* may have been added to the deque immediately prior to *q*'s $pop\_nonempty$ action, and at a point when the deque was nonempty.)

The only action at which the value *v* finally returned by *p* is guaranteed to be at the appropriate side of the deque is $observe_p$. For this reason, we choose *p*'s most recent $observe_p$ action as the linearisation point of *p*'s pop operation. Note that the $observe_p$ action does not modify $kdeq$. Therefore, if we are to make this counter-intuitive scheme work, we need to account for the fact that the sequence of values in the abstract deque value represented at any state of *IntAut* is *not* the sequence of values obtained by mapping $keyed\_val$ across $kdeq$. The backward simulation presented in Section 4.3 defines the relationship between the two sequences precisely.

To be consistent with the linearisation points of operations that take a $contend$ action, we linearise each operation that takes a $pop\_nonempty$ action, and later returns a value $v \neq \perp$, at the most recent $observe$ action of that operation. We now need to find linearisation points for pop operations that return $\perp$, without taking a pop_empty action. These are the operations that take a $lose\_val$ action during their execution. We want to find a point between the invocation and response of each such operation at which $kdeq$ is empty. For each process *p*, the precondition of $lose\_val_p$ asserts that $\neg val\_ok(key_p)$. Thus, there must be some other process $q \neq p$ that executed the $secure\_val_q$ action at some point prior to the $lose\_val_p$

action, with $key_p = key_q$. Therefore, either *p* or *q* executed a $contend$ action during its operation. The precondition of this $contend$ action asserts that $kdeq$ is empty. Therefore, we linearise each operation by a process *p* that takes a $lose\_val_p$ action during its execution, at the prior $contend_{p'}$ that took place when $key_p = key_{p'}$. Note that in general, several processes may be linearised at this $contend$ action, because several processes may take a $lose\_val$ action with the same key.

There are three things to note about the scheme of linearisation points outlined above.

- The question of whether an action is the linearisation point of an operation can only be answered by considering events occurring later in the execution.
- An action of one process can be the linearisation point of another process.
- The linearisation point of pop operations that do not return $\perp$ has no effect on the shared data structure of *IntAut*, but does have an effect on the abstract deque being represented.

These three properties make the verification challenging.

4.2.5 Snark Implements *IntAut*

We briefly describe the relationship between the Snark algorithm and *IntAut*. A forward simulation exists between Snark and *IntAut*, which we do not discuss in detail in this thesis. However, we give a brief overview of the simulation, paying particular attention to the step correspondence. The actual verification involves an automaton whose transitions model the steps of the Snark algorithm in the same way as the concrete automaton of Chapter 3 models the M&S queue. However, during this discussion, we speak of the steps of the Snark algorithm as though they were actions of an automaton that models the Snark algorithm.

The keys of *IntAut* model pointers in the Snark algorithm, and $kdeq$ of *IntAut* which contains a sequence of keys $kdeq.seq$, models the doubly-linked list of Snark (not including the sentinels). We identify the set of pointers of Snark with the set of keys of *IntAut*. The forward simulation asserts that, for each pointer `nd` appearing in the doubly-linked list, `nd->V` is the value $keyed\_val(\texttt{nd})$ in *IntAut*, and that the order in which pointers occur in the doubly-linked list of Snark is the same as the order in which pointers occur in $kdeq.seq$. These properties imply that the sequence of values contained in the doubly-linked list is the same as the sequence of values obtained by mapping $keyed\_val$ across $kdeq.seq$. In particular, if the doubly-linked list is empty in some state, then $kdeq$ is guaranteed to be empty in related states of *IntAut*.

The step correspondence used in the forward simulation associates actions representing the successful execution of a DCAS in a push operation with the $do\_push_p(k)$ action of the executing process, where *k* is the new node added onto the doubly-linked list. Note that in transitions labelled by $do\_push_p(k)$, *k* has not yet been added to $kdeq$, and the value being

pushed is associated with *k*. These two properties of the transition model the Snark algorithm's allocation and initialisation of the new node. All other internal actions that represent steps of the Snark algorithm that are taken during push operations are stutter steps.

When the doubly-linked list is nonempty, the step correspondence associates actions that represent reading a hat variable during a pop operation with the $observe_p$ action of the executing process. The relationship between the doubly-linked list and $kdeq$ ensures that the pointer read from the hat variable is the same pointer observed in *IntAut*. If the doubly-linked list is empty, the step correspondence associates the read of the hat with the pop_empty_p action.

The step correspondence associates the successful execution of the DCAS at line P8 with the $pop\_nonempty_p$ action if the doubly-linked list is nonempty. Such a DCAS removes a node from the doubly-linked list, and the simulation relation guarantees that the pointer to that node is removed from $kdeq$. If the doubly-linked list is empty, the step correspondence associates a successful DCAS with the $contend_p$ action. It is possible to show that, if the DCAS at line P8 can be executed successfully and the doubly-linked list is empty then the value of the hat that is one of the targets of the DCAS is $popped$ in related states of *IntAut*, and so the precondition of $contend_p$ is satisfied.

The simulation relation asserts that each pointer whose `val` field is not `secured` in a state of the Snark algorithm, is not $secured$ in any related state of *IntAut*. The step correspondence associates the successful execution of the CAS at line P11 with $secure\_val_p$. Steps of the Snark algorithm during which a process discovers that a node has already been secured (either the failure of the test at line P10, or the unsuccessful execution of the CAS at line P11) are associated with $lose\_val_p$.

As usual, the step correspondence associates each invocation or response of the Snark algorithm with the same invocation or response of *IntAut*.

4.3 The Backward Simulation

We now describe the backward simulation used in our proof that *IntAut* implements *AbsAut*. Figure 4.15 presents the definition of backward simulation, taken from Definition 2.18 on page 38, applied to the automata *IntAut* and *AbsAut*. As discussed in Chapter2, the existence of such a relation allows us to inductively construct, for any (finite) execution of *IntAut*, an execution of *AbsAut* with the same trace, and thus guarantees that *IntAut* implements *AbsAut*.

The simulation relation *R* that we use in this verification is presented in Figure 4.16. We describe the motivation behind the relation *R*, and discuss the highlights of the proof.

SeqOk and WinnerUnique $SeqOk$ describes the relationship between the abstract deque and *IntAut*'s key sequence. We first consider a simple assertion that fails to adequately describe this relationship. The variables $kdeq$ and $keyed\_val$ of the intermediate automaton together yield a sequence of values, thus:

$$\sigma(is) = \lambda\, i \bullet is.keyed\_val(is.kdeq.seq(i))$$

$$(\forall\, is_0 \bullet (\exists\, as \bullet R(is, as))) \tag{4.1}$$

$$\begin{array}{l}(\forall\, is, is', as', a \bullet \\ \quad R(is', as') \wedge is \xrightarrow{a} is' \Rightarrow \\ \quad\quad (\exists\, as, \beta \bullet \\ \quad\quad\quad R(is, as) \wedge as \stackrel{\beta}{\Longrightarrow} as' \wedge \\ \quad\quad\quad trace(\beta) = trace(\langle a \rangle))\end{array} \tag{4.2}$$

$$\begin{array}{l}(\forall\, is : start(IntAut), as \bullet R(is, as) \Rightarrow \\ \quad\quad as \in start(AbsAut))\end{array} \tag{4.3}$$

Figure 4.15: A relation $R \subseteq states(IntAut) \times states(IntAut)$ is a *backward simulation* from *IntAut* to *AbsAut* if these conditions hold, where $is, is' : states(IntAut)$; $as, as' : states(AbsAut)$; $a : acts(IntAut)$; $\beta : acts(AbsAut)^*$

$$\begin{array}{l}R(is, as) \mathrel{\hat{=}} \\ \quad CorrespondenceOk(as, is) \wedge \\ \quad WinnerUnique(as, is) \wedge \\ \quad (\exists\, m \bullet SeqOk(as, is, m))\end{array}$$

Figure 4.16: The simulation relation R.

It might seem tempting to build a simulation relation around a simple relationship between this sequence and the deq variable of the abstract automaton, i.e.,

$$\begin{array}{c}as.deq.seq = \sigma(is) \wedge is.left = as.kdeq.left \wedge \\ is.right = as.kdeq.right\end{array} \tag{4.4}$$

However, the linearisation points of the intermediate automaton preclude this. We need a weaker property that allows the key sequence in the intermediate automaton to contain values that have been removed from the abstract deque, so that we can choose do_pop_p for transitions labelled by $observe_p$ (at least, when p is executing an operation that returns a value). The predicate $SeqOk$, presented in Figure 4.17, defines such a property. $SeqOk$ describes states of *AbsAut* and *IntAut* and a *match* function $m : \mathbb{Z} \to \mathbb{Z}$, that associates indexes between the limits of the abstract deque with indexes between the limits of the intermediate automaton (Clause 4.5). For any i between the limits $as.kdeq.left$ and $as.kdeq.right$, this function

$$
\begin{array}{l}
SeqOk(as, is, m) \mathrel{\hat{=}} \\
\quad (\forall\, i \bullet as.deq.left < i < as.deq.right \Rightarrow \\
\qquad is.left < m(i) < is.right \wedge & (4.5) \\
\qquad \sigma(is)(m(i)) = as.deq.seq(i) \wedge & (4.6) \\
\qquad \neg WinnerExists(as, is, is.kdeq.seq(m(i)))) & (4.7) \\
\quad \wedge \\
\quad (\forall\, i, j \bullet as.deq.left < i < j < as.deq.right \Rightarrow \\
\qquad m(i) < m(j)) & (4.8) \\
\quad \wedge \\
\quad (\forall\, i \bullet is.left < i < is.right \Rightarrow \\
\qquad InMatchRange(as, m, i) \vee & (4.9) \\
\qquad WinnerExists(as, is, is.keys(i))) & (4.10)
\end{array}
$$

$$
\begin{array}{l}
WinnerUnique(as, is) \mathrel{\hat{=}} \\
\quad \forall\, p, q \bullet as.pc_p = pop\_resp(v_1) \wedge \\
\qquad as.pc_q = pop\_resp(v_2) \wedge is.key_p = is.key_q \wedge \\
\qquad\quad v_1 \neq \bot \wedge v_2 \neq \bot \Rightarrow p = q & (4.11)
\end{array}
$$

Figure 4.17: The $SeqOk$ and $WinnerUnique$ predicates.

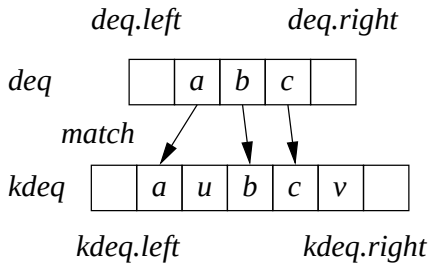


Figure 4.18: The $match$ function.

$$
\begin{aligned}
&WinnerExists(as, is, k) \mathrel{\hat{=}} \\
&\quad \exists\, p, v \bullet as.pc_p = pop\_resp(v) \wedge \\
&\qquad v \neq \bot \wedge is.key_p = k \\
&InMatchRange(as, m, i) \mathrel{\hat{=}} \\
&\quad \exists\, j \bullet as.deq.left < j < as.deq.right \wedge \\
&\qquad m(j) = i \\
&OtherDeciderExists(is, p) \mathrel{\hat{=}} \\
&\quad \exists\, q \bullet q \neq p \wedge is.key_p = is.key_q \wedge \\
&\qquad is.pc_q = deciding(s)
\end{aligned}
$$

Figure 4.19: Auxilliary predicates.

satisfies:

$$\sigma(is)(m(i)) = as.deq.seq(i)$$

Thus *m* takes each abstract index to an intermediate index that is associated with the same value. This is illustrated in Figure 4.18.

Not all indexes of σ between $is.left$ and $is.right$ are in the range of the function *m*. Each index between these limits not in the range of *m* is associated by $kdeq.seq$ with a key *k* that has already been observed by some process *p* during an $observe_p$ transition that is the linearisation point for *p*'s operation. When this has occurred, we say that *p* has *won* the key *k*. For the indexes that are in the range of *m*, no process has won the associated key (Clause 4.7). For a key *k*, $WinnerExists$ formalises the idea that some process *p* has observed *k* at the linearisation point of *p*'s pop operation (Figure 4.19). This formalisation is achieved by asserting that $is.pc_p = k$ and $as.pc_p = pop\_resp(v)$, where *v* is the value associate with *k*.

Clause 4.8 asserts that *m* preserves the order of its domain and is injective. The final

conjunct of $SeqOk$ constrains the properties of indexes in the intermediate automaton. Each index between *is*.$left$ and *is*.$right$ is either in the range of *m* (Figure 4.19), or is associated with a key that has already been won.

$WinnerUnique$ (Figure 4.17) asserts that at most one process has won each key, which enables us to prove that $SeqOk$ is preserved during intermediate transitions for which a do_pop abstract transition is chosen when the abstract deque is nonempty.

One consequence of the $SeqOk$ predicate is that when $left = right - 1$, the abstract deque is empty. This is because, in order for *m* to be injective, the set [*is*.$kdeq$.*left*..*is*.$kdeq$.*right*] must have at least as many elements as the set [*as*.$kdeq$.*left*..*as*.$kdeq$.*right*]. This is important in the coming discussion.

CorrespondenceOk. $CorrespondenceOk$ asserts that, in related abstract and intermediate states, every process satisfies one of six (mutually exclusive) predicates. Each predicate constrains the values of local (that is, *p*-indexed) variables in the given intermediate and abstract states. The specific predicate that a given process satisfies changes during the execution of each operation. Figure 4.20 presents the $CorrespondenceOk$ predicate, and its simpler subpredicates. The remaining subpredicates are presented later in the discussion.

To explain $CorrespondenceOk$, we describe the actions that each process may take during the execution of an operation, and show which disjunct each process satisfies at each point in its operation. During this discussion, we define the step correspondence used in the verification. This step correspondence is essentially the scheme of linearisation points described in Section 4.2.4. Because we are dealing with a backwards simulation, we will traverse *backwards* through the actions of each operation.

Fix a transition *is* $\xrightarrow{a}$ *is*′, where action *a* is indexed by process *p*. In addition, fix an abstract state *as*′ such that $R(as', is')$. We discuss each of the possible values of *a* in turn. During this discussion, we repeatedly claim that given the abstract and intermediate program-counter values of a process, only one of the disjuncts of $CorrespondenceOk$ can be satisfied. It is easy to convince yourself of claims like this by inspecting the various definitions.

We begin with the push operations. At different points in the execution of a push operation, a process *p* satisfies $IdleOk$ and $PushOk$. These predicates are presented in Figure 4.20. $IdleOk$ asserts that a process *p* is not executing any operation in the abstract or intermediate state, and $PushOk$ asserts that *p* is executing a push operation in both the abstract and intermediate states and that *p* is "at the same stage" in its operation. Assume that $a = push\_resp_p$. Then *is*.$pc_p = push\_Resp$ and *is*′, $pc_p = idle$. Because of this, *p* must satisfy $IdleOk(as', is', p)$, which is the only disjunct of $CorrespondenceOk$ that allows *p* to have the $idle$ program counter. Let *as* be the abstract state that is the same as *as*′ at every variable, except that *as*.$pc_p = push\_Resp$. Then $PushOk(as, is)$. Note that *as* $\xrightarrow{a}$ *as*′, so the step correspondence can associate each action of the form $push\_resp_p$ with the same action and obtain a transition of the abstract automaton (as required by the conditions for *R* to be a backward simulation relation).

None of the disjuncts of $CorrespondenceOk$ except $PushOk$ can be true for any pro-

$$
\begin{aligned}
&CorrespondenceOk(as, is) \triangleq \\
&\quad \forall\, p \bullet IdleOk(as, is, p) \vee \\
&\qquad PushOk(as, is, p) \vee \\
&\qquad FinishedPopOk(as, is, p) \vee \\
&\qquad LosingPopOk(as, is, p) \vee \\
&\qquad WinningPopOk(as, is, p) \vee \\
&\qquad StartingPopOk(as, is, p) \\
&IdleOk(as, is, p) \triangleq \\
&\qquad as.pc_p = is.pc_p = idle \\
&PushOk(as, is, p) \triangleq \\
&\qquad as.pc_p = is.pc_p = push\_inv(s, v) \vee \\
&\qquad as.pc_p = is.pc_p = push\_Resp \\
&FinishedPopOk(as, is, p) \triangleq \\
&\qquad as.pc_p = is.pc_p = pop\_resp(r) \\
&StartingPopOk(as, is, p) \triangleq \\
&\qquad as.pc_p = is.pc_p = pop\_inv(s)
\end{aligned}
$$

Figure 4.20: $CorrespondenceOk$, and subpredicates.

cess p where $pc_p = push\_Resp$, so if $a = do\_push_p(k)$ then $PushOk(as', is', p)$. Let as be the abstract state that is the same as as', except that $as.pc_p = is.pc_p = push\_inv(s, v)$ for some side s and $v \in V$, and that deq is modified so that $as'.deq = push(as.deq, s, v)$. Then $PushOk(as, is, p)$, and $as \xrightarrow{do\_push(s,v)} as'$.

A similar line of reasoning can be applied when $a = push\_inv_p(s, v)$. In this case, $is.pc_p = idle$ and $is'.pc_p = push\_inv_p(s, v)$ for some side s and value v, and so $PushOk(as', is', p)$. Therefore, $as'.pc_p = push\_inv(s, v)$. Let as be the state like as' except that $as.pc_p = idle$. Then $IdleOk(as, is, p)$ and $as \xrightarrow{a} as'$.

This covers the actions that may be taken during push operations. We turn now to the actions that occur during pop operations, which are more complicated. During different intervals in every pop operation, each process p satisfies $FinishedPopOk$ and $StartingPopOk$, which are presented in Figure 4.20. $FinishedPopOk$ asserts that a process p has completed its pop operation in both abstract and intermediate states and is waiting to return. $StartingPopOk$ asserts that a process p has just begun its pop operation in both abstract and intermediate states. Between intervals in which p satisfies $FinishedPopOk$ and

$$
\begin{aligned}
&WinningPopOk(as, is, p) \triangleq \\
&\quad WinningDeciding(as, is, p) \vee \\
&\quad WinningPopping(as, is, p) \\
&WinningDeciding(as, is, p) \triangleq \\
&\quad is.key_p \neq \bot \wedge && (4.12)\\
&\quad as.pc_p = pop\_resp(is.keyed\_val(is.key_p)) \wedge && (4.13)\\
&\quad is.pc_p = deciding(s) \wedge && (4.14)\\
&\quad is.key_p \in is.popped \wedge && (4.15)\\
&\quad is.key_p \in is.val\_ok && (4.16)\\
&WinningPopping(as, is, p) \triangleq \\
&\quad is.key_p \neq \bot \wedge is.pc_p = pop\_inv(s) \wedge && (4.17)\\
&\quad as.pc_p = pop\_resp(is.keyed\_val(is.key_p)) \wedge && (4.18)\\
&\quad (is.key_p \notin is.popped \vee is.key_p \in is.val\_ok) && (4.19)
\end{aligned}
$$

Figure 4.21: The $WinningPopOk$ predicate, and subpredicates.

$StartingPopOk$, p satisfies either $WinningPopOk$ or $LosingPopOk$, which are presented in Figures 4.21 and 4.22, respectively. Process p satisfies $WinningPopOk$ during a pop operation if and only if p returns a value (not $\bot$) from the operation. On the other hand, if process p satisfies $LosingPopOk$ during execution of a pop operation, then that operation returns $\bot$. (Process p may also return $\bot$ by taking the pop_empty_p action during the operation.)

A process p satisfies $WinningDeciding$ during the interval after p has taken a $pop\_nonempty$ or $contend$ action, but before executing $secure\_val$. p satisfies $WinningPopping$ during the interval between p's last $onserve$ action and its $pop\_nonempty$ or $contend$ action. p's operation is linearised at the beginning of this interval.

A process p satisfies $LosingPostLin$ during the interval beginning with the linearisation point of p's operation (a $contend$ action of some process), and ending when p takes the $lose\_val_p$ action. A process p satisfies $LosingPreLin$ during the interval beginning with the $pop\_nonempty_p$ or $contend_p$ action. If p takes a $contend_p$ action, then p will not satisfy $LosingPreLin$ during that operation. This is because the $contend_p$ action is the linearisation point of p's operation. p will only satisfy $LosingPreLin$ if it takes the pop_empty_p action, and later takes the $lose\_val_p$ action.

As before, we fix a transition $is \xrightarrow{a} is'$, and fix an abstract state as' such that $R(as', is')$. Assume first that $a = push\_resp_p(r)$ for some p and $r \in V_\bot$. As before $is'.pc_p = idle$ and thus $IdleOk(as', is', p)$. Further, $is.pc_p = push\_Resp(r)$. Let as be the state like as' except

$$
\begin{aligned}
&LosingPopOk(as, is, p) \triangleq \\
&\quad LosingPostLin(as, is, p) \vee \\
&\quad LosingPreLin(as, is, p) \vee \\
&LosingPostLin(as, is, p) \triangleq \\
&\quad as.pc_p = pop\_resp(\bot) \wedge && (4.20)\\
&\quad is.pc_p = deciding(s) \wedge && (4.21)\\
&\quad is.key_p \neq \bot \wedge \\
&\quad (is.key_p \in is.val\_ok \Rightarrow \\
&\qquad\quad WinnerExists(as, is, k) \wedge && (4.22)\\
&\qquad\quad OtherDeciderExists(is, p)) \\
&LosingPreLin(as, is, p) \triangleq \\
&\quad as.pc_p = pop\_inv(s) \wedge && (4.23)\\
&\quad is.pc_p = deciding(s) \wedge is.key_p \neq \bot \wedge && (4.24)\\
&\quad (is.key_p \in is.val\_ok \Rightarrow \\
&\qquad\quad WinnerExists(as, is, k)) && (4.25)
\end{aligned}
$$

Figure 4.22: The $LosingPopOk$ predicate, and subpredicates.

that $as.pc_p = push\_Resp(r)$. Thus, $FinishedPopOk(as, is, p)$, and $as \xrightarrow{a} as'$.

Now, assume $a = lose\_val_p$. In this case, $is.pc_p = deciding(s)$ for some side s and $is.key_p \notin isval\_ok$. Also, $is'.pc_p = pop\_resp(\bot)$ so that $FinishedPopOk(as', is', p)$ (since this is the only disjunct of $CorrespondenceOk$ that allows p to have the program counter value $pop\_resp(\bot)$ in the intermediate automaton). The $lose\_val_p$ actions are not linearisation points, and thus are stutter steps in our step correspondence. We show that $LosingPopOk(as', is, p)$ by showing that $LosingPostLin(as', is, p)$. $LosingPostLin$ asserts that p's pop operation is linearised in the abstract state. That is, $pc_p = pop\_resp(\bot)$. We already know that $as'.pc_p = pop\_resp(\bot)$ (by $FinishedPopOk$ and the intermediate transition relation) and $is.pc_p = deciding(s)$. Note that because $is.pc_p = deciding(s)$, p must already have observed a key, and thus $is.key_p \neq \bot$ (this is an invariant of *IntAut*). Also, the precondition of the transition implies that $is.key_p \notin isval\_ok$.

Note that if $is.val\_ok(is.key_p)$, $LosingPostLin$ implies the a winner exists, for key_p, and there is some other process $q \neq p$ with $is.key_p = is.key_q$ and $is.pc_p = deciding(s)$ ($OtherDeciderExists$ is defined in Figure 4.19). We describe why this is so shortly.

$LosingPostLin$ is true for some p in the poststate of some transition, but false in the prestate under two conditions: if $a = contend_p$, or if $a = contend_q$ for some process $q \neq p$ such that $is'.key_p = is'.key_q$ and q is the winner for $is'.key_p$. In either case, the $contend$ action is the linearisation point of p's operation. Let as be the abstract state like as' except $as.pc_p = pop\_inv(s)$ (where s is the side that p is popping from, and satisfies $is'.pc_p = deciding(s)$). Note that because $is.kdeq = is'.kdeq$ is empty, $as.deq = as'.deq$ is also empty. Thus $as \xrightarrow{do\_pop_p} as'$. Given this, if $a = contend_p$ p satisfies $StartingPopOk$ in the prestate. Otherwise, it can be shown that if $a = contend_q$ for some process $q \neq p$ satisfying $is'.key_p = is'key_q$, then p satisfies $LosingPreLin$ in the prestate.

Consider the case where $a = pop\_nonempty_p$ for some p, where $as'.pc_p = pop\_resp(\bot)$. In this case, it must be that p satisfies $LosingPopOk$. Moreover, the assertion that $OtherDeciderExists$ within $LosingPostLin$ enables us to prove that when a process p satisfies $LosingPreLin$ in the poststate. (This is a consequence of the fact that $is'.key_p$ has only one "decider" immediately after it is removed from $kdeq$.) This is important, because the $pop\_nonempty_p$ action is *not* a linearisation point for p, and the step correspondence does not associate this intermediate action with an abstract action. However, the fact that p satisfies $LosingPreLin$ in the poststate enables us to prove that p satisfies $StartingPopOk$ in the prestate.

We turn now to pop operations that return a value $v \in V$. Such operations execute a sequence of internal actions of the following form:

$$observe_p, X, secure\_val_p$$

where X is either $pop\_nonempty_p$ or $contend_p$. In either case, it can be show that p satisfies $WinningDeciding$ between the X action, and the $secure\_val_p$ action, and satisfies $WinningPopping$ between the $observe_p$ action (which is p's last $observe$ action during the operation) and the X action. The $secure\_val_p$ action is a stutter step, and the $observe_p$ action

is associated by the step correspondence with the do_pop_p action.

Finally, if $a = pop\_empty_p$, we can show that p satisfies $FinishedPopOk$ in the post-state, and satisfies $StartingPopOk$ in the prestate. The step correspondence associates such actions with the do_pop abstract action, and $SeqOk$ together with the transition relation of *IntAut* guarantee that $as.deq = as'.deq$ is empty.

4.4 Verifying the Simulation

There are three components to the proof that R is a backward simulation. The initial state condition, the totality condition, and the transfer condition.

For the initial state condition, we must prove that, for all is and as such that $R(is, as)$, $as \in start_{AbsAut}$. This amounts to showing that $as.deq$ is empty and that for all p, $as.pc_p = idle$. The predicate $SeqOk$ guarantees that if $is.kdeq$ is empty (as is the case initially), then $as.deq$ is empty. Furthermore, for all p, when $is.pc_p = idle$, p must satisfy $IdleOk$. Therefore, $as.pc_p = idle$. Furthermore,

For the totality condition, given an intermediate state is, we must be able to construct an abstract state ws such that $R(is, ws)$. It is possible to do this in such a way that the simple relationship defined by 4.4 on page 114 holds between is and ws. This is achieved by letting $ws.deq.seq = \sigma(is)$, and letting the limits of $is.kdeq$ match the limits of $ws.deq$ thus: We define each $ws.pc_p$ so that no process that is still popping has yet executed its do_pop in the abstract automaton. Except when $is.pc_p = deciding$, we set $ws.pc_p = is.pc_p$.

In order to satisfy the relation R, we need to construct ws so that each process with $is.pc_p = deciding$ is either $WinningPopOk$ or $LosingPopOk$ (those are the only disjuncts of $CorrespondenceOk$ in which $pc_p = deciding$ is possible). Because satisfaction of $LosingPopOk$ by a process p implies that if $key_p \in val\_ok$ then there is some process that has won the key (the content of the $WinnerExists$ predicate), we need to choose some winner for each key k, such that there is a process p with $is.key_p = k$ and $is.pc_p = deciding$. We do this using a a *choice* function $winner : KEY \to PROC$ satisfying

$$(\exists\, p \bullet is.key_p = k \wedge is.pc_p = deciding) \Rightarrow winner(k) \in \{p \mid key_p \wedge is.pc_p = deciding\}$$

If $is.pc_p = deciding$, $is.key_p \in is.val\_ok$ and $p = winner(is.key_p)$ we set $as.pc_p = pop\_resp(is.keyed\_val(is.key_p))$. In any other case when $is.pc_p = deciding$ we set $as.pc_p = pop\_resp(\perp)$.

Proving the transfer condition is by far the most involved aspect of the proof. The proof is a long and tedious case analysis on transitions, and has been checked using the PVS proof assistant. As we did in Chapter 3, we present only a small fragment in detail: the proof that the $SeqOk$ predicate is preserved by $observe$ actions.

Successful $observe$ Transitions

Fix a process p, intermediate states is, is' such that $is \xrightarrow{observe_p} is'$ and abstract state as' and match function m such that $R(as', is')$ and $SeqOk(as', is', m)$. We must choose a state as such that $SeqOk(as, is, m')$ where

$$m' = \begin{cases} m \oplus \{\, as'.deq.left \to is'.kdeq.left + 1\} & \text{if, for some } v\ as'.pc_p = pop\_resp(v) \wedge \\ & \quad is'.pc_p = pop\_inv(left) \\ m \oplus \{\, as'.deq.right \to is'.kdeq.right - 1\} & \text{if, for some } v\ as'.pc_p = pop\_resp(v) \wedge \\ & \quad is'.pc_p = pop\_inv(right) \\ m & \text{otherwise} \end{cases}$$

Furthermore, we must show that either $as = as'$ or $as \xrightarrow{do\_pop_p} as'$.

There are three cases to consider:

1. $as'.pc_p = pop\_resp(v)$ for some v, and $cs'.pc_p = pop\_inv(left)$
2. $as'.pc_p = pop\_resp(v)$ for some v, and $cs'.pc_p = pop\_inv(right)$
3. $as'.pc_p \neq pop\_resp(v)$

We first describe the construction of as and the proof that $SeqOk(as, is, m')$ and $as \xrightarrow{do\_pop_p} as'$ for the case where $cs'.pc_p = pop\_inv(left)$. The construction and proof for the case where $cs'.pc_p = pop\_inv(right)$ is symmetric. Second, we describe a proof that if $as'.pc_p \neq pop\_resp(v)$, then $SeqOk(as', is, m')$. Note that we are choosing an abstract action and prestate based on whether p's operation has "already" been linearised.

In the case where $as'.pc_p = pop\_resp(v)$, we define as to be the unique state satisfying

$$as.pc_p = pop\_inv(left) \wedge \tag{4.26}$$
$$(\forall\, q \bullet q \neq p \Rightarrow as.pc_q = as'.pc_q) \wedge \tag{4.27}$$
$$as.deq.right = as'.deq.right \wedge \tag{4.28}$$
$$as.deq.left = as'.deq.left - 1 \wedge \tag{4.29}$$
$$as.deq.seq = as'.deq.seq \oplus \{as'.deq.left \to is'.keyed\_val(is'.key_p)\} \tag{4.30}$$

Informally, we change p's program counter to $pop\_inv(left)$ indicating that p's operation has not yet been linearised in as; we extend the sequence $deq.seq$ on the left by one; and we set the value at this new index to be the value associated with key_p after the pop. Going backwards, we are adding the value that p will eventually return to the sequence. Everything else remains the same.

Note that there is only one disjunct of $CorrespondenceOk(as', is')$ that is consistent with

$$\begin{aligned} is'.pc_p &= pop\_inv(s) \wedge \\ as'.pc_p &= pop\_resp(v) \wedge \\ v &\neq \bot \end{aligned}$$

That is, $WinningPopping$. Because of this fact,

$$v = is'.keyed\_val(is'.key_p)$$

We must prove that $as \xrightarrow{do\_pop_p} as'$. We prove that

$$as.pc_p = pop\_inv(left) \wedge \tag{4.31}$$

$$\neg\, empty(as.deq) \wedge \tag{4.32}$$

$$as.deq.left = as'.deq.left - 1 \wedge \tag{4.33}$$

$$as.deq.right = as'.deq.right \wedge \tag{4.34}$$

$$as'.pc_p = pop\_resp(as.deq.seq(as.deq.left + 1)) \tag{4.35}$$

The first is true by construction. The second is true because

$$\begin{aligned} as.deq.left &= as'.deq.left - 1 \\ &< as'.deq.right && < as.deq.right - 1 \end{aligned}$$

The third and fourth statements are true by construction. The fifth is true because

$$as.deq.seq(as.deq.left + 1) = is'.keyed\_val(is'.key_p) \qquad = v$$

and$as'.pc_p = pop_resp(v)$.

We now prove that $SeqOk(as', is, m')$, defined in Figure 4.17 on page 115. There are three universally quantified formulae to verify. To verify the first two of these, fix an i, such that $as.deq.left < i < as.deq.right$. Assume first that $i \neq as.deq.left + 1$. In this case, $m'(i) = m(i)$ and $as'.deq.left < i < as'.deq.right$. We prove each of 4.5 to 4.7 in turn.

- $is.kdeq = is'.kdeq$, so 4.5 follows from the fact that $SeqOk(as', is', m)$, and $as'.deq.left < i < as'.deq.right$.
- Because $is.keyed\_val = is'.keyed\_val$ and $is.kdeq = is'.kdeq$, we have $\sigma(is) = \sigma(is')$. This, and the fact that $SeqOk(as', is', m)$ gives us

$$\begin{aligned} \sigma(is)(i) &= \sigma(is')(i) \\ &= as'.deq.seq(i) \\ &= as.deq.seq(i) && \text{since } i \neq as.deq.left + 1 \end{aligned}$$

- The negation of the $WinnerExists$ predicate applied to *as*, *is*, and $k = is.kdeq.seq(m'(i))$ implies that there is no process *q* that such that $is.key_q = k$ and $as.pc_q = pop\_resp(v)$ for some *v*. If there were no such process in the states *as*′ and *is*′ (as is guaranteed by the fact that $SeqOk(as', is', m)$ and $i \neq as.deq.left + 1$), then it is enough to show that *p* is not such a process in the states *as* and *is*. This is easy to see, because

  $$is.pc_p = pop\_inv(left) \neq pop\_resp(v)$$

  for any *v*.

To prove 4.8, fix a *j* such that $i < j < as.deq.right$. Because $i \neq as.deq.left + 1$ and $j \neq as.deq.left + 1$, we have $m'(i) = m(i)$ and $m'(j) = m(j)$, and thus $m'(i) < m'(j)$.

Now assume that $i = as.deq.left + 1$, so that $m'(i) = is'.kdeq.left + 1$. Again, we prove each of 4.5-4.7 in turn.

- $$\begin{aligned} is.deq.left &< is.deq.left + 1 \\ &= m'(i) \end{aligned}$$
  $$\begin{aligned} m'(i) &= is.deq.left + 1 \\ &< is.deq.right \end{aligned}$$

  The last inference holds because the transition relation implies that $empty(is.kdeq)$.

- $$\begin{aligned} \sigma(is)(i) &= is.keyed\_val(is.kdeq.seq(i)) \\ &= is.keyed\_val(is'.key_p) \\ &= as.deq.seq(i) \end{aligned}$$

- Again, because $as.pc_p = pop\_inv(left)$, we know that *p* is not the winner of $k = is.kdeq.seq(m'(i))$. However, because $i = as.deq.left + 1$, we cannot simply argue that there was no winner for *k* in the intermediate and abstract poststates, so there is no winner in the prestates. In fact, *p* is a winner for *k* in the poststates. This is because

  $$\begin{aligned} is'.key_p &= is'.kdeq.seq(is'.kdeq.left + 1) \\ &= is.kdeq.seq(is.kdeq.left + 1) \\ &= is.kdeq.seq(m'(i)) \\ &= k \end{aligned}$$

  and $as'.pc_p = pop\_resp(v)$. However, the fact that $WinnerUnique(as', is')$ is enough to show that *p* is the only winner for *k* in the poststates. Therefore, because *p* is not a winner in the prestates (on account of its program-counter value), we have $\neg WinnerExists(as, is, k)$.

To prove 4.9 and 4.10, fix an i such that $is.kdeq.left < i < is.kdeq.right$. Assume that $i = is.kdeq.left + 1$. In this case $m'(as.deq.left + 1) = i$ and

$$\begin{aligned} as.deq.left &< as.deq.left + 1 \\ &< as.deq.right \end{aligned}$$

so $as.deq.left + 1$ provides a witness that $InMatchRange(as, m', i)$.

Now assume that $i \neq is.kdeq.left + 1$. If $InMatchRange(as', m, i)$, then $InMatchRange(as, m', i)$, because the range of m' over the set $[as.deq.left..as.deq.right]$ is a superset of the range of m over the set $[as.deq.left..as.deq.right]$.

If $WinnerExists(as', is', is'.kdeq.seq(i))$ then the situation is more complicated. We must show that there is still a winner for $k = is'.kdeq.seq(i)$. To do this, we must show that the winner of k is not the process p. Assume for a contradiction that $is'.key_p = k$. By the transition relation of *IntAut*, this implies that

$$k = is.kdeq.seq(is.kdeq.left + 1)$$

which gives us

$$\begin{aligned} k &= is.kdeq.seq(is.kdeq.left + 1) \\ &= is'.kdeq.seq(i) \\ &= is.kdeq.seq(i) \end{aligned}$$

So we can prove, under the assumption that $is'.key_p = k$, that

$$is.kdeq.seq(i) = is.kdeq.seq(is.kdeq.left + 1)$$

Recall that $i \neq is.kdeq.left + 1$. *IntAut* has the invariant that, for all x, y

$$\begin{aligned} kdeq.left < x < kdeq.right \wedge \\ kdeq.left < y < kdeq.right \wedge \\ kdeq.seq(x) = kdeq.seq(y) \Rightarrow x = y \end{aligned}$$

It is easy to see why this is so. *IntAut* has the variable $used$ which constrains keys to be fresh when they are pushed onto $kdeq$. However, this invariant implies that if $i \neq is.kdeq.left + 1$, then

$$is.kdeq.seq(i) \neq is.kdeq.seq(is.kdeq.left + 1)$$

which provides our contradiction.

It remains to consider the case where $as'.pc_p \neq pop\_resp(v)$ for any v. In this case, we set $as = as'$.

4.5 Related Work

Backward simulations have been used several times to do verifications involving IO automata [BGLR01, SAGG+93, Smi96]. All of these verifications have followed the same pattern as ours: a forward simulation demonstrating trace inclusion between a concrete automaton and an intermediate automaton; and a backward simulation showing trace inclusion between the intermediate automaton and the abstract automaton. Moreover, the intermediate automaton is defined to be as close as possible to the abstract automaton, differing only in that it captures the "backwards" behaviour. What separates the verification presented in this chapter from these prior examples is the relative complexity of our backward simulation. While we cannot argue formally that this complication is essential, it seems to us that the linearisation points of the intermediate automaton force us to use a complicated simulation relation. As has already been noted, the pop operations may be linearised at actions belonging to another operation, and may be linearised at actions that do not modify the shared data structure. This forces us to use a nonobvious relationship between the shared structures (the $kdeq$ and deq, expressed in $SeqOk$), as well as maintain subtle information about the relationship between processes, within the simulation. For these reasons, the complexity of our simulation relation approaches or exceeds that of many backward simulations in the literature.

There are other approaches to dealing with prophetic linearisation in the context of transition systems. These approaches make use of auxiliary variables to relay information about the future of an execution to the point where a linearisation decision must be made. Prophecy variables [AL91] can be used in this way. Eternity variables [Hes02b, Hes05] are similar to prophecy variables in that they can be used to verify algorithms exhibiting prophetic linearisation, but differ in that they avoid technical limitations on the soundness of prophecy variables and backward simulation. (In our setting, this amounts to avoiding the requirement of image finiteness.)

[Hes02a] applies a correctness condition adapted to the verification of read/write registers (locations supporting only read and write operations) to the algorithm of [Blo88], which exhibits prophetic linearisation. It is unclear whether this correctness condition can be generalised to other datatypes.

4.6 Concluding Remarks

In this chapter we have presented an elaborate backward simulation. This work has two goals: to complete a proof that the Snark algorithm is correct, and to explore the construction of backward simulation relations for the verification of algorithms that require nontrivial backward simulations. Our interest in such techniques arises from the relative prevalence of nonblocking algorithms that exhibit prophetic linearisation, and the fact that such algorithms often require subtle backward simulations. As with all the verifications in this thesis, our complete proof has been checked using the PVS proof assistant.

Part II

Dynamic-sized Nonblocking Data Structures

Chapter 5

Nonblocking Storage Reclamation

Recall from Chapter 1 that many nonblocking algorithms suffer from serious drawbacks that restrict their range of applicability. These include the inability to reliably release memory back to the system; the need to know in advance the number of processes that will access a given data structure; and dependence on rare or unimplemented synchronisation primitives. Part II of this thesis develops techniques for overcoming these limitations. Whereas Part I is about the verification of nonblocking algorithms, Part II is about their design. The work on verification is of independent interest, but also informs the work in Part II. We apply techniques developed in Part I to the verification of the principle result of Part II: the implementation of LL/SC variables described in Chapter 6.

This chapter describes a novel *lock-free reference counting* technique (abbreviated LFRC), that enables processes to safely and reliably release memory back to the system. The technique has two principle advantages over previous proposals. It does not require advanced knowledge of the number of processes that will ever use the system, nor does it depend on the existence of exotic synchronisation primitives such as DCAS. However, it does require that the system provide a CAS or LL/SC operation capable of atomically testing and modifying a pointer and an adjacent integer. This implies that the technique cannot be used in many contemporary systems. However, Chapter 6 describes an implementation of the LL/SC synchronisation primitive (the LL/SC operations are defined in Section 1.1.2) that can be applied to a location containing a pointer, as well as other information. This LL/SC implementation can be combined with the LFRC technique of this chapter to overcome the reliance on an operation that can atomically modify a pointer and an integer. Further, this combination preserves all the advantages of the LFRC technique regarding memory reclamation and the number of processes that will use the system.

We present our LFRC technique as a programming interface that could be used by a client application. We also present a lock-freedom preserving and mechanical transformation from code that does not recycle memory to behaviourally equivalent code that uses our LFRC interface to recycle memory.

The remainder of the Chapter is organised as follows. Section 5.1 defines important con-

cepts necessary for understanding the contributions contained in Part II. Section 5.2 gives a brief overview of reference counting. Section 5.3 describes the interface to our LFRC functionality, and Section 5.4 describes the transformation of code that does not recycle memory to code that does. Section 5.5 applies our transformation to Treiber's stack algorithm [Tre86]. Section 5.6 describes the LFRC implementation in detail. Section 5.7 provides an overview of several other nonblocking memory management techniques, and a description of the strengths and weaknesses of our approach, in comparison with other proposals.

5.1 Pointer-cleanliness, Space-adaptivity, and Population Obliviousness

Before describing the LFRC result, we explain and define some desirable properties of nonblocking algorithms that have been difficult to achieve. These properties are important for understanding the contributions presented in Part II of this thesis.

5.1.1 Wide Synchronisation Primitives and Pointer Cleanliness

For some time, 64-bit architectures have been available [Hei91, Mot93, Sit92, WG94]. These architectures support 64-bit addresses, allowing direct access to huge virtual address spaces [CBHLL92]. They also support atomic access to 64-bit values using synchronisation primitives such as CAS. Operating systems and application software that exploit 64-bit addressing have been slower to emerge. Thus, many important 32-bit operating systems and applications are still in common use, and most 64-bit architectures support them. As a result, for a period of several years, techniques that use 64-bit synchronisation primitives to atomically manipulate 32-bit pointers together with other information, such as version numbers, have been broadly applicable. As discussed in Chapter 1, practical lock-free data structures commonly exploit such techniques (e.g., [MS96b, Tre86]). The increasing prevalence of 64-bit operating systems and applications signals the end of this era. Therefore, it is important to develop algorithms that do not depend on the ability to atomically manipulate a pointer and other information.

A *wide synchronisation primitive* is a primitive that can atomically modify a location containing a pointer simultaneously with modifications to some set of other locations. A *narrow* synchronisation primitive is one that is not wide. For example, a CAS operation in a 32-bit system where CAS can be applied to a 64-bit value is a wide synchronisation primitive; a CAS operation in a 64-bit system where CAS can be applied to 64-bit values is narrow. DCAS is another wide synchronisation primitive. The DCAS operation is defined in Section 4.1.1 on page 95, but recall that DCAS enables a simultaneous comparison and conditional modification of two independent locations. An algorithm is called *pointer-clean* if it can run on systems that do not provide any wide synchronisation primitives.

5.1.2 Space-adaptivity

Recall from Chapters 1 and 3 that in both the Treiber stack and M&S queue, it is not safe to simply return unused memory to the system. This is because neither algorithm provides any way for a process to determine when some other process has a pointer to some node that may be dereferenced. This limitation is very common among nonblocking algorithms, and is an important drawback in many systems where available memory is restricted. An algorithm is *space-adaptive* when it does not suffer from this problem.

More precisely, a space-adaptive implementation of a datatype uses space that is bounded by a function proportional to the current size of the data structure, plus the number of pending operations.[1] The notion of *size* of the data structure must be defined precisely for each datatype being implemented. To provide an example, we follow [HLM03b] and analyse the adaptivity of the M&S queue. Define *queue size in state s* to be the number of queue nodes reachable from the `Head` pointer in *s* minus one (ie., the number of nodes in the queue, not counting `Dummy`). This is the number of elements contained in the abstract queue represented by *s*. The M&S queue is not space adaptive because it does not free queue nodes to memory, and so the total memory consumed is not bounded by the current size of the queue. The maximum space consumed by the M&S queue in a given state *s* is in fact proportional to the maximum queue size in any state that occurs prior to *s* in any execution.

This analysis suggests the flavour of space adaptivity, but applies only to the M&S queue. To make the notion of space adaptivity precise in general, we define a notion of *f-space-adaptivity*, where f is a function from the states of an algorithm to the natural numbers. The idea is that the function f returns the current size of the abstract data structure that is represented by the given state. We thus refer to f as a *size function*.

Definition 5.1 (f-space-adaptive)
An algorithm is *f-space-adaptive* if, for every state appearing in any execution of the algorithm, the space used is proportional to f applied to that state plus a constant times the number of pending operations in that state.

Clearly, the choice of size function is important. For container objects like stacks and queues, a natural choice is the number of values that the object currently contains. We might think of an LL/SC variable as a container that contains precisely one element (the variable's value), in which case a natural notion of size for an LL/SC variable would be some constant. However, in Section 6.1, we argue that the appropriate notion of size for an LL/SC variable depends on the number of outstanding LL operations for that variable.[2]

We frequently suppress the size function f when discussing the space-adaptivity properties of algorithms. Thus we say that a given algorithm is *space-adaptive* (rather than *f-space-adaptive*) when there is some reasonable size function f for which the algorithm in question

[1] Recall from Section 2.2.2 that a pending operation is an operation with an invocation, but no matching response.

[2] Recall from Section 1.1.2 that an outstanding LL operation is an LL operation that is not matched by an SC operation of the same process.

is f-space-adaptive.

5.1.3 Weak Space-adaptivity

The LL/SC implementation presented in the next chapter is space-adaptive according to the definition just given. However, the LFRC-based transformation presented in Section 5.4 is not guaranteed to yield space-adaptive algorithms in the sense just described. This is a limitation of reference counting itself, and is not particular to our approach. Section 5.2 describes this limitation in detail. In this section we define a notion of *weak space-adaptivity* that captures the space-adaptivity properties of reference-counting based transformations, under reasonable assumptions about the original algorithm. As discussed in Section 5.7, some other nonblocking memory-reclamation techniques satisfy this notion of weak space-adaptivity, without satisfying space-adaptivity proper. A *quiescent state* is a state where there are no pending operations.

Definition 5.2 (Weakly f-space-adaptive)
An algorithm is *weakly f-space-adaptive* if, in every quiescent state appearing in any execution, the space consumed by the algorithm is proportional to f applied to that state.

So weak f-space-adaptivity constrains memory use, but only in quiescent states. An f-space-adaptive algorithm is also weakly f-space-adaptive. This is because an f-space-adaptive algorithm may consume some bounded quantity of memory for each pending operation, but when there are no pending operations (as in quiescent states), the space used must be proportional to f.

We sometimes use the expression *strongly space-adaptive*, when we wish to emphasize that an algorithm is space-adaptive, rather than merely weakly space-adaptive.

5.1.4 Population Obliviousness

A common technique in nonblocking algorithm design is to provide each process with one or more *single-writer/multi-reader* variables (SW/MR variables). All processes are able to read these variables, but only one process ever modifies each variable. (Some examples of this technique [AM95, HLM02b, JP03, Mic04] are discussed in Section 5.7, and in Section 6.4. Further examples can be found in [Her91, LMS03a].) Typically, such SW/MR variables are implemented using an array, with processes using their own identifiers as indexes into the array. Each variable is thus an entry in the array that can be written by the process whose identifier is the index of the entry, and can be read by any other process. This technique requires knowledge of the maximum number of processes that will ever use a given instance of the algorithm, so that an array of appropriate size can be allocated. Therefore, these algorithms are implicitly parameterised by the maximum number of processes for which a given instance of the algorithm will function correctly. We call an algorithm that is parameterised by the maximum number of processes *population aware*. An algorithm is *population oblivious* iff it is not population aware [BMV+07].

In systems where processes can be created dynamically, it may be impossible to determine the maximum number of processes that may ever access a data structure. In such situations, population-oblivious algorithms are required. Further, when SW/MR variables are used, space must be allocated for these variables. This space cannot be deallocated for the life-time of the data structure. This precludes space-adaptivity.

5.1.5 LFRC and LL/SC

The reference-counting technique presented in this chapter is population oblivious, but not pointer clean. Using our transformation, it is possible to obtain weakly space-adaptive algorithms from garbage collection dependent algorithms. Further, it is possible to modify an existing algorithm, such that the result of our LFRC-based transformation yields a strongly space-adaptive algorithm. In Section 5.5.1, we show by example how this is achieved. These modifications require insight on the part of the programmer, and cannot be described as a mechanical transformation.

The LL/SC implementation presented in Chapter 6 is strongly space-adaptive, pointer-clean and population oblivious. To our knowledge, it is the first published nonblocking LL/SC implementation to enjoy all three of these important properties. Also, because it has these three properties, it can be combined with the LFRC result to obtain a general memory management technique that is pointer clean, population oblivious, and enables the construction of strongly space-adaptive algorithms. We describe this combination in Section 6.3.

5.2 Reference Counting

We briefly review reference counting, before going on to describe our technique. Reference counting [Col60] is a classical technique for reclaiming unused memory, that is used in some garbage collection systems (e.g., [AKW88, WS91]). There are numerous variations (for example [Wis93, DB76, LP01]), but here we recap the main idea. Readers are referred to [JL96] for a detailed account.

Each object is associated with a *reference count* that counts the number of references to that object (references to that object currently stored in local variables or shared locations). This count is typically stored in a field of each object. When the reference count of an object falls to zero, the object is no longer accessible in the heap, and so can be deallocated. Every time a reference to some object `o` is created (which happens when a reference to `o` is stored in some variable or location) the reference count associated with `o` is incremented. Every time a reference to some object `o` is destroyed (which happens when a variable or location containing a reference to `o` is over-written, or an object containing a reference to `o` is deallocated) the reference count associated with `o` is decremented. Whenever the reference count of an object falls to zero, the object is deallocated. Because the object may hold references to other objects, this may result in further deallocation.

Reference counting alone can only reclaim memory from data structures that do not contain cycles of references: i.e., structures such that there is no path of references from any

object back to itself. In a cycle of references, every object has a non-zero reference count: for every object o in the cycle, there is some other object in the cycle holding a reference to o. A cyclic structure may thus become unreachable, but its reference counts will never fall to zero, and so it will not be deallocated by the basic reference counting scheme described above. Reference counting systems typically overcome this problem in one of two ways. Some use a back-up garbage collector that periodically searches the entire heap for unreachable memory (for example, [DeT90]). Others use a *cycle collector* [Chr84], that searches for cycles among objects that have nonzero reference counts (for example, [MWL90, PBK+05]).

In this chapter, we apply our LFRC solution to transform code that does not recycle memory into code that does. All reference counting techniques can be used to obtain such a transformation. The resulting code is guaranteed to be functionally equivalent to the original code, and to be free from accesses to deallocated memory. Further, reference-counting based transformations provide the following guarantee about which objects will be freed. If the original code has the property that no object is part of a reference cycle when it becomes unreachable, then in the transformed code, all objects will be deallocated before they become unreachable [JL96]. This guarantee is important to our discussion of the space-adaptivity properties of code transformed using our LFRC technique (in Sections 5.4.4 and 5.5.1).

5.2.1 Lock-free Reference Counting

The implementation of lock-free reference counting is challenging because it is difficult to safely update the reference count of an object in a lock-free context. A process *p* may read a reference to an object o from a location, but o may be subsequently deallocated before *p* can increment o's reference count. This can happen when another process causes o's reference count to fall to zero after *p*'s read, and deallocates the memory. This is another instance of the problem that precludes memory being released from data structures such as the Treiber stack and M&S queue.

Lock-free reference counting is much simpler in an environment where it is legal to access the reference-count field of an object after it has been deallocated. Some lock-free reference-counting techniques are designed to work in such an environment [Val95, Rei04], and we describe these solutions in Section 5.7. However, in most systems, once an object has been deallocated, there are no guarantees as to the legality of any particular access to the fields of the object, nor any guarantees about the contents of those fields. An environment in which access to deallocated objects is legal can be emulated using an application freelist (as in [Val95]), but this precludes freeing memory to the system, and thus precludes space-adaptivity.

Our LFRC technique works by distinguishing between different contributions to the reference count of each object. For each object o, the count of the number of shared locations (locations accessible to more than one process) containing a reference to o is maintained separately from the count of the number of local variables that reference o. The object o cannot be deallocated until both counts fall to zero. When a process *p* reads a reference to o from a shared location into a local variable, the count of the number of local references to o

is incremented, without *p* needing to access `o` directly. Section 5.6 describes in detail how this is achieved.

5.3 The Lock-free Reference Counting Interface

We first present the LFRC interface. Later, in Section 5.4, we describe how to use this interface to transform code that does not recycle memory to code that does. As usual, we employ C-style pseudocode to describe the LFRC interface.

Fix a type `T` to represent the type of application level objects (that is, the type of objects that are to be reclaimed using the LFRC technique). In a real programming language, this type could be specified using a type parameter, or it could be a particular type, or it could be identified with an `Object` type at the top of the type hierarchy. Fixing the type `T` simplifies the following presentation.

Our LFRC technique uses several counters for each object, each of which counts references from a different source. However, in order to describe the LFRC interface we pretend that each object is directly associated with a single abstract reference count. This pretence allows us to abstractly specify the behaviour of the LFRC interface, without describing the implementation details.

Two types are exported from the LFRC interface: `RC_Ref` and `RC_Obj`. Members of the type `RC_Ref` represent references to objects, and members of the the type `RC_Obj` contain the application level objects. Each value of type `RC_Ref` has a field `ref` that yields a value of type `RC_Obj*` (i.e., a pointer to an `RC_Obj`). Each value of type `RC_Obj` contains an object of type `T`, which can be accessed using its `v` field. Abstractly, each object of type `RC_Obj` has an associated reference count, with the exception of `null`, whose reference count is undefined.

As has already been mentioned, our LFRC technique depends on a distinction between *local* and *shared* locations. For our purposes, local locations exist on the stack of some process and are only accessible to that process. Shared locations exist in statically or dynamically allocated storage, and may be accessible to more than one process. References in shared locations are represented using objects of type `RC_Ref`. References in local locations are represented using objects of type `RC_Obj*`. To ensure that memory is not reclaimed prematurely, values of type `RC_Obj*` should never be written directly into shared locations by application code.

We now describe the procedures provided by the LFRC interface.

```
void RC_Load(RC_Obj **o, RC_Ref *r)
```

`RC_Load(o,r)` copies the pointer stored at `r->ref` into the location pointed to by `o`. If `r->ref != null` the reference count associated with `*(r->ref)` is incremented. If the value of `*o` before the operation is not `null`, then the reference count associated with `**o` is decremented. The location pointed to by `o` must be local (i.e., must be on the stack of the

process).

`void RC_Store(RC_Ref *r, RC_Obj *o)`

`RC_Store(r,o)` stores `o` into `r->ref`. If the value of `r->ref` before the operation is not `null`, then the reference count associated with this value is decremented. If `o != null` the reference count associated with `*o` is incremented. The location pointed to by `o` must be local.

`boolean RC_CAS(RC_Ref *r, RC_Obj *old, RC_Obj *new)`

`RC_CAS` is an implementation of the CAS operation to be used against instances of `RC_Ref`. If `r->ref = old` the CAS is successful and `r->ref` is changed to `new`. Otherwise, the CAS is unsuccessful and `r->ref` is unchanged. If the CAS is successful and `old != null`, then the reference count associated with `*old` is decremented, and if `new != null`, then the reference count associated with `*new` is incremented. If the CAS is unsuccessful, no reference counts are modified.

`void RC_Destroy(RC_Obj *o)`

`RC_Destroy` is used to destroy local references before they are overwritten or go out of scope. If `o != null` the reference count associated with `*o` is decremented.

`void RC_Alloc(RC_Obj **o)`

`RC_Alloc` allocates `RC_Obj` objects. `RC_Alloc` sets `*o` to be a pointer of type `*RC_Obj` that was previously unallocated and that has a reference count of 1 after the allocation (to account for the reference created by the allocation). If the previous value of `*o` is not `null`, then the reference count associated with `**o` is decremented. `o` must point to a local location.

`void RC_SharedCopy(RC_Ref *r, RC_Ref *s)`

`RC_SharedCopy(r, s)` copies the pointer `s->ref` into `r->ref`. If `s->ref != null` then the reference count associated with `*(s->ref)` is decremented. If the value of `r->ref` before the operation is not `null`, then the reference count assocated with `*(r->ref)` is decremented. This copy operation is not atomic, in the sense that the values of `s->ref` and `r->ref` may never be identical during the operation. The `RC_SharedCopy` operation only guarantees that at some point in the execution of the operation, the value of `s->ref` is the value eventually written into `r->ref`, and that if no other LFRC operations overlap with a given `RC_SharedCopy` operation, then `s->ref = r->ref` after the operation.

`void RC_LocalCopy(RC_Obj **o, RC_Obj *p)`

`RC_LocalCopy(o,p)` copies the pointer `p` into the location pointed to by `o`. If `p != null` the reference count associated with `*p` is incremented. If the value of `*o` is not `null`

before the operation, then the reference count of `**o` is decremented. `o` must point to a local location.

```
RC_Obj *RC_Pass(RC_Obj *o)
```

`RC_Pass(o)` returns the pointer `o` (so that `RC_LocalCopy(o) = o` is always true) and if `o != null` the operation increments the reference count associated with `*o`. The purpose of `RC_Pass` is to allow reference values to be passed during procedure invocations.

5.4 Transformation

We now describe a transformation from code that does not recycle memory, to functionally equivalent code that recycles memory using our LFRC technique. Our transformation provides an alternative to garbage collection, for use in environments where garbage collection is inappropriate, and a transformation like ours could also be used in an implementation of garbage collection.

The main point of this section is to convince the reader that our LFRC technique could be used to make a large class of algorithms space-adaptive. The source and target of our transformation is essentially the C-style pseudocode that we have been using throughout the thesis, and its syntax and semantics are not formally specified. However, we feel that the presentation is precise enough to be used as the basis for a formal transformation over a specific programming language.

Because the LFRC technique handles shared and local references differently, we need to carefully distinguish between expressions that can be evaluated without reading references stored in shared locations, and those that require reading a shared reference. In order to make this possible, we restrict the expressions and statements that are allowed in the domain of our transformation. Section 5.4.1 defines the set of expressions that may appear in programs that we transform. The constructs from which these expressions are built should be familiar, and should have familiar (informal) meanings. In Section 5.4.2, we define the set of allowed statements, and the transformation itself. Throughout the discussion, we assume that the objects that are to be recycled using our LFRC transformation have type `T`.

5.4.1 Allowable Expressions

The goal of this section is to define a set of expressions, called *allowable expressions*, that may appear within programs in the domain of our transformation. We first define some important categories of expressions, and then the allowable expressions themselves. In what follows, let an *S-variable* be a variable of type `S`.

In accordance with our distinction between local and shared locations, we divide the set of expressions into categories, according to whether the expression is evaluated by reading

local locations, or by reading a shared location. The expressions that are evaluated by reading only local locations are called *local* S*-expressions*, where S is some type.

Definition 5.3 (Local S-expression)
A *local* S*-expression* is an expression of type S in which the only variables are local variables, and the only operators are arithmetic operators and the operators & (address-of), * (dereference), -> (pointer-to-member), new (allocation) and ·[·] (array application).

Note that the restriction on what operators may appear is meant to prohibit procedure invocations. We have prohibited field access (the dot operator) in the interests of simplicity.

Below, we syntactically define a category of expressions that are evaluated by reading precisely one shared location. We call members of this category *shared* S*-expressions*. The definition is complicated by the need to obtain a reasonably broad class of allowable expressions. We first define the category *shared* S*-lvalue*,[3] which are expressions that are evaluated by reading a single shared location, and to which the address-of operator & may be applied.

Definition 5.4 (Shared S-lvalue)
A *shared* S*-lvalue* is an expression of one of the following forms:

- a shared S-variable X,
- an expression x->f, where x is a local variable and x->f has type S,
- an expression of the form a[e] where e is a local integer-expression, and a is a statically allocated array with elements of type S.

The following category of S*-address expressions* includes the expressions A of type S* such that *A is evaluated by reading a shared location.

Definition 5.5 (S-address expression)
An S*-address expression* is a local variable x of type S*, or an application of the address-of operator & to a shared S-lvalue.

Every S-address expression has type S*, but not every expression of type S* is an S-address expression. S-address expressions can be evaluated without reading any shared locations. If A is a shared address expression, then *A may be evaluated by reading a single shared location.

Some programs contain one or more S-address expressions A such that *A is sometimes evaluated by reading only local locations (this is possible, for example, after a process writes the address of one of its local variables into a shared location). This is not a syntactic property of *A, because it depends on the behaviour of the running program. We exclude from the domain of our transformation all programs in which there is any expression *A that is ever evaluated by reading a local location, and such that A is a T*-address expression. This is

[3]The set of shared S-lvalues defined here is a subset of the lvalues of the C programming language.

necessary to enable us to syntactically distinguish expressions of type `T*` that are evaluated by reading a shared location from those that are not.

Now we are in a position to define the class of expressions that are evaluated by reading at most one shared location.

Definition 5.6 (Shared S-expression)
A *shared* S*-expression* is a shared S-lvalue, or an expression of the form `*A` where `A` is an S-address expression.

Each shared S-expression has a single location that must be read in order to evaluate the expression. This location can itself be evaluated using the following function.

Definition 5.7 (Location of a shared S-expression)
The *location* of a shared S-expression `E`, written *loc*(`E`), is defined by cases as follows:

- if `E` is a shared S-lvalue, then *loc*(`E`) is the expression `&E`,
- if `E` is of the form `*A` where `A` is an S-address expression, then *loc*(`E`) is the expression `A`.

The location of a shared S-expression generalises the address-of operator `&`.

The *allowable expressions* are those that fall within one of the following categories, for some type S:

- Expressions of the form `CAS(E, e, f)` where `E` is an S-address expression, and `e` and `f` are local S-expressions. The only shared location that must be read to evaluate such a CAS is the location `E`.
- Expressions of the form `new S()`, each of which allocates a new object of type S, and returns a pointer to that object.
- Local S-expressions.
- Shared S-expressions. Recall that the only shared location must be read to evaluate a shared S-expression `E` is the value of *loc*(`()E`).

So we prohibit expressions that are evaluated by reading more than one shared location. We can obtain the effect of expressions involving more than one shared location by introducing one or more local variables, and decomposing the expression into several statements. For example, the expression `*Y`, where `Y` is a shared variable, is prohibited by the above rules. However, we can emulate a statement of the form

```
x := *Y
```

with the statements

```
x1 := Y;
x := *x1;
```

Note that expressions that are evaluated by reading more than one shared location cannot usually be evaluated atomically, and are typically not appropriate in code that purports to describe a nonblocking algorithm.

In the interests of simplicity, we restrict the types of variables and fields that occur in programs.

- local and shared variables, arrays, and fields can all have any type that does not contain an occurrence of `T*`,
- a local variable whose type contains an occurrence of `T*` must have type `T*` or `T**`,
- a shared variable or field whose type contains an occurrence of `T*` must have type `T*`,
- an array whose type contains an occurrence of `T*` must have type `[T*]` (i.e., the type of the array elements must be `T*`).

The next section describes the transformation of programs whose expressions are all allowable expressions, and whose variables and fields satisfy the given restrictions.

5.4.2 The Transformation

Our transformation is composed of several modifications. We modify the types used in the program, so that dynamically allocated objects are equipped with reference counts. We also modify statements involving CAS operations, assignments and allocations. Each of these latter modifications is designed to ensure that the reference count of each object is updated to accurately reflect the references created and destroyed by each statement. Finally, we add statements to the end of each procedure that decrement the reference count of each local variable declared in the procedure. This reflects the fact that references to objects from local variables of a procedure are destroyed when the procedure exits.

The first step is to translate the types of variables and fields appearing in the program. We denote a type `S` that contains an occurrence of some type `U` as `S(U)`. The key to our translation of types is to turn local `S(T*)`-expressions into expressions of type `S(RC_Obj*)`, and shared `S(T*)`-expressions into expressions of type `S(RC_Ref)`. We translate types according to the following scheme:

1. local `T*`-variables become variables of type `RC_Obj*`,
2. local `T**`-variables become variables of type `RC_Ref*`,
3. shared `T*`-variables become variables of type `RC_Ref`,
4. local and shared `S`-variables, where `S` does not contain an occurrence of `T*`, preserve their type,
5. fields of type `T*` become fields of type `RC_Ref`,

6. arrays of type `[T*]` become arrays of type `[RC_Ref]`,

7. fields and arrays whose type does not involve `T*` preserve their type.

Given these rules, for any type `S(T*)` (that contains an occurrence of `T*`), and any expression `exp` of type `S(T*)` in the original code, `exp` has type `S(RC_Obj*)` in the transformed code iff it is a local `S`-expression, and type `S(RC_Ref)` in the transformed code iff it is a shared `S`-expression.

Next, we replace each expression of the form `e->f`, where `e` is a local `T*`-expression, with the expression `(e->v).f`. This does not change the type of the expression, it simply reflects the fact that we need to evaluate the `v` field of an `RC_Obj*` value, in order to obtain a value of the original type `T`, to which we can apply the field access `.f`. Also, we replace each expression of the form `CAS(E, e, f)` where `E` is a `T*`-address expression and `x` and `y` are local `T*`-expressions, with the expression `RC_CAS(E, e, f)`.

Next, we translate each statement of each procedure one-by-one, in a way that reflects the references created and destroyed by the statement. Below, we define the translation of statements by cases. Any statement not covered by one of these cases is not allowed in the domain of our transformation.

1. An assignment in which both the left- and right-hand sides are allowable expressions, and are not of type `T*`, is simply copied.

2. An assignment of the form `x := new T();`, where `x` is a local `T*`-variable becomes `RC_Alloc(&x);`.

3. An assignment of the form `x := E;`, where `x` is a local `T*`-variable and `E` is a shared `T*`-expression becomes `RC_Load(&x, E);`.

4. An assignment of the form `E := e;`, where `E` is a shared `T*`-expression and `e` is a local `T*`-expression becomes `RC_Store(`*loc*`(E), e)`.

5. An assignment of the form `E := F;`, where `E` and `F` are shared `T*`-expressions becomes `RC_SharedCopy(`*loc*`(E), `*loc*`(F))`.

6. An assignment of the form `x := e`, where `x` is a local `T*`-variable and `e` is a local `T*`-expression becomes `RC_LocalCopy(&x, e)`.

7. A procedure invocation of the form $\texttt{P(e}_1\texttt{,...,e}_n\texttt{)}$ where $\texttt{e}_1\texttt{,...,e}_n$ are the local `T*`-expressions that are actual arguments in `P` becomes

   $$\texttt{P(RC\_Pass(e}_1\texttt{),...,RC\_Pass(e}_n\texttt{))}$$

   `P` may not contain any expressions of type `T*` that are not local expressions.

8. A statement of the form `return e;`, where the expression `e` is not of type `T*` is simply copied.

Note that no `return` statement can return a value of type `T*`.

Finally, for each procedure, and for each local `T*`-variable `x` of that procedure, the statement `RC_Destroy(x)` is added to the end of the procedure by the transformation. This reflects the fact that a reference to each variable is destroyed when the procedure exits.

In Section 5.5, we apply our transformation to an example algorithm.

5.4.3 Limitations

Our LFRC transformation can be used to transform a large class of programs. However, there are several important restrictions. First, references may only be created by allocation, or by copying existing references. Pointers cannot be generated by pointer arithmetic, for example. Other lock-free reference counting techniques have the same restriction [DMMm01, HLMM05].

The second, and most important limitation of our transformation is that the only operators appearing within expressions are arithmetic operators, and the operators `&`, `*`, `->`, `new` and $\cdot$`[`$\cdot$`]`. For example, tuples of values are not allowed, which are commonly used in the presentation of algorithms that exploit version numbering techniques. Moreover, procedure invocations cannot occur within expressions, and a pointer of type `T*` cannot be returned from any procedure. We justify these restrictions on the grounds of simplicity. A formal transformation over a well-defined programming language may be able to relax these restrictions, but such an effort is beyond the scope of this presentation.

However, the utility of our LFRC technique extends beyond the transformation. It is possible for a programmer to determine when a reference count needs to be modified based on application level knowledge, as opposed to simple syntax. For example, the value of a local `T*`-variable could be returned from a procedure by eliding the call to `RC_Destroy` for that variable. The programmer could then arrange to decrement the reference count of that value at some later point. Clearly, care would be needed.

5.4.4 Obtaining Weak Space-adaptivity

Our transformation produces code satisfying weak f-space-adaptivity when two conditions are satisfied:

- In quiescent states of the original algorithm, all memory, except for some quantity proportional to f applied to the state, is unreachable.
- This unreachable memory does not contain reference cycles.

It is easy to see why these conditions are sufficient for weak f-space-adaptivity. Recall that code transformed to use reference counting will deallocate all objects before they become unreachable, so long as, in the original code, no object is part of a reference cycle when it becomes unreachable. Thus, if the second condition above holds, then in quiescent states, all unreachable objects have been deallocated. Further, if the first condition holds, the set of

reachable objects consumes memory allowed by the size of the data structure being implemented.

To see why our transformation is not enough to guarantee *strong* space-adaptivity, consider the case where some process is delayed while it holds a reference to some object. This object is reachable, and thus will not be deallocated. In general, it may contain references to other objects, giving them a nonzero reference count. In turn, these objects may contain references to further objects, none of which may be deallocated by reference counting. The algorithm in question may be such that this set of objects reachable from the delayed process may be unbounded. Thus, the space consumed may exceed any bound on the memory allowed for pending operations by the definition of strong space-adaptivity. We discuss this issue with regards to a specific algorithm in Section 5.5.1.

5.5 Transforming A Stack Algorithm

In this section, we apply the transformation described above to a variant of the Treiber stack given in Section 1.1.3. We show how to obtain both weakly and strongly space-adaptive versions of this algorithm.

Section 1.1.3 describes several variations of the Treiber stack. The variant that we transform is one that relied on garbage collection to recycle memory (presented in Figure 1.6 on page 10). Using LFRC, we transform it to an algorithm that explicitly recycles memory.

We present the original stack algorithm in Figures 5.1 and 5.2. This implementation uses CAS rather than LL/SC to modify the `Head` variable during push and pop operations. The type of `Head` is simply `*Node`, no version numbers are used. Nodes are not explicitly freed after being removed from the stack. Indeed, as discussed in Section 1.1.3, explicitly freeing nodes at the end of pop operations would result in an incorrect algorithm. This is for two reasons, both of which stem from the fact that the algorithm provides no way for one process to determine that no other process can dereference a pointer to a node, after the node has been removed from the stack.

- A process executing a pop operation may dereference a pointer to a node that has just been freed (by following a local `head` pointer).
- A pointer may be prematurely recycled back onto the stack while some process still has a pointer to the node, giving rise to the ABA problem.

Our LFRC transformation produces an algorithm in which it is possible to determine that a pointer will no longer be dereferenced.

We now apply the LFRC transformation. The transformed declarations are presented in Figure 5.3; the transformed pseudocode is presented in Figure 5.4. These declarations and code are obtained directly from Figures 5.1 and 5.2 by a simple application of the transformation described in Section 5.4. Shared variables and objects are associated with reference counts. Every time a reference is created, the corresponding object's reference count is incremented, and every time a reference is destroyed the corresponding object's reference count

```
struct node {                          node *Head;
  val val; node *next                  initially Head = null;
}
```

Figure 5.1: The node structure, the global variable `Head`, and the initial condition for the original stack implementation.

```
void push(val v) {
H1.  node * nd, head;
H2.  nd := new node();
H3.  nd->val := v;
H4.  while(true) {
H5.    head := Head;
H6.    nd->next := head;
H7.    if (CAS(&Head, head, nd))
         break;
H8.  }
H9.  return;
}


val pop() {
P1.  node * head, next;
P2.  while (true) {
P3.    head := Head;
P4.    if (head = null)
P5.      return empty;
P6.    next := head->next;
P7.    if (CAS(&Head, head, next))
         break;
P8.  }
P9.  return head->val;
}
```

Figure 5.2: Pseudocode for the original stack operations.

```
struct node {                          RC_Ref Head;
  val val; RC_Ref next                 initially Head.ref = null;
}
```

Figure 5.3: The transformed node and stack structures.

is decremented. Note that initially `Head.ref = null`. We do not constrain the initial reference count of `Head` because the reference count of `null` is undefined.

As with any garbage collection technique, reference counting solves the ABA problem. In particular, once a popping process has executed the read at line TP3, the reference count of `*head` is guaranteed to be nonzero until the process executes line TP12. Therefore, it cannot be deallocated and subsequently pushed back onto the stack.

Because the stack implementation never produces reference cycles, and there is one node per value in the stack, the transformed stack implementation presented in Figure 5.4 is weakly space-adaptive. However, it is not strongly space-adaptive. To see why, consider a popping process *p* that is delayed after reading a pointer to some object `o` from `Head` at line TP3. Once this load has completed, the reference count of `o` will not fall to zero, at least until *p* completes one iteration of the loop. Neither will the reference counts of any of the objects reachable from `o`. All these objects may be removed from the stack by pop operations that complete while *p* is delayed. Once this has happened, the memory consumed by these objects cannot be accounted for as part of the stack data structure. Further, this memory is not bounded by any constant, as the stack may be of any size when *p* executes TP3. Therefore, the transformed code is not strongly space-adaptive.

5.5.1 Obtaining Strong Space-adaptivity

We now describe how to obtain a strongly space-adaptive version. The key is to "break the chain" of references that allow one delayed process to keep an unbounded number of objects from being deallocated.

We only need to modify the `pop` implementation. Observe that, once a node has been removed from the stack, its `next` field can be overwritten without affecting the representation of the stack. Thus, we make popping processes overwrite the `next` field with `null` after a successful CAS. It is possible for a delayed popping process to read `null` from this field after it has been overwritten. However, this can only happen after the `Head` variable has been modified since the delayed process read `Head` at TP3. Thus, the CAS of the delayed process is doomed to fail, so there can be no visible change to the state of the data structure, and the delayed process will simply retry the loop.

Figure 5.5 presents code for the modified `pop` operation. The only change is at line P9, where `head->next` is overwritten with `null`. We claim that the memory consumed by this transformed stack is bounded by a multiple of the number of nodes in the stack, plus *three* nodes for every pending operation. The short explanation for this is that no chain of nodes from the `head` variable of either operation can be longer than one node, without there being at least one process with a pending pop operation that has not yet executed the assignment at P9. However, since this is the first claim that an algorithm presented in this thesis is strongly space-adaptive, we provide a more detailed argument.

Because reference counting guarantees to deallocate memory before it would become unreachable in the original algorithm, we need only account for memory that is reachable from `Head`, or a local variable of a process executing a pending operation. The memory

```
void push(val v) {
TH1.   RC_Obj* nd, head;
TH2.   RC_Alloc(&nd);
TH3.   RC_Store(&(nd->v).val, v);
TH4.   while(true) {
TH5.     RC_Load(&head, &Head);
TH6.     RC_Store(&(nd->v).next, head);
TH7.     if (RC_CAS(&Head, head, nd))
           break;
TH8.   }
TH9.   RC_Destroy(nd);
TH10.  RC_Destroy(head);
TH11.  return;
}


val pop() {
TP1.   RC_Obj* head, next;
TP2.   while (true) {
TP3.     RC_Load(&head, &Head);
TP4.     if (head = null)
TP5.       return empty;
TP6.     RC_Load(&next, &(head->v).next);
TP7.     if (RC_CAS(&Head, head, next)){
TP8.       RC_Destroy(next);
TP9.       break;
TP10.    }
TP11.  }
TP12.  RC_Destroy(head);
TP13.  RC_Destroy(next);
TP14.  return (head->v).val;
}
```

Figure 5.4: Pseudocode for the transformed stack operations.

```
val pop() {
P1.   Node * head;
P2.   while (true) {
P3.     head := Head;
P4.     if (head = null)
P5.       return empty;
P6.     next := head->next;
P7.     if (CAS(&Head, head, next))
          break;
P8.   }
P9.   head->next := null;
P10.  return head->val;
}
```

Figure 5.5: Variant of the pop operation. The transformation applied to this code, along with the push implementation of Figure 5.2, yields a strongly space-adaptive algorithm.

reachable from `Head` is proportional to the size of the stack (one object per value). So if our claim is false, then there is some state in which there are four or more nodes reachable from the local variables of some process, that are not reachable from `Head`, nor reachable from some other process. We show that this is impossible for each of the operations. Our proof depends on the the following important property of the stack:

> If there is a chain of $n + 1$ nodes reachable from the `head` variable of either `push` or `pop` or the `next` variable of `pop`, none of which are reachable from `Head`, then there must be at least n pending pop operations. Moreover, the `head` variable of each of these pending `pop` operations refers to a node in the chain.

Because each node is not reachable from `Head`, each node must have been removed from the stack during a pop operation. If the `next` field of any node but the last in the chain had been set to `null`, then the chain would be less than $n + 1$ nodes long. Thus, there must be at least n pop operations that have removed a node from the stack, but not yet executed the assignment at P9. These are all pending pop operations.

We apply the above observation to each of the procedures in the stack implementation. The only local pointer variable of the `pop` procedure is `head`. If there are $n + 1$ nodes reachable from the `head` variable of some process p executing a `pop` operation, but not reachable from `Head`, then there are n pending pop operations. One of these belongs to p. Thus if $n + 1 > 2$, there is at least one other pending `pop` operation whose `head` variable refers to one of the $n + 1$ nodes reachable from p's `head` variable.

The `push` procedure has two local pointer variables: `head` and `nd` (the new node). These variables can be in one of three possible situations. We describe each in turn:

1. `nd->next = null`. This is the situation during the first iteration through the loop

in the `push` procedure, before the assignment at H5. If there are $n + 1 > 2$ nodes reachable from the `head` variable of the `push` procedure, then there are n pending `pop` operations with `head` variables referring to one of the nodes in the chain.

2. `nd->next = head`. This is the situation after the assignment of H5 in Figure 5.2. The argument for this situation is just like that for the previous.

3. `nd->next =` *h* where *h* is the pointer value of the `head` variable the last time that the loop was executed. This is the situation during the second and subsequent iteration through the loop, prior to the assignment at H5. In this situation, it is possible that there are three nodes reachable from the local variables of the `push` procedure that are not reachable from `Head` or any other local variables (i.e., `nd`, `head` and *h*). However, if there are $n + 1 > 1$ nodes reachable from *h* that are not reachable from `Head`, then there are $n > 0$ pending pop operations with `head` variables pointing into that chain. Likewise for the `head` variable of the `push` operation.

Thus, if the number of nodes reachable from the local variables of the `push` procedure exceeds three, there is always some other pending operation with its `head` variable referring to one of those nodes.

5.5.2 Optimising Transformed Code

So far, we have stipulated that the only way to copy a local expression into a shared variable is to use an invocation of `RC_Store`. As discussed in the next section, `RC_Store` is a "heavyweight" procedure that uses a loop around a CAS operation to atomically modify the shared reference. There are cases when this heavyweight approach is unnecessary. It may be that the location being stored to may only be modified by one process. This is the case when the location is within a region of memory that has been newly allocated by a process, and not yet exposed to other processes. The stores at lines TH3 and TH6 of Figure 5.4, where a `push` operation initialises the newly allocated node are examples. In this case, the newly allocated node has not yet been pushed onto the stack, and is not yet visible to any other process. Another example is the store at line P9 of Figure 5.5, where `null` is written into the `next` field of the node removed from the stack by a `pop` operation. Here, the only process that can modify the `next` field is the process that just removed the node from the stack.

We provide a way to exploit these opportunities for optimisation by extending the interface with a procedure `RC_UnsafeStore` that has the same behaviour as `RC_Store` in situations where only one process can modify the shared reference. However, `RC_UnsafeStore` uses a simple write (without a loop) to modify the shared reference and can be expected to be substantially faster than `RC_Store`. A simple way to optimise transformed code is to use `RC_UnsafeStore` where-ever it is safe to do so.

```
RC_UnsafeStore(RC_Ref*r, RC_Obj*o)
```

```
struct RC_Ref {
  RC_Obj *ref;
  int holdC;
}

struct RC_Status {
  int sharedC;
  int localC;
}

struct RC_Obj {
  RC_Status status;
  T v;
}
```

Figure 5.6:

`RC_UnsafeStore(r,o)` stores `o` into `r->ref`. If the value of `r->ref` before the operation is not `null`, then the reference count associated with `*(r->ref)` is decremented. If `o != null` the reference count associated with `*o` is incremented.

5.6 The Implementation

We now describe our LFRC implementation. We begin with an overview before proceeding to a detailed description of the implementation. As before, fix some object type `T`, representing the type of the application level objects that are to be collected.

Overview

LFRC makes use of three types: `RC_Ref`, `RC_Status` and `RC_Obj`, which are presented in Figure 5.6. The type `RC_Obj` is the type of objects that have an associated reference count. Each `RC_Obj` has a field holding an object of type `T` and an `RC_Status` field. The `RC_Status` field contains two counters `sharedC` (which we call the *shared count*) and `localC` (the *local count*), the purpose of which is explained below. The type `RC_Ref` represents shared references to `RC_Obj` objects. It contains a pointer `ref` of type `RC_Obj *`, and a counter `holdC` (the *hold count*).

Our LFRC technique depends on both fields of the `RC_Ref` structure being atomically modifiable by a CAS operation. For example, if pointers are 32 bits on a given system, and we represent an integer using 32 bits, then we need a 64-bit CAS operation to use LFRC on that system. For this reason, our LFRC technique is not 64-bit clean. However, as we discuss

below, incrementing the `holdC` field of a `RC_Ref` structure has the effect of incrementing the reference count of the associated object. Because of this, it is possible to increment the reference count of an object without dereferencing a pointer to the object, and without the attendant risk that the object has already been deallocated.

We explain the LFRC implementation by first describing invariants of the heap and processes' stacks in reachable states of the LFRC algorithm. We begin by considering properties that are guaranteed to hold in reachable states in which there are no pending LFRC operations. We call such states *quiescent states*.[4] Let $AllocRefs$ be the set of locations allocated on the heap that contain `RC_Ref` structures; and let $StackObjs$ be the set of locations on the stack that contain allocated pointers of type `RC_Obj *`. For each non-null `RC_Obj *o`, let $S(\mathtt{o})$ be the set of shared locations that hold references to `*o`:

$$S(\mathtt{o}) = \{\mathtt{RC\_Ref\ *r} \in AllocRefs \mid \mathtt{r->ref = o}\}$$

LFRC guarantees that in reachable quiescent states, for each non-null `RC_Obj *o` we have:

$$|\, S(\mathtt{o})\,| = \mathtt{o->status.sharedC} \qquad \text{(i)}$$

For each object `o`, we call the quantity `o->status.sharedC` the *shared-reference count* of `o`. As we discuss below, an LFRC operation that creates a shared reference to an object `RC_Obj*o` increments `o->status.sharedC`, and an operation that destroys a shared reference decrements `o->status.sharedC`.

We define for each `RC_Obj *o` the set of local locations that contain references to `*o`, $L(\mathtt{o})$.

$$L(\mathtt{o}) = \{\mathtt{RC\_Obj\ **l} \in StackObjs \mid \mathtt{*l = o}\}$$

Define the *local-reference count* of `o`, written $lrc_{\mathtt{o}}$, as follows:

$$lrc_{\mathtt{o}} = \mathtt{o->status.localC} + \sum_{\mathtt{r} \in S(\mathtt{o})} \mathtt{r->holdC}$$

So the local-reference count of `*o` is distributed between `o`'s status field and the `holdC` field of all the `RC_Ref` objects that refer to `*o`. LFRC guarantees that in reachable quiescent states, for each `RC_Obj *o` we have:

$$|\, L(\mathtt{o})| = lrc_{\mathtt{o}} \qquad \text{(ii)}$$

As we discuss below, an LFRC operation that creates a local reference to an object `RC_Obj *o` either increments the `holdC` value of one of the `RC_Ref` objects that refer to `*o`, or increments `o->status.localC`. An LFRC operation that destroys a local reference to an object `*o` decrements `o->status.localC`.

[4] It is important to distinguish between quiescent states of the LFRC algorithm, and quiescent states of a client algorithm that uses the LFRC functionality, such as the transformed stack of Section 5.5. In particular, a state that is quiescent for the LFRC algorithm may not be a quiescent state of the client algorithm.

In nonquiescent states (those states in which at least one LFRC operation is in progress) these invariants are broken in certain ways. The shared-reference count of each object is decremented *after* a shared reference to that object is destroyed, and the shared count of each object is incremented *before* a shared reference to that object is created. (These modifications all occur within `RC_Store` and `RC_CAS` operations.) Because of this, the shared-reference count of each object is always *greater than or equal to* the number of shared references pointing to that object. Thus, for all `RC_Obj *o`:

$$\mid S(\mathtt{o}) \mid \leq \texttt{o->status.sharedC} \qquad \text{(iii)}$$

Similarly, local-reference count values are decremented only after associated references have been destroyed, and are incremented simultaneously with or before the creation of local references. However, it is possible for a local-reference count to underestimate the number of local references to an object. This can occur during execution of the `RC_Store` and `RC_CAS` operations. Briefly, after an `RC_Store` or `RC_CAS` operation over-writes a `RC_Ref` structure that refers to an object `*o`, the process executing the operation adds the `holdC` value of that structure to `o->status.localC`. This occurs when the process decrements the shared-reference count of `*o`. Thus, until the `holdC` has been transferred to the `localC`, $lrc_{\mathtt{o}}$ may underestimate $\mid L(\mathtt{o}) \mid$. However, because the shared-reference count of `*o` is not decremented until the point at which this transferral completes, underestimation of the local-reference count of an object can only occur when the shared-reference count is *strictly greater than* the actual number of shared references to that object. Thus, for all `RC_Obj *o`:

$$\text{if } lrc_{\mathtt{o}} < \mid L(\mathtt{o}) \mid \text{ then } \mid S(\mathtt{o}) \mid < \texttt{o->status.sharedC} \qquad \text{(iv)}$$

These properties imply that when the `sharedC` and `localC` of an object are both zero, there are no references to that object and the object may be deleted. This is because when the `sharedC` is zero, the `localC` is at least as great as the number of local references to the object. Assume for some `RC_Obj *o` that `o->status.sharedC` $= 0$. By (iii) above, $S(\mathtt{o}) = \varnothing$ and by the definition of lrc, $lrc_{\mathtt{o}} =$ `o->status.localC`. These two facts and Invariant (iv) imply that if `o->status.localC` $= 0$, then $\mid L(\mathtt{o}) \mid \leq lrc_{\mathtt{o}} = 0$ and thus, $\mid L(\mathtt{o}) \mid = \varnothing$. So when `o->status.sharedC` $= 0$ and `o->status.localC` $= 0$ the object `o` may be freed.

Reference counting typically associates each object with a single reference count, and we explained our LFRC interface in terms of a single count. We now describe how our shared- and local-reference counts implement a single reference count for each object. We define the *abstract reference count* of `RC_Obj *o` to be the sum of $lrc_{\mathtt{o}}$ and `o->status.sharedC`. In quiescent states, we have the following identity:

$$\mid S(\mathtt{o}) \mid + \mid L(\mathtt{o}) \mid = lrc_{\mathtt{o}} + \texttt{o->status.sharedC} \qquad \text{(v)}$$

So in quiescent states the abstract reference count does in fact count the number of references to each object. Further, when both the local- and shared-reference counts of an object reach zero in nonquiescent states, there are no references to that object either in the heap or from local variables of the application. Thus, the abstract reference count is zero precisely when the object is eligible for deallocation.

```
void RC_Load(RC_Obj **o, RC_Ref *r) {
L1.   RC_Ref a;
L2.   RC_Obj *oldo := *o
L3.   do {
L4.     a := *r;
L5.     if (a.ref = null) {
L6.       *o := null;
L7.       break;
L8.     }
L9.   } while(!CAS(r, a, <a.ref, a.holdC+1>));
L10.  *o := a.ref;
L11.  RC_Destroy(oldo);
}
```

Figure 5.7: The `RC_Load` procedure.

The Implementation

We first describe the implementation of the `RC_Load` operation. `RC_Load(o, r)` loads the pointer `r->ref` into the local location `*o`, simultaneously incrementing `r->holdC` using a CAS, and so incrementing `o`'s local-reference count. This reflects the fact that a local reference to `*(r->ref)` is created by the operation. Pseudocode for the `RC_Load` procedure is presented in Figure 5.7. A process *p* executing `RC_Load` first saves the current value of `*o` in the local variable `oldo` (L2). This is so that the local-reference count of the value over-written by `RC_Load` can be decremented before the procedure exits (which is achieved by invoking `RC_Destroy` on line L11). Then *p* enters a loop (L3) in which it attempts to read the pointer `r->ref` and atomically increment `r->holdC`. Process *p* loads the current value of `r` into the local variable `a` (L4). Then *p* tests whether the value of `r->ref` was `null` (L5). If it was, *p* sets `*o` to `null` and jumps out of the loop (L7). In the case where `r->ref` was not `null` when *p* executed line L4, *p* uses a CAS to increment `r->holdC` while ensuring that `r->ref` is the same as it was when *p* executed line L4. After successfully executing the CAS, *p* completes the loop and decrements the local reference count of `*oldo` (L11).

We now describe an important procedure `UpdateStatus` that is not part of the LFRC interface, but is used throughout the LFRC implementation. `UpdateStatus` is used whenever the `status` component of an object needs to be modified. `UpdateStatus(o,scD,lcD)` adds `scD` to the shared count of `*o` and adds `lcD` to to the local count of `*o`. Figure 5.8 presents pseudocode for the `UpdateStatus` procedure.

A process *p* executing `UpdateStatus` first checks whether `o` is `null`. If it is, *p* returns. Otherwise, *p* enters a loop in which it repeatedly loads `o->status` (U4), constructs a new status value by respectively adding `scDelta` and `lcDelta` to the `sharedC` and

```
void UpdateStatus(RC_Obj *o, int scDelta, int lcDelta) {
U1. RC_Status s, new_s;
U2. if (o = null) return;
U3. do {
U4.   s := o->status;
U5.   new_s := <s.sharedC+scDelta, s.localC+lcDelta>;
U6. } while(!CAS(&(o->status), s, new_s));
U7. if (new_s = <0,0>)
U8.   DeleteObject(o);
}
```

Figure 5.8: The `UpdateStatus` procedure.

```
void RC_Store(RC_Ref*r, RC_Obj *o) {
S1. RC_Ref a;
S2. UpdateStatus(o,1,0);
S3. do {
S4.   a := *r;
S5. } while(!CAS(r, a, <o,0>));
S6. UpdateStatus(a.ref, -1, a.holdC); }
```

Figure 5.9: The `RC_Store` procedure.

`localC` components of the values just loaded (U5), and then attempts to use CAS to update `o->status` to this new status value. Finally, *p* tests whether the new status value indicates that no references to the object exist (U7). If this is so, *p* invokes `DeleteObject` on the object. `DeleteObject` is not part of the LFRC interface, but its function is to free the memory pointed to by `o`. (In some circumstances, it will delete objects that `*o` contains references to. `DeleteObject` is discussed fully below.)

We turn now to `RC_Store`. `RC_Store(r,o)` sets `r->ref` to `o` and modifies reference counts appropriately by decrementing the shared-reference count of the previous value of `r->ref` and incrementing that of `*o`. Pseudocode for the `RC_Store` procedure is presented in Figure 5.9. Process *p* executing `RC_Store(r, o)` first increments the shared-reference count of `*o` by calling `UpdateStatus` (S2). (In the case where `o = null` the call to `UpdateStatus` has no effect.) Then, *p* enters a loop (S3) in which it loads `*r` into the local variable `a` (S4) and uses a CAS to set `r->ref` to the new value `o` and `r->holdC` to zero (S5). To see why `r->holdC` is set to zero, consider Invariant (iv) of Section 5.6. The size of $L(\mathtt{o})$ is unchanged, so to maintain the relationship between $\mid L(\mathtt{o}) \mid$ and $lrc_{\mathtt{o}}$ stipulated by Invariant (iv), we need to ensure that $lrc_{\mathtt{o}}$ remains unchanged. Thus, we set `r->holdC` to zero.

Note that once the S3-S5 loop has completed, `a.ref` is the value of `r->ref` when

```
bool RC_CAS(RC_Ref *r, RC_Obj *old, RC_Obj *new) {
C1. RC_Ref a;
C2. UpdateStatus(new,1,0);
C3. do {
C4.   a := *r;
C5.   if (a.ref != old) {
C6.     UpdateStatus(new,-1,0);
C7.     return false;
C8.   }
C9. } while(!CAS(r, a, <new, 0>));
C10.UpdateStatus(a.ref, -1, a.holdC);
C11.return true;
}
```

Figure 5.10: The RC_CAS procedure.

the CAS successfully executed. After the loop has completed, the local-reference count of `a.ref` will underestimate the number of local references if `a.holdC` is nonzero. However, with the successful CAS, *p* destroyed a shared reference to `*o`, but has not yet decremented `o->status.sharedC`. Thus, both properties (iii) and (iv) of Section 5.6 are preserved. We fix-up the local and shared counts of `a.ref` by invoking `UpdateStatus` to decrement the shared count of `a.ref` and add `a.holdC` to the local count of `a.ref` (S6).

We now describe the implementation of RC_CAS. RC_CAS implements the semantics of the CAS operation on shared `RC_Ref` targets, while managing reference counts. Pseudocode for the RC_CAS procedure is presented in Figure 5.10. RC_CAS works in a similar fashion to `RC_Store`, the only added complexity being that the update must be conditional. A process *p* executing `RC_CAS(r, old, new)` first increments the shared reference count of `*new` by calling `UpdateStatus(o,1,0)` (C2). (As before, in the case where `new = null` the call to `UpdateStatus` will have no effect.) This extra reference count is required in the case where this execution of RC_CAS succeeds and creates another shared reference to `*new`. If this execution of RC_CAS fails, the reference count must be decremented (which occurs at line C6).

Next *p* enters a loop (C3) in which it loads `*r` into the local variable `a` (C4) and checks whether `a.ref` is equal to the expected value `old` (C5). If is is not, the RC_CAS fails. In this case *p* decrements the shared reference count of `*o` (C6) and returns `false` indicating failure. In the case where `a.ref = old`, *p* attempts to update `r.ref` to `new` and `r.holdC` to zero using a CAS (C9). The `holdC` value is set to zero for the same reason as it is in the `RC_Store`. As with `RC_Store`, the shared and local reference counts of `a.ref` must be updated. This is achieved by calling `UpdateStatus(a.ref,-1,a.holdC)` (C10). Finally, *p* returns `true`, indicating success.

We now describe the implementation of `RC_Destroy`, which decrements the local ref-

```
void RC_Destroy(RC_Obj *o) {
D1. UpdateStatus(o,0,-1);
}
```

Figure 5.11: The RC_Destroy procedure.

```
void RC_Alloc(RC_Obj **o) {
A1. if (*o != null)
A2.   RC_Destroy(*o);
A4. *o := malloc(sizeof(RC_Obj<T>));
A5. (*o)->status := <0, 1>;
}
```

Figure 5.12: The RC_Alloc procedure.

erence count of a given object. Pseudocode for RC_Destroy is presented in Figure 5.11. A process executing RC_Destroy(o) calls UpdateStatus (D1) to decrement o->status.localC. Note that it is possible for the localC of an object to fall below zero. This can occur, for example, when a process creates a local reference to an object by calling RC_Load, incrementing the holdC of a shared reference to the object, and then calls RC_Destroy without the shared reference being overwritten. However, localC only falls below zero when sharedC is greater than zero.

We now describe the implementation of RC_Alloc, which allocates RC_Obj objects. Pseudocode for the RC_Alloc procedure is presented in Figure 5.12. A process *p* executing RC_Alloc first tests whether the location into which the new reference will be stored contains a non-null pointer (A1). If so, it decrements the local-reference count of that pointer by calling RC_Destroy. Then *p* allocates a new RC_Obj object (expressed here by malloc(sizeof(RC_Obj<T>)) into the given location (A4). Next *p* sets the status field of the new object to <0, 1> (A5). This reflects the fact that there are no shared references to the new object (a consequence of the semantics of malloc) and that the operation creates one local reference (in the location o).

We now describe the implementation of DeleteObject, which frees the memory

```
void DeleteObject(RC_Obj *o) {
O1. for each RC_Ref field f of *o do
O2.   UpdateStatus(&o->f, -1, (o->f).holdC);
O3. free(o);
}
```

Figure 5.13: The DeleteObject procedure.

```
void RC_SharedCopy(RC_Ref*r, RC_Ref*s) {
S1. RC_Obj *x := null;
S2. RC_Load(&x, s);
S3. RC_Store(r, x);
S4. RC_Destroy(x);
}
```

Figure 5.14: The `RC_SharedCopy` procedure.

```
void RC_LocalCopy(RC_Obj **o, RC_Obj *p) {
L1. RC_Obj a = *o;;
L2. UpdateStatus(p,0,1);
L3. *o := p;
L2. RC_Destroy(a);
}
```

Figure 5.15: The `RC_LocalCopy` procedure.

associated with `RC_Obj` objects. Pseudocode for the `DeleteObject` procedure is presented in Figure 5.13. As well as releasing memory associated with an `RC_Obj` object, `DeleteObject` must modify reference counts associated with objects that are referenced by fields of the object being deleted. During an execution of `DeleteObject(o)`, all the fields of `*o` of type `RC_Ref` are deallocated. This means that the locations containing those references are removed from the set of allocated locations. `DeleteObject` only functions correctly in cases where no other operation can modify the fields of the object being deleted. This is acceptable because `DeleteObject` is only invoked when some (unique) process has determined that no references to the object exist.

We now describe the `RC_SharedCopy` operation, which copies a reference from one shared location to another. Pseudocode is presented in Figure 5.14. A process executing `RC_SharedCopy` first creates a local variable and initialises it to `null` (S1), and then loads the pointer from the location referenced by its first argument into that local variable (S2). Then it stores that pointer into the location referenced by its second argument (S3). The new copy of the pointer created during these operations must be destroyed, so that the associated object can be reclaimed. This is achieved by calling `RC_Destroy` (S4), which decrements the local-reference count of the object. The `RC_SharedCopy` operation does not guarantee that the copy is atomic.

The `RC_LocalCopy(o, p)` operation (Figure 5.15) copies `p` into the location `*o`, overwriting the previous value at that location. Therefore, the operation increments the local-reference count of `*p` (L1), and decrements that of `**o` (L2). Between these modifications, `RC_LocalCopy` simply assigns `p` to `*o`, thus effecting the copy.

The `RC_Pass` operation (Figure 5.16) first uses `UpdateStatus` to increment the local-

```
RC_Obj * RC_Pass(RC_Obj *o) {

P1. UpdateStatus(o, 0, 1);
P2. return o;
}
```

Figure 5.16: The RC_Pass procedure.

```
void RC_UnsafeStore(RC_Ref *r, RC_Obj *o) {
U1. UpdateStatus(o,1,0);
U2. RC_Ref a := *r;
U3. *r := <o,0>;
U4. UpdateStatus(a.ref, -1, a.holdC);
}
```

Figure 5.17: The RC_UnsafeStore procedure.

reference count of the object being passed as an argument, and then returns a pointer to that object.

The RC_UnsafeStore(r, o) operation (Figure 5.17) efficiently copies o into the shared location referenced by r. The operation first increments the shared-reference count of o (U1). Next, the operation records the current value of *r, so that the reference counts of r->ref may be updated after the location has been modified. Then, RC_UnsafeStore simply assigns the pointer being stored into the shared location, with a zero holdC (reflecting the fact that no new local references are being created). This use of a write to update the location is the source of the procedure's efficiency. The loop and CAS of RC_Store is avoided. Finally, RC_UnsafeStore updates the status of the object whose pointer was just overwritten.

5.7 Related Work

Significant work has been done on developing techniques for reclaiming memory from nonblocking data structures. We first review the techniques based on reference counting, before describing other approaches.

Valois proposed a lock-free reference counting technique[5] and applied it to nonblocking implementations of a queue [Val94] and a linked-list [Val95]. In his technique, each object has a single reference-count field that counts the number of (local and shared) references to that object. When a process executes a read operation, it first loads the pointer at the location

[5] As originally presented, the technique has two bugs. These bugs are explained and corrected in [MS95]. The essentials of the technique remain the same.

being read and then increments the associated object's reference count. Because the memory may be deallocated between the initial read and the increment of the reference count, Valois' technique can only be used in an environment where it is legal to read and modify the reference count of unallocated memory. This is in contrast to our LFRC technique, which requires no such guarantee. [Rei04] describes a reference-counting technique based on LL/SC that is similar to Valois' proposal and shares the same limitation.

The authors of [MS95] report that Valois' queue implementation [Val94], which uses reference counting, suffers from out-of-memory errors, even when the queue is small relative to available memory (12 elements or less, with a free pool of 12,000 nodes). Valois' queue is weakly space-adaptive, and a process delayed during a `dequeue` operation can prevent any node added to the queue during the delay from being freed until the `dequeue` operation completes. This suggests that weakly space-adaptive algorithms must be used with care.

The authors of [DMMm01] implement lock-free reference counting using the DCAS primitive. Each object has a reference-count field that is incremented whenever a reference to the object is created. DCAS is used to solve the problem of incrementing this counter while guaranteeing that the object is not deallocated. A process *p* loads a value from a shared location as follows:

1. *p* reads the pointer stored in the location. Call this pointer `o`.
2. *p* reads the reference-count field of `*o`.
3. DCAS is used to simultaneously increment the reference count and to test that `o` is in the given location.
4. If the DCAS is successful, the pointer `o` is returned from the operation. Otherwise, *p* retries the operation.

The requirement that the system provide the DCAS primitive is the most important limitation of this technique. It implies that the solution is not pointer clean, and can only be used on the very few systems that support DCAS. A further limitation is that the environment must allow reads from deallocated memory (in step 2), and must allow DCAS to be applied to a location in deallocated memory, but only in the case that the DCAS fails (in step 3).

As well as containing the first presentation of the Treiber stack, [Tre86] presents a technique for reclaiming memory from nonblocking data structures. Each data structure is equipped with a *use count* that counts the number of operations that have been invoked, but not yet completed. Whenever this count is zero, there are no pending operations, and thus any memory that has been removed from the data structure may be freed to the system. The technique is simple and reasonably efficient, the only manipulations of shared locations being two modifications to the use count per operation (an increment at the beginning, and a decrement at the end). However, no memory can be freed until a quiescent state is reached. Thus the technique only allows the construction of weakly space-adaptive algorithms.

Reference-counting techniques provide more opportunities to free memory than Treiber's proposal. This is because, using reference counts, all memory that is not reachable from

shared references will eventually be freed, so long as no process fails. In Treiber's approach, if no quiescent state is reached, then no memory whatsoever can be freed, even if no process fails. (This happens during intervals when new operations continuously begin, before all other operations have completed.) However, note that both techniques can be used to construct weakly space-adaptive algorithms. Thus, it seems that our notion of weak space-adaptivity is not precise enough to capture some distinctions that we might want to make between memory reclamation techniques. We discuss possibilities for improvement in the conclusions to the thesis (Chapter 7).

Herlihy et al. [HLM02b, HLMM05] and Michael [Mic04] independently proposed general techniques that enable memory to be freed from nonblocking data structures. We describe the basic idea, while ignoring important subtleties in the implementations, and differences between the two approaches. Prior to accessing a block of memory, each process saves a pointer to the block in an SW/MR register, which we call a *guard*, and then checks that the pointer still exists in some other shared location. Prior to freeing memory, each process checks that no guard contains a pointer to any block about to be freed. This technique guarantees that no memory is accessed after it has been freed, so long as no attempt is made to free memory while it is referenced by a pointer in some shared location.

We expect the guard-based proposals to allow traversal of pointers significantly faster than our LFRC approach. Using the techniques of [HLM03a] or [Mic04], a memory block may be accessed after executing two reads and a write in the best case (reading a pointer, writing it to a guard, and then checking that the reference still exists). Compare this with the use of a CAS to increment a counter on every read in LFRC. CAS operations are typically much more expensive than reads and writes, and in LFRC, several CAS operations may be needed for each read, even when the pointer value being read has not changed. (This is because other processes may increment the counter.) These performance disadvantages are shared by the other reference-counting techniques, and to a lesser extent by the use-count technique of Treiber [Tre86].

The guard-based approach has been used to enable memory reclamation from the M&S queue [HLMM02, Mic04], and used in the construction of a lock-free reference counting algorithm [HLMM05]. The techniques can be used to allow memory reclamation from a very broad range of data structures. However, the resulting algorithms are not population-oblivious as originally presented. Although they can be made population-oblivious [HLM03b], the resulting solutions are still not strongly space-adaptive. In the M&S queue, in the worst case, they require space proportional to the number of processes that ever access the queue, plus space proportional to the size of the data structure. Both these drawbacks are a direct consequence of the reliance on SW/MR registers. However, techniques presented in [HLM03b] enable the technique to be used to construct weakly space-adaptive data structures. This is achieved using a counting technique (akin to reference counting) to enable the deallocation of the SW/MR registers. However, we would expect these techniques to come with a significant performance cost, compared with the original approach of [HLMM02, Mic04]. We revisit these issues in Chapter 6.

5.8 Concluding Remarks

The main result of this chapter is a lock-free reference counting technique and a transformation based on this technique that produces algorithms that recycle memory from algorithms that do not. The transformed algorithms are guaranteed to be weakly space-adaptive, and may be strongly space-adaptive.

The most important limitation of the technique is that it is not pointer clean. We address this problem in the next chapter, by presenting a pointer clean, strongly space-adaptive, population oblivious implementation of an LL/SC variable. Not only is this implementation pointer clean, but the LL and SC operations are *wide* synchronisation primitives, in the sense defined in Section 5.1.1. Thus, the LL/SC implementation can be used to obtain pointer-clean versions of algorithms that depend on wide synchronisation primitives. In particular, we show how to apply the LL/SC implementation to our LFRC technique, obtaining a general pointer clean, space-adaptive and population-oblivious memory management technique for nonblocking algorithms.

Chapter 6

A Pointer-clean LL/SC

The main result of this chapter is a novel, lock-free, strongly space-adaptive and population oblivious implementation of LL/SC variables. The implementation enables the manipulation of values of arbitrary width, while being pointer clean. That is, the size of the value over which the LL/SC variable ranges is not limited by properties of the underlying system such as the size of locations that can be atomically modified. Thus, in the terminology of Section 5.1.1 we implement a *wide* LL/SC variable. Section 6.1 discusses the definition of space-adaptivity, as applied to LL/SC variables, and Section 6.2 describes the LL/SC implementation.

The LL/SC implementation presented in this chapter is the first pointer-clean, space-adaptive, population oblivious, wide LL/SC variable. Moreover, to our knowledge it is the first published [DHLM04] nonblocking algorithm that uses dynamically-allocated memory to possess all these properties. Because the LL/SC implementation enjoys these properties, it can be used to implement the LFRC technique described in the previous chapter. This means that most extant lock-free algorithms that are not weakly space-adaptive or that depend on wide synchronisation primitives can be transformed into lock-free algorithms that do not suffer these limitations. The use of the LL/SC variable in the LFRC technique is described in Section 6.3.

Our LL/SC algorithm is somewhat complicated, and it will not be immediately clear to the reader that it satisfies its specification. Therefore, we have employed the techniques developed in Part I to verify our LL/SC implementation. Section 6.5 describes our verification of the LL/SC implementation.

Section 6.4 discusses previous LL/SC implementations, and other related work. We conclude the chapter in Section 6.6.

6.1 Space-adaptivity

In Section 5.1.2, we mentioned that the appropriate notion of *size* for an LL/SC variable should depend on the number of outstanding LL operations. There are two reasons for this.

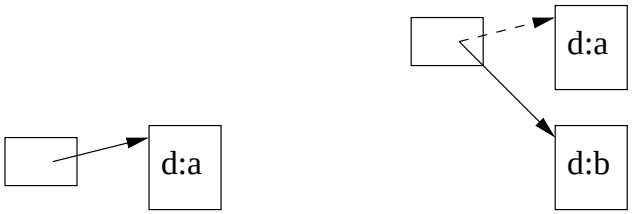


Figure 6.1: Simple LL/SC implementation. (a) A state of the implementation where the variable contains value *a*. (b) The effect of the SC operation, changing the value of the variable from *a* to *b*. The dashed line indicates the previous value of the pointer.

First, any LL/SC algorithm needs to record which pending LL operations can be matched by successful SC operations, and at least some memory must be used to store this information. Simply recording which of N processes can execute a successful SC operation requires N bits. All LL/SC implementations consume memory for this purpose. This memory is either allocated per-operation (possibly on the stack), or per-process (using single-reader/multi-writer variables).

Second, because each LL is eventually matched by an SC,[1] any memory consumed by a pending LL will be released when the matching SC executes, and thus will be released by the time the operation that invoked the LL completes. Memory consumed by each outstanding LL operation can be accounted for as consumed by the pending operation that invoked the LL. Thus, an LL/SC implementation that is (strongly or weakly) space-adaptive, but consumes memory for outstanding LL operations, preserves the (strong or weak) space-adaptivity of any data structure that uses it.

Our LL/SC implementation is f-space-adaptive, where the size function f is one plus the number of outstanding LL operations in the state. Therefore, our LL/SC algorithm consumes memory bounded by f plus a constant times the number of pending operations.

6.2 The LL/SC Implementation

A lock-free implementation of a population oblivious and pointer clean LL/SC variable is almost trivial if we assume unbounded memory. The idea, illustrated in Figure 6.1, is to use an extra level of indirection to enable operations to detect changes to the LL/SC variable. We would store values in contiguous regions of memory called *nodes*, each containing a value, and maintain a pointer to the *current* node. An LL operation would simply read the pointer to the current node and return the contents of the node it refers to. An SC operation would allocate a new node, initialise it with the value to be stored, and then use CAS to attempt

[1] In some contexts, it is desirable for a process to "abandon" an LL operation by never invoking a matching SC. Section 6.2.4 describes an `unlink` operation that provides this capability. For the purposes of the present discussion, it is simplest to assume that each LL is eventually matched by an SC.

to replace the previously current node with the new one. So long as we never reclaim and reuse any node, the CAS in each SC succeeds if and only if there is no change to the pointer between the CAS and the read in the preceding LL. Thus, the SC succeeds if and only if the CAS succeeds. This technique is well-known and used in systems that use garbage collection to provide the illusion of unbounded memory. For example, the JSR-166 library [JSR], which provides tools for building highly-concurrent and nonblocking data structures in Java, uses this technique.

Our implementation builds on this simple idea, but is complicated by the need to explicitly free and reuse nodes in order to bound memory consumption. If we reclaim (and possibly reuse) a node too soon, one of several problems can arise. First, an access to a node that has been reclaimed may cause an error, as discussed in Chapter 1. Second, an LL reading the contents of a node might in fact read part or all of a value stored by an SC that is reusing the node. Third, the CAS might succeed despite changes since the previous read because of the recycling of a node: the ABA problem.

One possible solution is to apply the LFRC technique presented in the previous chapter, by transforming the unbounded memory algorithm described above into a version that recycles storage. This would involve introducing a hold count to the location containing the pointer to the current node, and associating the node with a shared-reference count. We could then use these counters to determine when a node was no longer reachable. We would have each SC operation allocate a new node to replace the old one, and implement the LL/SC semantics in essentially the same way as we did under the assumption of unbounded memory.

The problem with this approach is that the LFRC technique from the previous chapter is not pointer-clean: the technique requires that the system provide a wide CAS operation that can atomically compare-and-swap both a pointer and the hold count. Our LL/SC algorithm uses a more complicated, but pointer-clean technique to give the effect of modifying a pointer and a hold count atomically. In Section 6.2.1, we give an overview of this technique, before moving to a detailed description of the algorithm in Section 6.2.2.

6.2.1 Overview

Rather than storing a pointer to the current node in a single location, we alternate between two locations `ptr0` and `ptr1`. One of these pointers is *current* and refers to a node containing the current value of the LL/SC variable. This node is called the *current node*. The pointer that is not current is called the *noncurrent pointer*, the location at which this pointer is stored is called the *noncurrent address*, and the node to which it refers is called the *noncurrent node*. We use a version number (stored independently of `ptr0` and `ptr1`) to indicate which of these is the current pointer: if the version number is even, then `ptr0` refers to the current node; otherwise `ptr1` does. For example, if the version number is changed from four to five, the current pointer before the change is `ptr0`, and the current pointer after the change is `ptr1`. A *hold count* is stored adjacent to the version number and we require that both

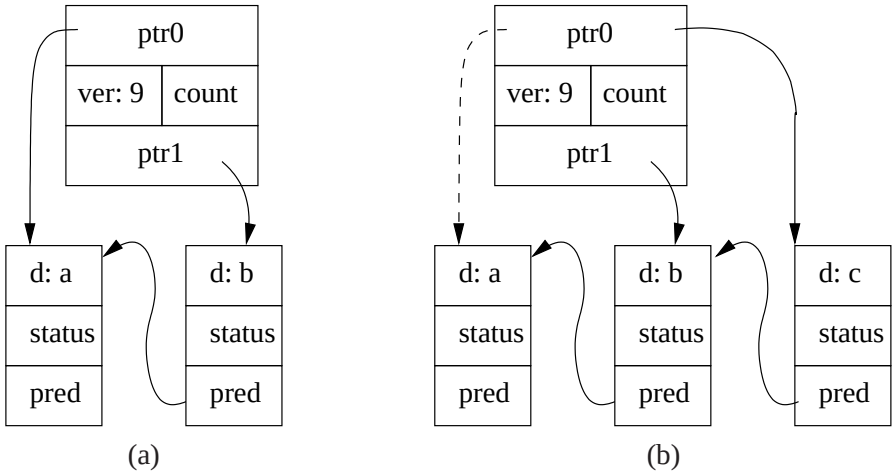


Figure 6.2: Two configurations of the LL/SC implementation. In both illustrations the version number, denoted `ver`, is odd. Therefore, in both illustrations `ptr1` is the current pointer and `ptr0` is the noncurrent pointer. The `d` field of each node contains the value stored in that node, and the `status` field is used to determine when it is safe to deallocate the node. The dashed arrow in (b) indicates the previous value of `ptr0`.

these integers be atomically modifiable by a CAS operation.[2] Because of this, the hold count can be modified by a CAS which at the same time guarantees that the version number has some expected value.

Each node *n* that has ever been the current node has a *predecessor*: the node that was current immediately before *n* last became the current node. We equip each node *n* with a `pred` field, which is guaranteed to point to *n*'s predecessor, from the point when *n* becomes the current node until *n* is deallocated.

Our algorithm ensures that the value of the current pointer is not changed in any interval during which the version number does not change. Also, our algorithm ensures that the noncurrent pointer may change (at most) once during an interval in which the version number does not change. During each interval in which the version number has a given value, the LL/SC implementation is in one of two configurations, which are illustrated in Figure 6.2:

a the noncurrent pointer refers to the current node's predecessor (Figure 6.2(a)), or

b the noncurrent pointer refers to a node that will become the current node after the next change of the version number (Figure 6.2(b)). In this case, the `pred` field of the noncurrent node will refer to the current node.

[2]For example, in a system with a 64-bit CAS operation, we can allocate 32-bits for the version number, and 32-bits for the hold count. This would allow more than four billion unmatched LL operations without risk of overflow, and another four billion successful SC operations, without risk of wraparound.

The algorithm moves from a state matching Configuration (a) to one matching (b) when the pointer in the noncurrent address is modified. This can only happen once during any interval when the version number has a given value. The version number is only incremented when the algorithm is in a state matching Configuration (b), and the point when this occurs is the linearisation point of some SC operation. When the version number is incremented, its parity changes, and the previously noncurrent pointer becomes current. Thus, we move from a state matching Configuration (b) to one matching Configuration (a).

Because the value of the current pointer does not change during any interval in which the version number does not change, an LL operation can determine the value of the current pointer using the following protocol:

a read the version number,

b read the value of the pointer that would be current, assuming that the version number has not changed from the previous step,

c check that the version number is the same as was previously observed, retrying if the version number has changed.

The linearisation point of the LL operation is the point where the check of the version number succeeds. The LL operation completes by returning the contents of the node that was current when the operation was linearised.

The check that the version number has not changed while the current pointer was read is achieved using a CAS operation that simultaneously increments the hold count. Our LL/SC implementation maintains the invariant that the hold count is the number of LL operations that have been linearised during the interval in which the current pointer had its present value. This count is used to ensure two properties about the deallocation of nodes after they have been current:

- No node *n* is deallocated until there is no outstanding LL operation that was linearised during the interval when *n* was current.

- No node *n* is deallocated until there is no outstanding LL operation such that *n* is the predecessor of the node that was current at the linearisation point of the operation.

These properties enable a process that has an outstanding LL operation to safely access both the node that was current when the LL was linearised, and that node's predecessor. Moreover, as we explain below, these properties guarantee that certain CAS operations can be executed without giving rise to the ABA problem. If an LL operation is linearised when a node *n* is current, we say that the LL operation *pins n*. Note that several LL operations can pin each *n*.

An SC operation begins by allocating a new node, initialising it with the value to be stored, and setting its `pred` field to *n*, where *n* points to the node that the matching LL operation pinned. The SC operation then attempts to write a pointer to the new node into the noncurrent address, using a CAS. The expected value in this CAS is the predecessor of

the node *n* (obtained from *n*'s `pred` field). Recall that *n*'s predecessor cannot have been deallocated since the linearisation point of the matching LL. Moreover, `ptr0` and `ptr1` are only ever over-written with values that have been newly allocated. These two facts mean that if the CAS to the noncurrent address is successful, then the version number has not changed since the matching LL, and the state immediately prior to the CAS matches Configuration (a). Afterwards, the state matches Configuration (b).

After executing the CAS to the noncurrent address, the process that executed the successful CAS, or some other process that observes that the algorithm is in a state matching Configuration (b), increments the version number, which is the linearisation point of the SC operation. When this version number is incremented, the hold count is set to zero, reflecting the fact that no LL operation has yet pinned the new node. The process that successfully increments the version number transfers the previous hold count value to a `status` field in the node that was just made noncurrent, in a similar fashion to the LFRC technique. This `status` field itself has three fields: `localC`, `nlC` and `nlP`. `localC` is used to count the remaining outstanding LL operations that pinned this node, and the previous value of the hold count is added to this field. `nlC` is a boolean flag that is set when the hold count value is transferred (i.e., after the node is "no-longer current"). For any node *n*, when `status.nlC` is true, `localC` is guaranteed to be at least as great as the number of remaining outstanding LL operations that pinned *n*. After the linearisation point of each SC, the `status.localC` count of the node pinned by the matching LL is decremented. Therefore, once `status.nlC` is true and `status.localC = 0`, the node may be deallocated once it has been determined that it is not the predecessor of any node pinned by an outstanding LL. The third field of `status`, another flag called `nlP` for "no-longer predecessor", is used to record this fact. For any node *n*, once *n*`->status.nlC` is true and *n*`->status.localC = 0`, *n*`->pred->status.nlP` is set. Finally, once both these conditions are satisfied, and *n*`->status.nlP = true`, *n* may be deallocated. We assume that the `status` field can be atomically manipulated by the CAS operation. In a 64-bit system, this would allow 2^{62} LL/SC operation pairs to complete before wrap-around occurred.

6.2.2 The Implementation

The overview of the algorithm just given ignores several important details. We now give a detailed description of the implementation. Figure 6.3 shows the types used in our implementation. Each LL/SC variable is accessed through an instance of the `Loc` structure, which has the `ptr0` and `ptr1` fields described above. The `entry` field contains the version number and hold count in a *casable record* of type `EntryTag`. A *casable record* is one that fits within the maximum word size that can be modified by a CAS instruction.

We assume that the LL/SC variable being implemented ranges over members of the type `Data`. The `d` field of the `Node` structure has this type, and contains the value stored in that node. The type `Data` may be of arbitrary width. Instances of `Node` are also equipped with the `pred` and `status` fields.

The `Status` structure has the integer field `localC`, and the flags `nlC` (standing for

```
typedef struct {
   Node *ptr0, *ptr1;
   EntryTag entry;
} Loc;

typedef struct {
   Data d;
   Node *pred;
   Status status;
} Node;

typedef struct {
   int ver;
   int count;
} EntryTag;

typedef struct {
   int localC;
   bool nlC;
   bool nlP;
} Status;
```

Figure 6.3: Data types used in the LL/SC algorithm. The `EntryTag` and `Status` types fit into 64 bits, so can be atomically accessed using CAS.

```
Macro:
  INITSTATUS (<0, false, false>)

initialise(Loc *L) {
  L->entry.ver := 0;
  L->entry.count := 0;
  L->ptr0 := p0;
  L->ptr1 := p1;
  L->ptr0->d := d0;
  L->ptr0->pred := ptr1;
  L->ptr0->status := <0, false, false>;
  L->ptr1->status := <0, true, false>;
}
```

Figure 6.4: Initial state of an LL/SC location, where `d0` is the initial value of the location and `p0` and `p1` are distinct non-`null` pointer values.

"no-longer current") and `nlP` (standing for "no-longer predecessor"). A node may be freed when both its `nlC` and `nlP` fields are true, and the `localC` has reached zero.

Figure 6.4 shows how an LL/SC location is initialised. The macro `INITSTATUS` gives the initial value for the `status` of a node. We set the version number `entry.ver` to zero, indicating that `ptr0` is the current pointer.[3] We set `ptr0` and `ptr1` to be any distinct pointer values (denoted `p0` and `p1`), and initialise the `d` field of `ptr0` (the current pointer) to be the initial value for the location (denoted `d0`). We set the `pred` field of `ptr0` to point to `ptr1` (the noncurrent pointer). At this point we have an instance of Configuration (a), where the `ptr0->pred = ptr1`. It only remains to set the fields associated with deallocation to

[3]The choice of initial version number is arbitrary, so long as we initialise the corresponding current and noncurrent nodes according to the parity of the the initial version number.

```
Macros:
  CURRENT(loc, ver) (ver%2 = 0 ?  loc->ptr0 :  loc->ptr1)

Data LL(Loc *loc) {
L1. do {
L2.  EntryTag e := loc->entry;
L3.  myver := e.ver;
L4.  mynode := CURRENT(loc, e.ver);
L5. } while (!CAS(&loc->entry, e, <e.ver, e.count+1>));
L6. return mynode->d;
}
```

Figure 6.5: Macros and the LL implementation.

the appropriate values. That is, we set `L->entry.holdCount` to 0 (which indicates that no LL operation has yet pinned the current node), and the `status` field of `*ptr0` to `<0, false, false>`. We set `ptr1->status` to `<0, true, false>`, indicating that no LL operation has pinned `ptr1`, and that `ptr1->status.localC` accurately reflects this fact.

Pseudocode for the LL operation is presented in Figure 6.5, along with a macro called `CURRENT`. `CURRENT(loc, ver)` obtains the current pointer of location `loc`, assuming that `loc->ver = ver`. Our implementation makes use of *persistent local variables*. These are variables like local variables in that they are only accessible to one process, but they retain their value across procedure invocations. In particular, each process has two persistent local variables, `mynode` and `myver`, which are set during the LL operation, and retain their values until the matching SC completes.[4]

A process executing an LL operation obtains a consistent view of the version number and current pointer by executing a loop (L1-L5) in which the process reads the `entry` field (L2), obtains the current pointer (L4), and then checks that the entry field has not changed, using a CAS (L5). If successful, the CAS increments the hold count, which guarantees that the current node will not be deallocated until after the linearisation point of the matching SC. The loop ends when the CAS succeeds. The version number and current pointer values are recorded in the persistent local variables `myver` and `mynode` (L3 and L4). Recall that the value of the current pointer does not change in any interval where the version number has not changed. Thus, once the loop completes, we know that `myver` and `mynode` were simulta-

[4]Programming languages typically do not provide persistent local variables. However, they can be emulated using *thread-local storage* as in Java [JTL], or the `pthreads` framework [But97, Section 5.4]. Alternatively, persistent local variables can be emulated by using an array or hash-table to map thread or process identifiers to variable values.

```
Macro:
  NONCURADDR(loc, ver) (ver%2 = 0 ?  &loc->ptr1 :  &loc->ptr0)

bool SC(Loc *loc, Data newd) {
S1.  Node *new_nd := alloc(Node);
S2.  new_nd->d := newd;
     new_nd->pred := mynode;
     new_nd->status := INITSTATUS;
S3.  Node *pred_nd := mynode->pred;
S4.  success := CAS(NONCURADDR(loc, myver), pred_nd, new_nd);
S5.  if (!success) free(new_nd);
S6.  while ((e := loc->entry).ver = myver) {
S7.    if (CAS(&loc->entry, e, <e.ver+1, 0>))
S8.      transfer(mynode, e.count);
     }
S9.  release(mynode);
S10. return success;
}
```

Figure 6.6: The SC implementation.

neously the version number and current node of the LL/SC variable when the successful CAS was executed. The LL operation ends by returning the value in the `d` field of the node that was determined to be current during the loop. The LL operation is linearised at the (unique) point at which *p* successfully executes the CAS at line L5.

Pseudocode for the SC operation is presented in Figure 6.6, along with a macro called `NONCURADDR`, obtains the address of the noncurrent pointer, for a given location and version number. To execute an SC operation, a process *p* allocates and initialises a new node with the value to be stored, and stores the node observed as current by the previous LL (recorded in `mynode`) in the node's `pred` field (lines S1 and S2). Then, *p* attempts to install the new node into the noncurrent pointer using CAS (line S4). The expected value for this CAS is the predecessor of the node that was current when the matching LL was linearised (obtained by reading the `pred` field of `mynode`). Recall that we cannot simply read the noncurrent pointer, because this may change while the version number has a given value.

If the CAS at S4 succeeds, then the SC operation is successful, although the operation is not linearised until the version number is next incremented. If the CAS fails, then the SC operation is unsuccessful, and the SC operation frees the newly allocated node (S5), which has not become visible to any other thread.

If the S4 CAS is successful then the state immediately before the CAS matches Configuration (a) and immediately after the CAS the state matches Configuration (b). If the S4 CAS is unsuccessful and the version number has not yet been incremented since the matching LL

was linearised, then the state already matched Configuration (b) when the CAS was executed (in which case some other SC has successfully executed an S4 CAS while the version number had its current value). Whether or not the S4 CAS is successful, the SC operation attempts to increment the version number. This is achieved using a CAS such that the version number component of the expected value in the CAS is the value of the version number when the matching LL was linearised. Thus, if the version number has already been modified at the point of the S4 CAS, this attempt to increment the version number will fail. If the increment succeeds, then the last SC operation to successfully modify the noncurrent pointer is linearised at the point where the increment occurs.

The loop test at line S6 first reads the current value of the `entry` field, and checks whether the version number has not changed since the linearisation point of the matching LL, falling out of the loop if it has.[5] The CAS at S7 attempts to increment the version number, and set the hold count to zero. If this CAS is successful (which can be true of only one SC operation for each version number), then the SC invokes the `transfer` procedure to update the `status` field of the node that was current immediately prior to the successful CAS. (We explain `transfer` shortly.) The linearisation point of an unsuccessful SC is the earliest point at which that SC observes that the version number no longer has the value that it had when the matching LL was linearised (which occurs either at S6 or S7). Once the loop has completed, `release` is invoked, which decrements the `localC` of `mynode`, indicating that there is one less outstanding LL operation that was linearised when `mynode` was current.

Figure 6.7 presents pseudocode describing the `release` and `transfer` operations, as well as another operation `setNLPred` and two macros `CLEAN` and `FREEABLE`. The expression `CLEAN(post)`, where `post` is a `Status` value, returns true iff for all nodes *n*, *n*`->status = post` implies that there are no outstanding LL operations pinning *n*. The expression `FREEABLE(post)` returns true iff `CLEAN(post) = true` and `post.nlP` has been set. In this case, it is safe to free any node *n* such that *n*`->status = post`.

The invocation `transfer(nd, cnt)` adds `cnt` to `nd->status.localC` and sets `nd->status.nlC` to true. This is achieved using a loop in which the procedure reads the current value of `nd->status` (T2), constructs the appropriate new value (T3), and attempts a CAS to set `nd->status` to the new value.

The `release` procedure is called when an SC operation passes its linearisation point and therefore needs to indicate that the number of outstanding LL operations that have pinned the node has fallen by one. The invocation `release(nd)` first copies `nd->pred` into a local variable `pred_nd` (R1), and then uses a loop and CAS to decrement `nd->status.localC` (R2-R5). After this decrement has been completed, the procedure checks whether the resulting `status` value is `CLEAN`, in which case the procedure sets the `nlP` flag of `pred_nd->status` using `setNLPred` (explained below). The `release` procedure then tests whether the new status value is `FREEABLE`, and frees the node if the test succeeds. It is necessary to read `nd->pred` (and remember the value in a local variable) prior to decrementing

[5] The value of an assignment `x := exp` is the value of `x` immediately after the assignment.

```
Macros:
  CLEAN(post) (post.count = 0 && post.nlC)
  FREEABLE(post) (CLEAN(post) && post.nlP)

void transfer(Node *nd, int cnt) {
T1. do {
T2.   Statuspre := nd->status;
T3.   Statuspost := <pre.localC+cnt, true, pre.nlP>;
T4. } while (!CAS(&nd->status, pre, post));
}

void release(Node *nd) {
R1. Node *pred_nd := nd->pred;
R2  do {
R3.   Status pre := nd->status;
R4.   Status post := <pre.localC-1, pre.nlC, pre.nlP>;
R5.   } while (!CAS(&nd->status, pre, post));
R6. if (CLEAN(post)) setNLPred(pred_nd);
R7. if (FREEABLE(post)) free(nd);
}

void setNLPred(Node *pred_nd) {
P1. do {
P2.   Status pre := pred_nd->status;
P3.   Status post := <pre.localC, pre.nlC, true>;
P4. } while (!CAS(&pred_nd->status, pre, post));
P5. if (FREEABLE(post)) free(pred_nd);
}
```

Figure 6.7: Helper procedures for the LL/SC implementation.

`nd->status.localC`, rather than afterwards, because after the decrement, some other process may observe `nd`'s `status` value becoming `FREEABLE`, and thus free the node.

`setNLPred(pred_nd)` uses a loop and CAS to set the `status.nlP` flag of `pred_nd` (P1-P4), and then tests whether the resulting status value is `FREEABLE`, freeing the node if the test succeeds (P5).

This concludes our description of the basic algorithm. We discuss certain optimisations and extensions in Section 6.2.4. We note here that it is straightforward to generalise this algorithm to several LL/SC variables. The persistent local variables `myver` and `mynode` must be managed on a per-LL/SC variable basis. This would be achieved by equipping each process with a map from LL/SC variables (represented as pointers to `Loc` objects) to `myver`/`mynode` pairs. Each LL operation would allocate a structure with space for a pointer to a node and a version number, and store `myver` and `mynode` in that structure using

the address of the `Loc` structure as the key. Each SC operation would get the appropriate values of `mynode` and `myver` from the map, and after the SC completed, deallocate the pointer/version-number pair.[6]

6.2.3 Space-adaptivity

We now state an invariant of the LL/SC algorithm that guarantees strong space-adaptivity. In every reachable state, every node *n* is in one of the following states:

1. *n* is free.
2. Some process *p* has allocated *n* during an SC operation (at line S1), and either *p* has not completed line S4 or `success = false` and *p* has completed line S4 but not S5.
3. *n* is the noncurrent node, but *n* is not the predecessor of the current node. (At this point, the current pointer has been set to *n*, but the SC operation that did so has not yet been linearised.)
4. *n* is the current node.
5. *n* is the predecessor of the current node.
6. Some process *p* has pinned *n*, but has not completed the invocation of `release(`*n*`)`.
7. Some process *p* has pinned the node *m* such that *m*`->pred =` *n* (i.e., *n* is the predecessor of *m*), but *p* has not completed the invocation of `release(`*m*`)`.

Note that these states are not mutually exclusive. For example, a node may be in States 4 and 6 simultaneously. The proof of this invariant is a straightforward induction on the executions of the algorithm. Initially, there are only two nodes allocated, `ptr0` and `ptr1`, which respectively satisfy States 3 and 4. All other nodes are free. To show that the invariant is preserved by each step of the algorithm, we argue for each step of the algorithm and each node *n*, if *n* is in one of the seven states immediately before the step, then there is some state containing *n* immediately after the step.

The following property, which we refer to as the *clean property* is important to this argument:

> For every node *n*, if *n* has been the current node since it was last allocated, and `CLEAN(`*n*`->status)` is false, then either *n* is the current node, or there is some process *p* that pinned *n* during *p*'s most recent LL operation, and *p* has not yet completed the invocation `release(`*n*`)`.

[6] In Section 6.2.4, we discuss what would happen if an SC were invoked without a prior matching LL.

This claim is justified as follows. If `CLEAN(`*n*`->status)` is false, either *n*`->status.nlC` has not been set, or *n*`->status.localC` is nonzero. In the first case, either *n* is the current node, or there is some process *p* executing the `transfer` procedure such that *p* pinned *n* during *p*'s most recent LL and has yet to complete `release(`*n*`)`. For the second case, recall that when *n*`->status.nlC` is true, *n*`->status.localC` counts the number of processes that pinned *n* but have not completed `release(`*n*`)`.

Fix a node *n*. Below, we consider all the steps during which *n* may "leave" one of the states, in the sense that *n* is in one of the states immediately before the step, but not in that state immediately after the step. We show for each such case that *n* is in one of the seven states enumerated above after the step. This is sufficient to prove the invariant.

1. The only step during which *n* can leave State 1 is by execution of line S1, if *n* is returned from the allocation. Afterwards, *n* is in State 2.

2. *n* can only leave State 2 by a successful CAS operation at S4 (if *n* is the "new" value of the CAS), or the completion of line S5 (if *n* is the argument to `free`). In the first case, *n* is in State 3 after the CAS. In the second case, *n* is free after the deallocation.

3. *n* can only leave State 3 by a successful CAS at S7, which increments the version number. Afterwards, *n* is in State 4.

4. The only step during which *n* can leave State 4 is again by a successful CAS at S7. Afterwards, *n* is in State 5.

5. *n* can only leave State 5 by a successful CAS at S4, overwriting *n* in the noncurrent address. When this CAS is executed, *n* is the predecessor of the current node. The process that executes this CAS pinned the current node during the matching LL. Therefore, *n* is in State 7, both before and after the CAS.

6. *n* can only leave State 6 when the last process *p* that pinned *n* completes its `release` operation. If *p* makes *n*'s `status` field `FREEABLE` during the operation, then *p* frees *n* before *p* completes the `release` procedure, placing *n* in State 1. Otherwise, because (by the clean property) *n*'s `status` field is `CLEAN` after *p* decrements *n*`->localC`, it must be that *n*`->status.nlP` is not set when *p* decrements *n*`->localC`. Therefore, either *n* is the predecessor of the current node, or there is some node *m* such that *m*`->pred =` *n* and some process *q* that pinned *m* but has not completed the `release` operation. In the first case, *n* is in State 4 after the `release` operation. In the second case, *n* is in State 7 after the `release` operation. (In fact, *n* is in either State 4 or State 7 both before and after the completion of the operation.)

7. *n* can only leave State 7 when the last process *p* that pinned the node *m*, such that *m*`->pred =` *n*, completes the `release` operation. In this case *p* makes *m*'s `status` field `CLEAN` (by the clean property), and thus invokes `setNLPred` on *n*. If this makes *n*'s `status` field `FREEABLE`, then *p* frees *n*, sending it to State 1. Otherwise, *n*'s

`status` field is not `CLEAN` after the `setNLPred` operation, and therefore, *n* is in State 6 after the operation.

Thus, for each process *p* that has executed an LL but not completed the matching SC, there are at most three nodes (other than the current node and its predecessor) that are not free: the node pinned by the LL, that node's predecessor, and possibly a newly allocated node. Therefore, the space consumed by an LL/SC variable with *k* outstanding operations is bounded by a multiple of $3k + 2$ (the two extra nodes being the current and noncurrent nodes). Thus, the LL/SC algorithm is strongly space-adaptive.

The space used by *V* LL/SC variables in a state with *k* outstanding LL operations is in $O(V + k)$. Furthermore, the space used by *V* LL/SC variables in a system with *N* processes is bounded by $O(V + N)$.

6.2.4 Optimisations and Extensions

Our LL/SC implementation can be made more efficient by observing that if `FREEABLE(post)` holds before the CAS on line R5 or line P4, then the CAS does not need to be executed; `mynode` can simply be freed because there are no processes that still have to `release` this node. Similarly, a process that calls `transfer` at line S8 will always subsequently call `release` at line S9. Therefore, we can combine the effect of the two CASes in those two procedures into a single CAS.

It is easy to extend our implementation to provide a method for "validating" the previous LL, that is, determining whether its future matching SC can still succeed. More precisely, the `validate` operation returns `true` if and only if no SC operation has completed successfully since the most recent LL of the process invoking `validate`. The implementation of the `validate` operation simply determines whether the version number has changed since the linearisation point of the earlier LL, returning `true` if no change has occurred, and `false` otherwise.

Hitherto, we have required every LL operation to be matched by an SC. There are applications in which it is desirable for a process to simply abandon an LL operation, without calling a matching SC. So it is desirable to provide a way to indicate that no future SC will be invoked, after an LL. If a process decides not to invoke a matching SC operation for a previous LL operation, it must instead invoke an `unlink` operation. The purpose of the `unlink` operation is to allow the LL/SC variable to free resources associated with any earlier unmatched LL operation of the same process. The only semantic effect of a process *p* executing `unlink` on a given LL/SC variable is to render the effect of any future SC by *p* undefined, until *p* executes another LL operation. The `unlink` operation can be implemented simply by invoking `release`, which indicates that the node which was pinned by the earlier SC can be deallocated. Note that `unlink` would not be needed in an implementation that did not allocate memory resources. It exists so that a process can indicate that memory resources associated with an earlier LL may be released.

So far we have not defined the effect of a process invoking an SC on a location without having invoked an earlier matching LL. In the case of our LL/SC implementation, an SC

without a matching LL could access memory that has been deallocated, or could successfully change the value of stored in the LL/SC variable. However, it is straightforward to modify our algorithm so that an unmatched SC operation is guaranteed to fail and return `false` without accessing deallocated memory. We introduce another persistent local boolean variable, which we call `matched`, that we use as follows:

- Each LL operation sets `matched` to `true`.
- Prior to executing the procedure defined in Figure 6.6, each SC operation checks `matched`. If it is `false`, then the operation simply returns `false`. If `matched` is `true`, then the `SC` operation continues as normal. After the SC operation completes the code in Figure 6.6, it sets `matched` to `false`.

To support several LL/SC variables in the application, we would keep the `matched` variable along with `mynode` and `myver` in the map from locations to persistent local variables.

6.3 Pointer-clean Lock-free Reference Counting

The LL/SC implementation just described is population-oblivious, pointer clean, space-adaptive and enables the manipulation of data values of arbitrary size (it is a *wide* synchronisation primitive). Because of these properties, it can be used in the implementation of the lock-free memory management technique of the previous chapter to overcome the problem that our LFRC technique is not pointer clean. This yields a general, population oblivious and pointer-clean, lock-free memory management technique.

We transform certain loops in our implementation that read a `RC_Ref` object and later use a CAS to conditionally update the `ref` to a new value. Within these loops, we replace the read with an LL operation and the CAS with an SC.[7] To make this work, we must change the type used to represent shared references. Therefore, we redefine the `RC_Ref` type to be the type of LL/SC variables that range over pointer/hold-count pairs. We declare a type `RefPair`,

```
typedef struct {
  RC_Obj *ref;
  int holdC;
}
```

This type is called `RC_Ref` in Chapter 5. We now redefine `RC_Ref` to be the type of `Loc` structures defined in Figure 6.3, where the `Data` type is identified with `RefPair`.

Figure 6.8 presents the implementation of `RC_Load` using LL/SC. The implementation is just like that in Figure 5.7 on page 154 except for two differences. On line L4, we use an LL operation to read the pointer and hold count contained at the location `r`. On line L9, we

[7] It is frequently straightforward to transform a nonblocking algorithm that depends on CAS into one that depends LL/SC operations, using this approach.

```
void RC_Load(RC_Obj **o, RC_Ref *r) {
L1.   RefPair a;
L2.   RC_Obj *oldo := *o
L3.   do {
L4.     a := LL(r);
L5.     if (a.ref = null) {
L6.       *o := null;
L7.       break;
L8.     }
L9.   } while(!SC(r, <a.ref, a.holdC+1>));
L10.  *o := a.ref;
L11.  RC_Destroy(oldo);
}
```

Figure 6.8: The `RC_Load` procedure, modified to use LL/SC.

use an SC operation to attempt to increment the hold count at this location, retrying if the SC fails.

The `RC_CAS` and `RC_Store` operations must also modify `RC_Ref` objects in shared location. Both use a pattern similar to `RC_Load`: each operation reads the current value of a `RC_Ref`, and later executes a CAS to modify the `RC_Ref`, repeating the read and CAS until the operation is successful. We replace each such read with an LL operation, and each such CAS with an SC. The resulting implementations are presented in Figures 6.9 and 6.9. All other LFRC operations (including `UpdateStatus`) are implemented as in Section 5.6.

Unfortunately, this approach requires (at least) one LL/SC pair for every `RC_Load`, `RC_CAS` and `RC_Store`, which implies the use of multiple CAS operations, as well as an allocation. This is likely to increase the latency of operations substantially, relative to the original implementation presented in Chapter 5.

6.4 Related Work

Moir [Moi97] presents a simple and direct wait-free LL/SC implementation that uses CAS, based on version numbering. The algorithm is lock-free and uses only $O(V+k)$ space (where V is the number of variables and k is the number of outstanding LL operations). Except when the variable is initialised (when space must be allocated for the variable's current value), Apart from the memory containing the version number and the variable's value, all memory allocated remains accessible to only one thread. Unfortunately, the algorithm is not pointer clean.

The only previous pointer clean, CAS-based implementation of LL/SC is due to Jayanti and Petrovic [JP03]. While their implementation is wait-free, it requires $O(VN)$ space (where N is the number of processes that can access the LL/SC variables); ours uses only $O(V+N)$

```
bool RC_CAS(RC_Ref *r, RC_Obj *old, RC_Obj *new) {
C1. RefPair a;
C2. UpdateStatus(new,1,0);
C3. do {
C4.   a := LL(r);
C5.   if (a.ref != old) {
C6.     UpdateStatus(new,-1,0);
C7.     return false;
C8.   }
C9. } while(!SC(r, <new, 0>));
C10.UpdateStatus(a.ref, -1, a.holdC);
C11.return true;
}
```

Figure 6.9: The RC_CAS procedure, modified to use LL/SC.

```
void RC_Store(RC_Ref*r, RC_Obj *o) {
S1. RefPair a;
S2. UpdateStatus(o,1,0);
S3. do {
S4.   a := LL(r);
S5. } while(!SC(r, <o,0>));
S6. UpdateStatus(a.ref, -1, a.holdC); }
```

Figure 6.10: The RC_Store procedure, modified to use LL/SC.

space in the worst case. Furthermore, the implementation in [JP03] is not population oblivious or space-adaptive. These limitations are all related to the fact that their technique uses one single-writer/multi-reader variable for each process and LL/SC variable implemented. When a process executes an SC operation, it stores the new value in the single-writer/multi-reader variable, and then attempts to modify a shared location so that all processes will observe the new value as the abstract value of the LL/SC variable.

More recently, Jayanti and Petrovic have developed a wait-free LL/SC implementation that is both pointer clean and population oblivious [JP05][8]. The new implementation employs single-writer/multi-reader registers in a fashion similar to their earlier proposal [JP03]. However, these registers are managed within a structure, called a *dynamic array*, that allows for the number of registers to be increased dynamically, thus achieving population obliviousness. The resulting algorithm uses $O(V^2 + N)$ space. Their dynamic array is an array that provides wait-free concurrent operations, and that allows writes to occur at any index, expand ing as necessary. The implementation of the dynamic array presented in [JP05] is not space-adaptive, and it is not clear how a space-adaptive implementation could be constructed.

Anderson and Moir [AM99] also describe a wait-free implementation of wide LL/SC variables that requires $O(VN^2)$ space. Again, their algorithm is neither population oblivious, nor space adaptive.

The general techniques for lock-free memory management outlined in Section 5.7 of Chapter 5 can be used to implement lock-free LL/SC variables, in much the same way as garbage collection can. However, none of the extant memory management techniques are both population oblivious and space-adaptive, so any LL/SC implementation based on them inherits these limitations. However, because the memory-management techniques do not involve modification of shared counters during each operation, they are likely to result in significantly faster LL/SC implementations.

6.5 Verifying the LL/SC Implementation

In this section, we describe the verification of the LL/SC algorithm given in this chapter. This verification uses forward simulation only, not requiring backward simulation. The simulation relation used is complicated relative to the forward simulation of Chapter 3, reflecting the complexity of the algorithm itself. However, the techniques used are fundamentally the same. For this reason, we do not describe the verification in as much detail as we did the verifications of Chapters 3 and 4.

As with the M&S queue, the LL/SC implementation uses dynamically allocated memory. However, unlike the M&S queue, it releases that memory back to the system. Therefore, we must use a heap model in which dereferencing an unallocated pointer causes an error. This reflects the fact that in most systems, accesses to unallocated memory are illegal. Such a model is described in Section 6.5.2. The primary interest in this verification (apart from the

[8] A population aware version of this algorithm is presented in [JP07]

assurance it provides that our LL/SC implementation is correct) is this heap model, and the implications that it has for our simulation proof.

The present verification uses only two automata: *AbsAut*, modelling the specification, and *ConcAut*, modelling the implementation. Because there is no prophetic linearisation, we do not need a backward simulation or an intermediate automaton. The specification automaton *AbsAut*, is the canonical automaton for the LL/SC datatype, both of which are described in Section 6.5.1. The implementation automaton *ConcAut*, models the LL/SC algorithm directly, and is explained in Section 6.5.3. We define a forward simulation between the two automata as defined in Section 6.5.4. A proof has been constructed using the PVS proof assistant that this relation is in fact a simulation.

6.5.1 The LL/SC Datatype and the Abstract Automaton

An LL/SC variable contains a current value (taken from some set *V*), and provides an LL operation that reads the current value and an SC operation that modifies the current value, assuming there has been no successful SC in the interval between the process's SC and its last LL. Lets $Vars = V \times \mathbb{P}(PROC)$. An *LL/SC variable* $x \in Vars$ is a pair $(x.val, x.procs)$, where $x.val \in V$ and $x.procs \subseteq PROC$. Informally, $x.val$ is the current value of the variable, and $x.procs$ is the set of processes that may currently execute successful SC operations. We model LL operations using the function $LL : PROC \times Vars \to Vars \times V$ defined by

$$LL(p, x) = ((x.val, x.procs \cup \{p\}), x.val)$$

We model SC operations using the function $SC : PROC \times Vars \times V \to Vars \times bool$ defined by

$$SC(p, x, v) = \begin{cases} ((v, \varnothing), \mathit{true}) & \text{if } p \in x.procs \\ (x, \mathit{false}) & \text{otherwise} \end{cases}$$

Given an initial value v_0, we define the initial states of the LL/SC datatype to be

$$Init = \{x : Vars \mid x.val = v_0 \wedge x.procs = \varnothing\}$$

Note that each operation of the LL/SC datatype depends on the invoking process, but we wish to constrain our specification automaton so that no process may invoke an LL or SC operation of a different process. This constitutes a constraint on the transition relation of the abstract automaton. One way to express this constraint is to give the invocations of the LL/SC datatype the invoking process as a parameter, and constrain the precondition of the corresponding invocations of the abstract automaton so that the process parameter of the LL/SC invocation matches the process-index of then abstract automaton. However, we feel that it is simpler to define the invocations of the LL/SC datatype so that they do *not* take a process as an argument, and then define the *do* steps of the abstract automaton so that they use

ll_inv_p :
pre $pc_p = idle$
eff $pc_p := ll\_inv$

$sc\_inv_p(v)$:
pre $pc_p = pc\_pending$
eff $pc_p := sc\_inv(v)$

do_ll_p :
pre $pc_p = ll\_inv$
eff $pc_p := ll\_resp(\pi_2(LL(p, var)))$
$var := \pi_1(LL(p, var))$

do_sc_p :
pre $pc_p = sc\_inv(v)$
eff $pc_p := sc\_resp(\pi_2(SC(p, var)))$
$var := \pi_1(SC(p, var))$

$ll\_resp_p(v)$:
pre $pc_p = ll\_resp(v)$
eff $pc_p := pc\_pending$

$sc\_resp_p(b)$:
pre $pc_p = sc\_resp(b)$
eff $pc_p := idle$

Figure 6.11: The transition relation of *AbsAut*.

the process-index of the *do* action as the first argument to the LL or SC functions. Therefore, we define the invocations and responses of the LL/SC datatype thus

$$I = \{ll\_inv\} \cup \{ll\_resp(v) \mid v \in V\}$$
$$R = \{ll\_resp(v) \mid v \in V\} \cup \{sc\_resp(b) \mid b \in bool\}$$

Because of this departure from the standard construction of the datatype, we do not define an update function for the LL/SC. We use the LL and SC functions directly in the transition relation of the abstract automaton.

There is a second constraint on the executions of the abstract automaton, which can be regarded as an extension of the well-formedness criterion of Section 2.2.2. Each process must invoke the LL and SC operations alternately. That is, SC may only be invoked by a process *p* in an execution when *p*'s most recent operation of *p* is an LL; and LL may only be invoked by *p* when *p*'s most recent operation (if it exists) is an SC. We ensure that the executions of the abstract automaton satisfy this constraint by introducing an extra program counter state $pc\_pending$. When a process *p* completes an LL operation, pc_p is set to $pc\_pending$, and the precondition of transitions representing the invocation of SC operations by *p* asserts that $pc_p = pc\_pending$.

Apart from the two caveats just describes, the abstract automaton *AbsAut* is just like the canonical automata used so far in this thesis. *AbsAut* has a shared variable var that holds the current value of the LL/SC variable. The initial status of *AbsAut* are defined as follows:

$$start_{AbsAut} = \{ab \mid ab.var \in Init \wedge \forall p \bullet ab.pc_p = idle\}$$

The transition relation is presented in Figure 6.11.

6.5.2 The Heap Model

The heap model that we present here is much like the model of Chapter 3, but is augmented with an operation $free$ to free pointers, as well as a way to represent the fact that dereferencing or freeing pointers that are not allocated can result in an error. We write $POINTER$ for the set of pointers, $HEAP$ for the set of heaps, and $FIELD = \{data, pred, status\}$ for the set of fields. A heap $h \in HEAP$ is a triple of the form

$$(h.unalloc, h.evalfn, h.error)$$

where

- $h.unalloc \subseteq POINTER$ is the set of unallocated pointers. We require that $h.unalloc$ be infinite.
- $h.evalfn : POINTER \times FIELD \to POINTER$ returns the value of each field of each node,
- and $h.error \in bool$ is a flag used to distinguish error states of the heap from ordinary states. That is, $h.error = true$ iff some unallocated pointer has ever been dereferenced, or passed to $free$.

We use several functions that access and modify the state of a heap, and the values of the various fields. As has already been mentioned, $free$ deallocates a node. Another function $load$ loads the value of a field, and $store$ updates the value of a field. Finally, `new` allocates a new node. We axiomatise these functions so that when the heap is in an error state (the *error* flag is set) the functions are undefined. Figure 6.12 presents these axiomatisations.

Note that these functions may be total (in fact, when formalised in PVS they are total). They are undefined on some heaps in the sense that we cannot conclude anything about the values that they take on those heaps.

The constraint that $unalloc$ be infinite allows us to avoid the additional complexity present in the verification of Chapter 3, where we made each process loop during allocation if there was no available pointer.

6.5.3 The Concrete Automaton

Our construction or the concrete automaton, called *ConcAut*, from the code is much like that of Chapter 3. A state *cs* of *ConcAut* has a program-counter variable $cs.pc_p$ for each process *p*, and a heap *cs.h*. Furthermore, *cs* has an *entry* variable $cs.entry \in \mathbb{N} \times bool \times bool$ modelling the `entry` value of the LL/SC algorithm. *cs.entry.count* models the `localC` field of the algorithm, *cs.entry.nlC* models the `nlC` flag, and *cs.entry.nlP* models the `nlP flag`.

The initial states of *ConcAut* are defined in Figure 6.13. *ConcAut* has the same external actions as *AbsAut*. We define the internal actions somewhat differently to the way we did in Chapter 3. We combine some of the steps of the algorithm into pairs, such that each pair

$$
\begin{aligned}
&\mathit{free}(h, pt) = h' \Rightarrow \\
&\qquad (\neg h.\mathit{error} \wedge pt \notin h.unalloc \Rightarrow \\
&\qquad\qquad h' = (\mathit{false}, h.unalloc \cup \{pt\}, h.eval)) \\
&\qquad \wedge \\
&\qquad (h.\mathit{error} \vee pt \notin h.unalloc \Rightarrow \\
&\qquad\qquad h'.\mathit{error})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{load}(h, pt, f) = (h', pt') \Rightarrow \\
&\qquad (\neg h.\mathit{error} \wedge pt \notin h.unalloc \Rightarrow \\
&\qquad\qquad h' = h \wedge pt' = h.eval(pt, f)) \\
&\qquad \wedge \\
&\qquad (h.\mathit{error} \vee pt \in h.unalloc \Rightarrow \\
&\qquad\qquad h'.\mathit{error})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{store}(h, pt, f, x) = h' \Rightarrow \\
&\qquad (\neg h.\mathit{error} \wedge pt \notin h.unalloc \Rightarrow \\
&\qquad\qquad h' = (h.\mathit{error}, h.unalloc, h.eval \oplus \{(pt, f) \rightarrow x\})) \\
&\qquad \wedge \\
&\qquad (h.\mathit{error} \vee pt \in h.unalloc \Rightarrow \\
&\qquad\qquad h'.\mathit{error})
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{new}(h) = (h', pt) \Rightarrow \\
&\qquad h.\mathit{error} = h'.\mathit{error} \\
&\qquad \wedge \\
&\qquad (\neg h.\mathit{error} \Rightarrow \\
&\qquad\qquad pt \neq \mathit{null} \wedge \\
&\qquad\qquad pt \in h.unalloc \wedge \\
&\qquad\qquad h'.eval = h.eval \wedge \\
&\qquad\qquad h'.unalloc = h.unalloc \setminus \{pt\})
\end{aligned}
$$

Figure 6.12: Axiomatisations of the heap functions.

$$
\begin{aligned}
&\mathit{start}_{\mathit{ConcAut}} \hat{=} && (6.1)\\
&\quad \{\mathit{cs} \mid \mathit{cs}.\mathit{entry} = (0,0) \wedge && (6.2)\\
&\qquad \mathit{valid\_pointer}(\mathit{cs}.h, \mathit{ptr}0) \wedge && (6.3)\\
&\qquad \mathit{valid\_pointer}(\mathit{cs}.h, \mathit{ptr}1) \wedge && (6.4)\\
&\qquad \mathit{cs}.h.eval(\mathit{ptr}0, \mathit{status}) = INITSTAT \wedge && (6.5)\\
&\qquad \mathit{cs}.h.eval(\mathit{ptr}0, \mathit{pred}) = \mathit{ptr}1 \wedge && (6.6)\\
&\qquad \mathit{cs}.h.eval(\mathit{ptr}1, \mathit{status}) = (0, \mathit{true}, \mathit{false})\} && (6.7)
\end{aligned}
$$

Figure 6.13: The initial states of *ConcAut*.

contains some local operation, and at most one read, write or CAS operation. For example, *ConcAut*has an action $trans\_3\_4_p$ modelling the execution by some process *p* of the lines T3 and T4 of the `transfer` procedure in Figure 6.7. This constitutes a local operation (the construction of a new `status` value), and one CAS.

This technique slightly reduces the number of actions that we must consider, and helps to reduce the complexity of the verification. However, it means that *ConcAut* does not directly model all the interleavings possible in the actual algorithm. That is, we only model executions in which certain pairs of actions are always adjacent, when in fact, they may be separated by the actions of other processes. We justify this on the basis that all but one action of each such pair may be reordered arbitrarily with the actions of all other processes. The actions that model more than one step of the algorithm can be identified from the fact that they contain more than one line number.

As in Chapter 3, we define a notation to use the heap functions in a more natural fashion. For $\mathit{cs} \in \mathit{statesConcAut}$, $\mathit{pt} \in POINTER$ and $f \in FIELD$, let

$$\mathit{pt} \xrightarrow{\mathit{cs}} f = load(cs.h, f)$$

6.5.4 The Simulation Relation

In this section we describe a forward simulation relation from *ConcAut* to *AbsAut*, the existence of which guarantees that the traces of *ConcAut* are traces of *AbsAut*. This is by far the most complicated simulation relation presented so far in the thesis, but it is constructed along the same lines as the forward simulation of Chapter 3. The most interesting aspect of the verification is how we show that no unallocated pointer is dereferenced.

Figure 6.19 presents the simulation relation *SR*, which is an existential quantification over five functions. The predicate rel, also in Figure 6.19 describes the properties of these functions, and their relationship to related states of the two automata. Each of these functions records some aspect of the history of an execution. The domain of each function is $\mathbb{N}$, and

ll_inv_p :
pre $pc_p = idle$
eff $pc_p := pc\_ll\_2\_3$

$ll\_2\_3_p$:
pre $pc_p = pc\_ll\_2\_3$
eff $pc_p := pc\_ll\_4$,
$ll\_e_p := entry$,
$myver_p := entry.ver$

ll_5_p :
pre $pc_p := pc\_ll\_5$
eff let $suc = (ll\_e_p = entry)$ in
$pc := suc \; ? \; pc\_resp\_ll$
$: \; pc\_ll\_2\_3$,
$entry :=$
$suc \; ? \; (entry.ver,$
$entry.count + 1)$
$: entry$

$ll\_resp_p(v)$:
pre $pc_p = pc\_resp\_ll \wedge$
$pt = \pi_2(load\_data(h, mynode_p))$
eff $pc_p := pc\_pending$,
$h := \pi_1(load\_data(h, mynode_p))$

ll_4_p :
pre $pc_p = pc\_ll\_4$
eff $pc_p := pc\_ll\_5$,
$mynode_p :=$
$CURRENT\_PTR(s, myver_p)$

,

Figure 6.14: The LL transitions of *ConcAut*.

$sc\_inv_p(v)$:
pre $pc_p := pc\_pending$
eff $pc_p := pc\_sc\_1$,
$sc\_newd_p := v$

$sc\_resp_p(b)$:
pre $pc_p := pc\_resp\_sc \wedge$
$sc\_success_p = b$
eff $pc_p := idle$

sc_1_p :
pre $pc_p = pc\_sc\_1$
eff let $(newh, pt) = new(h)$ in
$pc_p := pc\_sc\_2a$,
$h := newh$,
$sc\_new\_nd_p := pt$

sc_2a_p :
pre $pc_p = pc\_sc\_2a$
eff $pc_p := pc\_sc\_2b$,
$h := store\_data(h,$
$sc\_new\_nd_p,$
$sc\_newd_p)$

sc_2b_p :
pre $pc_p = pc\_sc\_2b$
eff $pc_p := pc\_sc\_2c$,
$h := store\_pred(h,$
$sc\_new\_nd_p, mynode_p)$

sc_2c_p :
pre $pc_p = pc\_sc\_2c$
eff let $newh = store\_stat(h,$
$sc\_new\_nd_p,$
$INITSTAT)$ in
$pc_p := pc\_sc\_3$,
$h := newh$

sc_3_p :
pre $pc_p = pc\_sc\_3$
eff $pc_p := pc\_sc\_4$,
$(h, pred\_nd_p) :=$
$load\_pred(h, mynode_p)$

sc_4_p :
pre $pc_p = pc\_sc\_4$
eff let $(newptr0, newptr1, suc) =$
$CAS\_NONCURADDR$
$(s, myver_p,$
$pred\_nd_p,$
$sc\_new\_nd_p)$ in
$pc_p := pc\_sc\_5$,
$sc\_success_p := true$,
$ptr0 := sc\_new\_nd_p$,
$ptr1 := newptr1$

sc_5_p :
pre $pc_p = pc\_sc\_5$
eff $pc_p := pc\_sc\_6$,
$h := \neg sc\_success_p\ ?$
$free(h, sc\_new\_nd_p)$
$: h$

sc_6_p :
pre $pc_p = pc\_sc\_6$
eff $pc_p := pc\_sc\_7\_9$,
$sc\_e_p := entry$

Figure 6.15: The SC transitions of *ConcAut* (continued in Figure 6.16).

$sc\_7\_9\_no_p$:
pre $pc_p = pc\_sc\_7\_9 \wedge$
$sc\_e_p.ver \neq myver_p$
eff $pc_p := pc\_rel\_3$

$sc\_7\_9\_yes_p$:
pre $pc_p = pc\_sc\_7\_9 \wedge$
$sc\_e_p.ver = myver_p \wedge$
$sc\_e_p = entry$
eff $pc_p := pc\_trans\_2$,
$entry := (sc\_e_p.ver + 1, 0)$,
$trans\_count_p := sc\_e_p.count$

$sc\_7\_9\_loop_p$:
pre $pc_p = pc\_sc\_7\_9$
$sc\_e_p.ver = myver_p \wedge$
$sc\_e_p \neq entry$
eff $pc_p := pc\_sc\_6$

Figure 6.16: The SC transitions of *ConcAut* (continued from Figure 6.15).

each function should be thought of as taking each *i* that has been the version number at some earlier point in the execution to some process or pointer that had a special relationship with the version number. For some execution of *ConcAut*, let *i* be a natural number such that *i* was the version number at some point in the execution.

- *buffer*(i) is the value of the pointer that was current when *n* was the version number.
- *winner*(i) is the process whose successful SC operation made *n* the current version number.
- If *i* is no longer the version number, *transferer*(i) is the process that executed the successful CAS operation that changed the version number from *i* to $i + 1$, and thus transferred the hold count from *entry* to *buffer*(i).
- If the *nlP* flag of *buffer*(i) has been set, *setter*(i) is the process that set that flag.
- If *buffer*(i) has been freed since *n* was the version number, *releaser*(i) is the process that called *free* on *buffer*(i), releasing its memory back to the system.

As we shall see, these functions are applied throughout the simulation relation to constrain properties of the nodes, and various processes. Furthermore, using these functions, it is easy to show that there is a unique process that sets the *nlP* flag of each node, and a unique process that releases each node.

As with previous verifications, the transfer condition of the definition of forward simulation allows us to define a new function for each poststate, so that the properties listed above can be preserved. In what follows, we use the convention that if *f* is one of the function

$trans\_2_p$:
pre $pc_p = pc\_trans\_2$
eff $pc_p := pc\_trans\_3\_4$,
$(h, pre_p) :=$
$load\_stat(h, mynode_p)$

$trans\_3\_4_p$:
pre $pc_p = pc\_trans\_3\_4$
eff let $post_p = (pre_p.count +$
$trans\_count_p,$
$true, pre_p.nlP)$ in
let $(newh, suc) = CSTAT(h,$
$mynode_p,$
$pre_p, post)$ in
$pc_p := suc\ ?\ pc\_rel\_3\ :\ pc\_trans\_2$,
$h := newh$

rel_3_p :
pre $pc_p = pc\_rel\_3$
eff $pc_p := pc\_rel\_4\_6$
$(h, pre_p) :=$
$load\_stat(h, mynode_p)$

$rel\_4\_6\_no_p$:
pre $pc_p = pc\_rel\_4\_6 \wedge$
$pre_p \neq status(h, mynode_p)$
eff $pc_p := pc\_rel\_3$

$rel\_4\_6\_yes_p$:
pre $pc_p = pc\_rel\_4\_6 \wedge$
$pre_p = status(h, mynode_p)$
eff let $post_p = (pre_p.count - 1,$
$pre_p.nlC, pre_p.nlP)$ in
let $newh = \pi_1(CSTAT(h, mynode_p,$
$pre_p, post))$ in
$pc_p := CLEAN(post)\ ?\ pc\_set\_2$
$:\ pc\_rel\_7$,
$rel\_post_p := post$,
$h := newh$

rel_7_p :
pre $pc_p = pc\_rel\_7$
eff $pc_p := pc\_resp\_sc$,
$h := FREEABLE(rel\_post_p)\ ?$
$free(h, mynode_p)$
$:\ h$

Figure 6.17: The `transfer` and `release` transitions of *ConcAut*.

set_2_p :
pre $pc_p = pc\_set\_2$
eff $pc := pc\_set\_3\_4$,
$(h, pre_p) :=$
$load\_stat(h, pred\_nd_p)$

$set\_3\_4_p$:
pre $pc_p = pc\_set\_3\_4$
eff let $post = (pre_p.count,$
$pre_p.nlC, true)$ in
let $(newh, suc) =$
$CSTAT(h, pred\_nd_p,$
$CSTAT\ pre_p, post)$ in
$pc_p := suc\ ?\ pc\_set\_5$
$:\ pc\_set\_2$,
$h := newh$

set_5_p :
pre $pc_p = pc\_set\_5$
eff let $post = (pre_p.count,$
$pre_p.nlC, true)$in
$pc_p := pc\_rel\_7$,
$h := FREEABLE(post)\ ?$
$free(h, pred\_nd_p)$
$:\ h$

Figure 6.18: The `setNLPred` transitions of *ConcAut*.

$$
\begin{array}{l}
SR(as, cs) \mathrel{\hat{=}} \\
\quad \exists\, buffer, winner, transferer, setter, releaser \bullet \\
\quad rel(as, cs, buffer, winner, transferer, setter, releaser)
\end{array}
$$

$$
\begin{array}{l}
rel(as, cs, buffer, winner, transferer, setter, releaser) \mathrel{\hat{=}} \\
\quad \neg error(h(cs)) \wedge \\
\quad rc\_ok(cs, buffer, transferer) \wedge \\
\quad buffers\_ok(as, cs, buffer, winner, transferer) \wedge \\
\quad ll\_lin\_ok(as, cs, buffer, winner) \wedge \\
\quad sc\_lin\_ok(as, cs, buffer, winner) \wedge \\
\quad persistents\_ok(cs, buffer) \wedge \\
\quad distinctness\_ok(cs, buffer, transferer, setter, releaser) \wedge \\
\quad ll\_ok(as, cs) \wedge \\
\quad sc\_ok(as, cs) \wedge \\
\quad trans\_ok(cs, transferer) \wedge \\
\quad set\_ok(cs, buffer, transferer, setter) \wedge \\
\quad release\_ok(cs, releaser) \wedge \\
\quad status\_ok(cs, buffer, transferer, setter, releaser)
\end{array}
$$

Figure 6.19: The simulation relation *SR*, and the predicate rel.

variables whose existence is asserted by *SR*, then f' is the new function used to witness the variable f in *SR* over the abstract and concrete poststates. For each transition of *ConcAut*, $cs \xrightarrow{a} cs'$, and abstract state *as* such that $SR(as, cs)$, we choose functions to witness $SR(as', cs')$ as follows:

- If $a = sc\_4\_yes\_p$ for some *p*, (modelling successful execution of the CAS at line S4), then

  $$buffer' = buffer \oplus \{cs.entry.ver + 1 \to cs.sc\_new\_nd_p\}$$

  Otherwise $buffer' = buffer$.

- If $a = sc\_4\_yes_p$ for some *p*, then

  $$winner' = winner \oplus \{cs.entry.ver + 1 \to p\}$$

  Otherwise $winner' = winner$.

- If $a = sc\_7\_9\_yes_p$ for some *p* (modelling successful execution of the CAS at line S7), then

  $$transferer' = transferer \oplus \{cs.entry.ver \to p\}$$

  Otherwise $transferer' = transferer$.

- If $a = rel\_4\_6\_yes_p$ for some *p* (modelling the successful execution of the CAS at line R6) and the value of $cs'.mynode_p \xrightarrow{cs'} status$ is CLEAN then

  $$setter' = setter \oplus \{cs.myver_p - 1 \to p\}$$

  Therefore, *p* becomes the setter for the node that was current prior to the linearisation point of *p*'s earlier LL operation. For other *a*, $setter' = setter$.

- If $a = rel\_4\_6\_yes_p$ for some *p*, and the value of $cs'.mynode_p \xrightarrow{cs'} status$ is FREEABLE then

  $$releaser' = releaser \oplus \{cs.myver_p \to p\}$$

  Otherwise, $releaser' = releaser$.

We elaborate on the importance of these functions shortly. For now, we remark that the functions $transferer$, $setter$ and $releaser$ make it easy to prove that for each version number, there is only one process that will perform the `transfer`, `setNLPred` or `free` operations on the current node of that version number.

An important concept in our description of the algorithm in Section 6.2 was the notion of *pinning* a node. Figure 6.21 presents the predicate $pinning$ that formalises an analogous

$$rc(cs, buffer, transferer, i) \mathrel{\hat{=}}$$

$$\begin{cases} 0 & cs.entry.ver < i \\ cs.entry.count & i = cs.entry.ver \\ buffer(i) \xrightarrow{cs} status.count + & \\ \quad trans\_count(cs)(transferer(i)) & during\_transfer(cs, transferer, i) \\ buffer(i) \xrightarrow{cs} status.count & \exists\, p \bullet pinning(cs, i)(p) \\ 0 & \text{otherwise} \end{cases} \tag{6.8}$$

Figure 6.20: The reference-counting function rc.

notion for *version numbers*. A process *p pins i* at every point during the interval beginning with the successful execution of the CAS at line L4 (ll_4_p), and ending with the successful execution of the CAS at line R5 ($rel\_4\_6\_yes_p$), and when $i = myver_p$. This is expressed using program counter values.

One key difficulty in this verification is being able to prove that no read, write or CAS to the heap dereferences a pointer that has been deallocated. We achieve this with the help of a reference counting function, which is defined over concrete states using the $buffer$ and $transferer$ functions. This function does not assign reference counts to nodes, rather, it assigns reference counts to natural numbers. As we shall see, the simulation relation asserts that, for each dereference in the algorithm (with some minor exceptions), the version number during the interval when the node was current has a nonzero reference count. Further, the simulation relation asserts that no node is deallocated until after the reference count of it's corresponding version number has reached zero. The reference-counting function rc is defined in Figure 6.20.

For each natural *i*, the reference count of *i* is zero until the version number reaches *i*. While *i* is the version number, its reference count is the value of the hold count. While the hold count is being transfered to $buffer(i)$ by the process $transferer(i)$, *i*'s reference count is the local count of $buffer(i)$ plus the value that $transferer(i)$ is about to add to $buffer(i) \rightarrow count$. (The predicate $during\_transfer$ describes this interval formally, and is presented in Figure 6.21). Once this transfer has been accomplished, *i*'s reference count is the value of $buffer(i) \xrightarrow{count}$, until no process is pinning *i*, at which point *i*'s reference count becomes zero. Note that because each natural number is only the version number once in any execution, the reference count cannot become nonzero after there are no pinning processes.

Fix a transition $cs \xrightarrow{a} cs'$. It is straightforward to prove the following:

- If $a = sc\_4\_yes_p$ for some *p*, then $rc(cs', buffer', transferer', cs'.entry.ver) = 0$.
- If $a = ll\_5_p$ for some *p*, and $cs.ll\_e_p = cs.entry$,

$$rc(cs, buffer, transferer, cs.myver_p) + 1 = rc(cs', buffer', transferer', cs'.myver_p)$$

$$
\begin{aligned}
&\mathit{pinning}(\mathit{cs}, i) \hat{=} \{p \mid \\
&\quad myver(\mathit{cc})(p) = i \wedge \\
&\quad (\mathit{cs}.pc_p = pc\_resp\_ll \vee \mathit{cs}.pc_p = pc\_pending \vee \\
&\quad \mathit{cs}.pc_p = pc\_sc\_1 \vee \mathit{cs}.pc_p = pc\_sc\_2a \vee \\
&\quad \mathit{cs}.pc_p = pc\_sc\_2b \vee \mathit{cs}.pc_p = pc\_sc\_2c \vee \\
&\quad \mathit{cs}.pc_p = pc\_sc\_3 \vee \mathit{cs}.pc_p = pc\_sc\_4 \vee \\
&\quad \mathit{cs}.pc_p = pc\_sc\_5 \vee \mathit{cs}.pc_p = pc\_sc\_6 \vee \\
&\quad \mathit{cs}.pc_p = pc\_sc\_7\_9 \vee \mathit{cs}.pc_p = pc\_trans\_2 \vee \\
&\quad \mathit{cs}.pc_p = pc\_trans\_3\_4 \vee \mathit{cs}.pc_p = pc\_rel\_3 \vee \\
&\quad \mathit{cs}.pc_p = pc\_rel\_4\_6)\}
\end{aligned}
$$

Figure 6.21: The set $pinning$ of processes that pin a natural number.

$$
\begin{aligned}
&\mathit{during\_transfer}(\mathit{cs}, \mathit{transferer}, i) \hat{=} && (6.9)\\
&\quad \text{let } \mathit{tr} = \mathit{transferer}(i) \text{ in} && (6.10)\\
&\qquad (\mathit{cs}.pc_{\mathit{tr}} = \mathit{pc\_trans\_2} \vee && (6.11)\\
&\qquad\ \mathit{cs}.pc_{\mathit{tr}} = \mathit{pc\_trans\_3\_4}) \wedge && (6.12)\\
&\qquad \mathit{cs}.\mathit{myver}_{\mathit{tr}} = i && (6.13)
\end{aligned}
$$

Figure 6.22: The $during\_transfer$ predicate.

$$\mathit{rc\_ok}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}) \mathrel{\hat{=}} \tag{6.14}$$

$$\forall\, i \bullet \colon \mathit{rc}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, i) = |\ \mathit{pinning}(\mathit{cs}, i)\ | \tag{6.15}$$

Figure 6.23: The rc_ok predicate.

- If $a = rel\_4\_6\_yes_p$ for some p, then

$$rc(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{cs.myver}_p) - 1 = rc(\mathit{cs}', \mathit{buffer}', \mathit{transferer}', \mathit{cs}'.\mathit{myver}_p)$$

- No other action changes the value of rc at any other $i \in \mathbb{N}$. In particular, if $a = trans\_3\_4_p$ and $\mathit{cs}.mynode_p \xrightarrow{\mathit{cs}} \mathit{status} = \mathit{cs}.\mathit{pre}_p$, then the reference count of $\mathit{cs}.\mathit{myver}_p$ (and all other integers) is unchanged.

Equipped with the functions rc and $pinning$, we are in a position to define an important property that enables us to prove that no dereference to unallocated memory occurs. Figure 6.23 defines the rc_ok predicate, which states that the rc function accurately counts the number of processes that have pinned each natural number.

After the reference count of a number i has fallen to 0, certain operations may still be applied to $\mathit{buffer}(i)$. In particular, $\mathit{buffer}(i) \rightarrow \mathit{status.nlP}$ may be set during a `setNLPred` operation, or $buffer(i)$ may be passed to $free$during a `release` or `setNLPred` operation. In each case, we need to show that when these operations occur, $buffer(i)$ is still allocated. The primary goal of the set_ok and $release\_ok$ predicates is to state properties of the algorithm during those operations that allow us to prove that no node is freed when it's associated version number has a nonzero reference count, and moreover, that no node is freed while a process executing `setNLPred` or `release` might access the node. (Figure 6.24 presents the set_ok predicate. Auxiliary predicates are presented in Figure 6.25. Figure 6.26 presents the $release\_ok$ predicate.)

The $status\_ok$ predicate is presented in Figure 6.27. This predicate describes properties of the `status` field of each node, during the interval where the version number corresponding to the node is *active*. A number i is active from the point when $i = \mathit{entry.ver} - 1$ until the point immediately before $\mathit{buffer}(i)$ is passed to *free*.

The predicate $buffers\_ok$, presented in Figure 6.29, describes the state of the $entry$ field, and the nodes referenced by $ptr0$ and $ptr1$. In particular, this predicate describes the two configurations of the algorithm that are illustrated in Figure 6.2 on page 166. It also asserts that the `data` field of the current node contains the same value as the LL/SC variable in related states of the abstract automaton.

$$
\begin{aligned}
&\mathit{set\_ok}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}) \mathrel{\hat{=}} \\
&\quad \mathit{set\_ok}1(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}) \wedge && (6.16)\\
&\quad \mathit{set\_ok}2(\mathit{cs}) && (6.17)\\
& && (6.18)\\
&\mathit{set\_ok}1(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}) \mathrel{\hat{=}} \\
&\quad \forall\, p \bullet (\mathit{in\_set}(\mathit{cs}, p) \wedge \neg \mathit{slow\_set}(\mathit{cs}, p) \Rightarrow && (6.19)\\
&\qquad \mathit{cs.myver}_p < \mathit{cs.entry.ver} \wedge && (6.20)\\
&\qquad p = \mathit{setter}(\mathit{cs.myver}_p - 1) \wedge && (6.21)\\
&\qquad \mathit{rc}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{cs.myver}_p) = 0 \wedge && (6.22)\\
&\qquad (\mathit{cs}.pc_p = \mathit{pc\_set}\_5 \vee && (6.23)\\
&\qquad\ \neg \mathit{thepred}(\mathit{cs}, p) \xrightarrow{\mathit{cs}} \mathit{status.nlP}) \wedge && (6.24)\\
&\qquad\ \neg \mathit{unallocated}(\mathit{cs.h}, \mathit{thepred}(\mathit{cs}, p))) && (6.25)\\
&\quad\ \wedge && (6.26)\\
&\quad\ (\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_set}\_5 \wedge && (6.27)\\
&\quad\ \mathit{FREEABLE}(\mathit{cs.pre}_p, \mathit{count}, \mathit{cs.pre}_p\mathit{.nlC}, \mathit{true}) \Rightarrow && (6.28)\\
&\qquad (\mathit{cs.pre}_p\mathit{.count}, \mathit{cs.pre}_p\mathit{.nlC}, \mathit{true}) = && (6.29)\\
&\qquad\quad \mathit{thepred}(\mathit{cs}, p) \xrightarrow{\mathit{cs}} \mathit{status}) && (6.30)\\
& && (6.31)\\
&\mathit{set\_ok}2(\mathit{cs}) \mathrel{\hat{=}} \\
&\quad \forall\, p, q \bullet \mathit{cs}.pc_p = \mathit{pc\_set}\_5 \wedge \\
&\qquad \mathit{FREEABLE}(\mathit{cs.pre}_p\mathit{.count}, \mathit{cs.pre}_p\mathit{.nlC}, \mathit{true}) \wedge && (6.32)\\
&\qquad (\mathit{in\_set}(\mathit{cs}, q) \vee \mathit{pc}(\mathit{cs})(q) = \mathit{pc\_rel}\_7) \wedge \\
&\qquad \mathit{cs.myver}_p = \mathit{cs.myver}_q + 1 \Rightarrow \\
&\qquad\quad \neg \mathit{FREEABLE}(\mathit{cs.rel\_post}_q) && (6.33)
\end{aligned}
$$

Figure 6.24: The set_ok predicate.

$$
\begin{aligned}
&\mathit{in\_set}(\mathit{cs}, p) \mathrel{\hat{=}} \\
&\qquad \mathit{cs}.pc_p = pc\_set\_2 \vee \\
&\qquad \mathit{cs}.pc_p = pc\_set\_3\_4 \vee \\
&\qquad \mathit{cs}.pc_p = pc\_set\_5
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{slow\_set}(\mathit{cs}, p) \mathrel{\hat{=}} \\
&\qquad \mathit{cs}.pc_p = pc\_set\_5 \wedge \\
&\qquad \neg FREEABLE((\mathit{cs}.pre_p.count, \mathit{cs}.\mathit{pre}_p.\mathit{nlC}, \mathit{true}))
\end{aligned}
$$

Figure 6.25: Auxiliary predicates of set_ok.

$$
\begin{aligned}
&\mathit{release\_ok}(\mathit{cs}, \mathit{releaser}) \mathrel{\hat{=}} \\
&\quad \forall p \bullet ((\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_3 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_4\_6 \vee \\
&\qquad \mathit{in\_set}(\mathit{cs}, p) \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_7) \\
&\qquad\quad \Rightarrow \mathit{myver}(\mathit{cs})(p) < \mathit{ver}(\mathit{entry}(\mathit{cs}))) && (6.34) \\
&\quad \wedge && (6.35) \\
&\quad ((\mathit{in\_set}(\mathit{cs}, p) \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_7) \wedge \\
&\qquad \mathit{FREEABLE}(\mathit{rel\_post}(\mathit{cs})(p)) \\
&\qquad\quad \Rightarrow p = \mathit{releaser}(\mathit{myver}(\mathit{cs})(p)) \wedge && (6.36) \\
&\qquad\quad \mathit{cs}.\mathit{rel\_post}_p = \mathit{cs}.\mathit{mynode}_p \overset{cs}{\rightarrow} \mathit{status} \wedge && (6.37) \\
&\qquad\quad \mathit{valid\_pointer}(\mathit{cs}.h, \mathit{cs}.\mathit{mynode}_p)) && (6.38)
\end{aligned}
$$

Figure 6.26: The $release\_ok$ predicate.

$$status\_ok(cs, buffer, transferer, setter, releaser) \mathrel{\hat{=}} \tag{6.39}$$
$$\quad status\_ok1(cs, buffer, transferer, setter, releaser) \wedge \tag{6.40}$$
$$\quad status\_ok2(cs, buffer, transferer, setter, releaser) \wedge \tag{6.41}$$
$$\quad status\_ok3(cs, buffer, transferer, setter, releaser) \tag{6.42}$$
$$\tag{6.43}$$
$$status\_ok1(cs, buffer, transferer, setter, releaser) \mathrel{\hat{=}} \tag{6.44}$$
$$\quad \forall\, i \bullet i < cs.entry.ver \wedge active\_version(cs, buffer, transferer, setter, releaser, i) \Rightarrow \tag{6.45}$$
$$\quad (\neg buffer(i) \xrightarrow{cs} status.nlC \Leftrightarrow during\_transfer(cs, transferer, i))) \tag{6.46}$$
$$\tag{6.47}$$
$$status\_ok2(cs, buffer, transferer, setter, releaser) \mathrel{\hat{=}} \tag{6.48}$$
$$\quad \forall\, i \bullet i < cs.entry.ver \wedge \tag{6.49}$$
$$\qquad active\_version(cs, buffer, transferer, setter, releaser, i) \Rightarrow \tag{6.50}$$
$$\qquad i = cs.entry.ver - 1 \vee \tag{6.51}$$
$$\qquad (\neg buffer(i) \xrightarrow{cs} status.nlP \wedge (\exists\, p \bullet p \in pinning(cs, i+1))) \vee \tag{6.52}$$
$$\qquad (\neg buffer(i) \xrightarrow{cs} status.nlP \Leftrightarrow during\_set(cs, setter, i)) \tag{6.53}$$

$$status\_ok3(cs, buffer, transferer, setter, releaser) \mathrel{\hat{=}} \tag{6.54}$$
$$\quad \forall\, i \bullet (i < ver(entry(cs)) \wedge \tag{6.55}$$
$$\qquad active\_version(cs, buffer, transferer, setter, releaser, i) \Rightarrow \tag{6.56}$$
$$\qquad i = ver(entry(cs)) - 1 \vee \tag{6.57}$$
$$\qquad 0 < rc(cs, buffer, transferer)(i) \vee \tag{6.58}$$
$$\qquad buffer(i) \xrightarrow{cs} status.count = 0) \tag{6.59}$$

Figure 6.27: The $status\_ok$ predicates.

$$
\begin{array}{l}
\mathit{active\_version}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}, i) \mathrel{\hat{=}} \\
\quad \text{let } \mathit{rl} = \mathit{releaser}(i) \text{ in} \\
\quad \text{let } \mathit{st} = \mathit{setter}(i) \text{ in} \\
\quad \text{let } \mathit{pre} = \mathit{cs}.pre_{\mathit{st}} \text{ in} \\
\qquad (i = \mathit{cs.entry.ver} + 1 \wedge \mathit{transient}(\mathit{cs})) \vee & (6.60) \\
\qquad i = \mathit{cs.ver.entry} \vee & (6.61) \\
\qquad i = \mathit{cs.ver.entry} - 1 \vee & (6.62) \\
\qquad 0 < \mathit{rc}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, i) \vee & (6.63) \\
\qquad 0 < \mathit{rc}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, i+1) \vee & (6.64) \\
\qquad (\mathit{in_set}(\mathit{cs}, \mathit{st}) \vee \mathit{cs.myver}_{\mathit{st}} = i + 1 \wedge & (6.65) \\
\qquad \neg(\mathit{cs}.pc_{\mathit{st}} = \mathit{pc_set}_5 \wedge & (6.66) \\
\qquad \neg \mathit{FREEABLE}((\mathit{pre}.count, \mathit{pre.nlC}, \mathit{true})))) \vee & (6.67) \\
\qquad ((\mathit{in_set}(\mathit{cs}, \mathit{rl}) \vee \mathit{cs}.pc_{\mathit{rl}} = \mathit{pc_rel}_7) \wedge & (6.68) \\
\qquad \mathit{FREEABLE}(\mathit{cs.rel_post}_{\mathit{rl}}) \wedge \mathit{cs.myver}_{\mathit{rl}} = i) & (6.69)
\end{array}
$$

Figure 6.28: The $active\_version$ predicate.

$$
\begin{aligned}
&\mathit{buffers\_ok}(\mathit{as}, \mathit{cs}, buffer, winner, transferer) \mathrel{\hat{=}} \\
&\quad \text{let } \mathit{cur} = \mathit{CURRENT\_PTR}(\mathit{cs}, \mathit{cs.entry.ver}), && (6.70)\\
&\quad \text{let } \mathit{noncur} = \mathit{OLD\_PTR}(\mathit{cs}, \mathit{cs.entry.ver}) \text{ in} && (6.71)\\
&\qquad (\mathit{cur} = \mathit{buffer}(\mathit{cs.entry.ver}) \wedge && (6.72)\\
&\qquad \mathit{cur} \xrightarrow{\mathit{cs}} \mathit{data} = \mathit{as}.var.val \wedge && (6.73)\\
&\qquad \mathit{cur} \xrightarrow{\mathit{cs}} \mathit{status} = \mathit{INIT\_STATUS} \wedge && (6.74)\\
&\qquad \mathit{cur} \xrightarrow{\mathit{cs}} \mathit{pred} = \mathit{buffer}(\mathit{cs.entry.ver} - 1) \wedge && (6.75)\\
&\qquad \mathit{valid\_pointer}(\mathit{cs.h}, \mathit{cur}) \wedge && (6.76)\\
&\qquad \mathit{valid\_pointer}(\mathit{cs.h}, \mathit{buffer}(\mathit{cs.entry.ver} - 1)) \wedge && (6.77)\\
&\qquad \neg \mathit{buffer}(\mathit{cs.entry.ver} - 1) \xrightarrow{\mathit{cs}} \mathit{status.nlP}) \wedge && (6.78)\\
&\qquad (\mathit{noncur} = \mathit{buffer}(\mathit{cs.entry.ver} - 1) && (6.79)\\
&\qquad \vee && (6.80)\\
&\qquad (\mathit{noncur} = \mathit{buffer}(\mathit{ver}(\mathit{entry}(\mathit{cs})) + 1) \wedge && (6.81)\\
&\qquad \mathit{transient}(\mathit{cs}) \wedge && (6.82)\\
&\qquad \mathit{awaiting\_lin}(\mathit{cs}, \mathit{cs.entry.ver}), \mathit{winner}(\mathit{cs.entry.ver} + 1) \wedge && (6.83)\\
&\qquad \mathit{as}.pc_p = sc\_resp(\mathit{old} \xrightarrow{\mathit{cs}} \mathit{data}) \wedge && (6.84)\\
&\qquad \mathit{old} \xrightarrow{\mathit{cs}} \mathit{status} = \mathit{INIT\_STATUS} \wedge && (6.85)\\
&\qquad \mathit{old} \xrightarrow{\mathit{cs}} \mathit{pred} = \mathit{CURRENT\_PTR}(\mathit{cs}, \mathit{cs.entry.ver}) \wedge && (6.86)\\
&\qquad \mathit{valid\_pointer}(\mathit{cs.h}, \mathit{old}))) && (6.87)
\end{aligned}
$$

Figure 6.29: The $buffers\_ok$ predicate.

$$
\begin{aligned}
&\mathit{awaiting\_lin}(cs, i, p) \mathrel{\hat{=}} \\
&\quad (cs.pc_p = pc\_set\_5 \vee && (6.88)\\
&\quad cs.pc_p = pc\_sc\_6 \vee && (6.89)\\
&\quad cs.pc_p = pc\_sc\_7\_9) \wedge && (6.90)\\
&\quad cs.myver_p = i && (6.91)\\
& && (6.92)\\
&\mathit{mods\_new\_node}(cs, p) \mathrel{\hat{=}} && (6.93)\\
&\quad cs.pc_p = pc\_sc\_2a \vee cs.pc_p = pc\_sc\_2b \vee && (6.94)\\
&\quad cs.pc_p = pc\_sc\_2c \vee cs.pc_p = pc\_sc\_3 \vee && (6.95)\\
&\quad cs.pc_p = pc\_sc\_4 \vee && (6.96)\\
&\quad (cs.pc_p = pc\_sc\_5 \wedge \neg cs.sc\_succ_p) && (6.97)
\end{aligned}
$$

Figure 6.30: Auxiliary predicates of the simulation relation.

$$
\begin{aligned}
&\mathit{transient}(cs) \mathrel{\hat{=}} \\
&\quad OLD\_PTR(cs, ver(cs.entry)) \neq \\
&\qquad \mathit{CURRENT\_PTR}(cs, cs.\mathit{entry}.\mathit{ver}) \overset{cs}{\rightarrow} pred
\end{aligned}
$$

Figure 6.31: The predicate *transient*, describing states in which the next SC to be linearised has been determined, but the linearisation point has not yet been reached.

$$
\mathit{thepred}(cs, p) \mathrel{\hat{=}}
\mathit{thep} \begin{cases}
\mathit{mynode}_p \overset{cs}{\rightarrow} \mathit{pred} & cs.pc_p = \mathit{pc\_resp\_ll} \vee cs.pc_p = \mathit{pc\_pending} \vee \\
& cs.pc_p = \mathit{pc\_sc\_1} \vee cs.pc_p = \mathit{pc\_sc\_2a} \vee \\
& cs.pc_p = \mathit{pc\_sc\_2b} \vee cs.pc_p = \mathit{pc\_sc\_2c} \vee \\
& cs.pc_p = \mathit{pc\_sc\_3} \\
cs.\mathit{pred\_nd}_p & \text{otherwise}
\end{cases}
$$

Figure 6.32: The *thepred* function, which returns the predecessor of the current node of a process.

$$
\begin{aligned}
&\mathit{during\_set}(cs, \mathit{setter}, i) \mathrel{\hat{=}} \\
&\quad \text{let } st = \mathit{setter}(i) \text{ in} && (6.98)\\
&\quad (cs.pc_{st} = \mathit{pc\_set\_2} \vee cs.pc_{st} = \mathit{pc\_set\_3\_4}) \wedge && (6.99)\\
&\quad cs.\mathit{myver}_{st} = i + 1 && (6.100)\\
& && (6.101)\\
&\mathit{after\_release}(cs, p) \mathrel{\hat{=}} \\
&\mathit{in\_set}(cs, p) \vee \mathit{pc}(cs)(p) = \mathit{pc\_rel\_7} && (6.102)
\end{aligned}
$$

Figure 6.33: Predicates defining important intervals in the execution of SC operations.

$$
\begin{aligned}
&\mathit{lin\_ok}1(\mathit{as}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p) \mathrel{\hat{=}} \\
&\quad (\mathit{cs.myver}_p = \mathit{cs.entry.ver} \equiv p \in \mathit{as}.var.procs) \wedge && (6.103)\\
&\quad (\mathit{thepred}(\mathit{cs}, p) = \mathit{OLD\_PTR}(\mathit{cs}, \mathit{cs.myver}_p) \Rightarrow \\
&\qquad \mathit{cs.myver}_p = \mathit{cs.entry.ver}) \wedge && (6.104)\\
&\quad (\mathit{thepred}(\mathit{cs}, p) \neq \mathit{OLD\_PTR}(\mathit{cs}, \mathit{cs.myver}_p) \Rightarrow p \neq \mathit{winnercs.myver}_p + 1) && (6.105)\\
& && (6.106)\\
&\mathit{lin\_ok}2(\mathit{as}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p) \mathrel{\hat{=}} \\
&\quad \neg \mathit{cs.sc\_success}_p \wedge && (6.107)\\
&\quad p \neq \mathit{winner}(\mathit{cs.myver}_p + 1) \wedge && (6.108)\\
&\quad (\mathit{cs.myver}_p = \mathit{cs.entry.ver} \Rightarrow \mathit{transient}(\mathit{cs})) \wedge && (6.109)\\
&\quad (\mathit{cs.myver}_p \neq \mathit{cs.entry.ver} \equiv p \notin \mathit{as}.var.procs) \wedge && (6.110)\\
&\quad \mathit{as}.pc_p = \mathit{pc\_do\_sc} && (6.111)\\
& && (6.112)\\
&\mathit{lin\_ok}3(\mathit{as}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p) \mathrel{\hat{=}} \\
&\quad \mathit{cs.sc\_success}_p) \wedge && (6.113)\\
&\quad p = \mathit{winner}(\mathit{cs.myver}_p + 1) \wedge && (6.114)\\
&\quad \mathit{cs.myver}_p = \mathit{cs.entry.ver} \wedge && (6.115)\\
&\quad \mathit{transient}(\mathit{cs}) \wedge && (6.116)\\
&\quad p \in \mathit{as}.var.procs \wedge && (6.117)\\
&\quad \mathit{as}.pc_p = \mathit{pc\_do\_sc} && (6.118)\\
& && (6.119)\\
&\mathit{lin\_ok}4(\mathit{as}, \mathit{cs}, p) \mathrel{\hat{=}} \\
&\quad \neg \mathit{cs.sc\_success}_p \wedge p \notin \mathit{as}.var.procs \wedge && (6.120)\\
&\quad \mathit{as}.pc_p = \mathit{pc\_do\_sc} && (6.121)\\
& && (6.122)\\
&\mathit{lin\_ok}5(\mathit{as}, \mathit{cs}, \mathit{winner}, p) \mathrel{\hat{=}} \\
&\quad (\mathit{cs.sc\_success}_p \equiv \mathit{as}.pc_p = sc\_resp(\mathit{true})) \wedge && (6.123)\\
&\quad \mathit{as}.pc_p = \mathit{pc\_resp\_sc} \wedge && (6.124)\\
&\quad \mathit{cs.myver}_p < \mathit{cs.entry.ver} && (6.125)
\end{aligned}
$$

Figure 6.34: The lin_ok predicates.

$$
\begin{aligned}
&\mathit{sc\_lin\_ok}(\mathit{as}, \mathit{cs}, \mathit{buffer}, \mathit{winner}) \mathrel{\hat{=}} \\
&\quad \forall\, p \bullet (\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_pending} \Rightarrow \\
&\qquad\quad \mathit{as}.pc_p = \mathit{pc\_pending} \wedge && (6.126)\\
&\qquad\quad \mathit{lin\_ok}1(\mathit{ab}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p)) && (6.127)\\
&\wedge && (6.128)\\
&\quad ((\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_1 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_2a \vee && (6.129)\\
&\quad \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_2b \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_2c \vee && (6.130)\\
&\quad \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_3 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_4) \Rightarrow \\
&\quad \mathit{pc}(\mathit{ab})(p) = \mathit{pc\_do\_sc} \wedge && (6.131)\\
&\quad \mathit{lin\_ok}1(\mathit{ab}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p)) && (6.132)\\
&\wedge && (6.133)\\
&\quad (\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_5 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_6 \vee && (6.134)\\
&\quad \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_sc}\_7\_9 \Rightarrow \\
&\quad \mathit{lin\_ok}2(\mathit{ab}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p) && (6.135)\\
&\quad \vee && (6.136)\\
&\quad \mathit{lin\_ok}3(\mathit{ab}, \mathit{cs}, \mathit{buffer}, \mathit{winner}, p) && (6.137)\\
&\quad \vee && (6.138)\\
&\quad (\mathit{lin\_ok}5(\mathit{ab}, \mathit{cs}, \mathit{winner}, p) \wedge && (6.139)\\
&\quad \mathit{success}(\mathit{ab})(p) \wedge && (6.140)\\
&\quad p = \mathit{winner}(\mathit{myver}(\mathit{cs})(p) + 1))) && (6.141)\\
&\quad \wedge && (6.142)\\
&\quad (\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_trans}\_2 \wedge p / = \mathit{winner}(\mathit{myver}(\mathit{cs})(p) + 1) \Rightarrow && (6.143)\\
&\quad \mathit{lin\_ok}4(\mathit{ab}, \mathit{cs}, p)) && (6.144)\\
&\quad \wedge && (6.145)\\
&\quad ((\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_trans}\_3\_4 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_3 \vee && (6.146)\\
&\quad \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_4\_6 \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel}\_7 \vee && (6.147)\\
&\quad \mathit{in}_s\mathit{et}(\mathit{cs}, p) \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_resp\_sc}) && (6.148)\\
&\quad \Rightarrow \mathit{lin\_ok}5(\mathit{ab}, \mathit{cs}, \mathit{winner}, p)) && (6.149)
\end{aligned}
$$

Figure 6.35: The *sc_lin_ok* predicate.

$$
\begin{aligned}
&\mathit{persistents\_ok}(\mathit{cs}, \mathit{buffer}) \mathrel{\hat{=}} \\
&\forall\, p \bullet (\neg(\mathit{pc}(\mathit{cs})(p) = \mathit{idle} \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_2\_3} \vee && (6.150)\\
&\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_4} \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_5} \vee && (6.151)\\
&\mathit{in\_set}(\mathit{cs}, p) \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_rel\_7} \vee && (6.152)\\
&\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_resp\_sc}) \Rightarrow \\
&\neg \mathit{unallocated}(\mathit{cs}.h, \mathit{thepred}(\mathit{cs}, p)) \wedge && (6.153)\\
&\neg \mathit{thepred}(\mathit{cs}, p) \overset{\mathit{cs}}{\rightarrow} \mathit{status.nlP}) && (6.154)\\
&\wedge && (6.155)\\
&(\neg(\mathit{pc}(\mathit{cs})(p) = \mathit{idle} \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_2\_3} \vee && (6.156)\\
&\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_4} \vee \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_ll\_5} \vee && (6.157)\\
&\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_resp\_sc}) \Rightarrow \\
&(\mathit{valid\_pointer}(h(\mathit{cs}), \mathit{mynode}(\mathit{cs})(p)) \vee \mathit{after\_release}(\mathit{cs}, p)) \wedge && (6.158)\\
&\mathit{thepred}(\mathit{cs}, p) \neq \mathit{null} \wedge && (6.159)\\
&\mathit{mynode}(\mathit{cs})(p) = \mathit{buffer}(\mathit{myver}(\mathit{cs})(p)) \wedge && (6.160)\\
&\mathit{thepred}(\mathit{cs}, p) = \mathit{buffer}(\mathit{myver}(\mathit{cs})(p) - 1) \wedge && (6.161)\\
&\mathit{mynode}(\mathit{cs})(p)/ = \mathit{thepred}(\mathit{cs}, p) \wedge && (6.162)\\
&\mathit{myver}(\mathit{cs})(p) <= \mathit{ver}(\mathit{entry}(\mathit{cs}))) && (6.163)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{distinctness\_ok1}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}) \mathrel{\hat{=}} \\
&\forall\, i, j \bullet \mathit{active\_version}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}, i) \wedge && (6.164)\\
&\mathit{active\_version}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}, j) \wedge i/ = j \\
&\Rightarrow \mathit{buffer}(i)/ = \mathit{buffer}(j) && (6.165)\\
& && (6.166)\\
&\mathit{distinctness\_ok2}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}) \mathrel{\hat{=}} && (6.167)\\
&\forall\, p, i \bullet \mathit{active\_version}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}, i) \wedge && (6.168)\\
&\mathit{mods\_new\_node}(\mathit{cs}, p) \Rightarrow \\
&\mathit{sc\_new\_nd}(\mathit{cs})(p)/ = \mathit{buffer}(i) && (6.169)\\
& && (6.170)\\
&\mathit{distinctness\_ok3}(\mathit{cs}) \mathrel{\hat{=}} \\
&\forall\, p, q \bullet \mathit{mods\_new\_node}(\mathit{cs}, p) \wedge \mathit{mods\_new\_node}(\mathit{cs}, q) \wedge p/ = q && (6.171)\\
&\Rightarrow \mathit{sc\_new\_nd}(\mathit{cs})(p)/ = \mathit{sc\_new\_nd}(\mathit{cs})(q) && (6.172)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{distinctness\_ok}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}) \mathrel{\hat{=}} \\
&\mathit{distinctness\_ok1}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}) \wedge && (6.173)\\
&\mathit{distinctness\_ok2}(\mathit{cs}, \mathit{buffer}, \mathit{transferer}, \mathit{setter}, \mathit{releaser}) \wedge && (6.174)\\
&\mathit{distinctness\_ok3}(\mathit{cs}) && (6.175)
\end{aligned}
$$

$$
\begin{aligned}
&ll\_ok(ab, cs) \mathrel{\hat{=}} \\
&\quad \forall p \bullet (pc(cs)(p) = pc_ll\_5 \Rightarrow myver(cs)(p) = ver(ll\_e(cs)(p)) \wedge && (6.176)\\
&\quad (\neg myver(cs)(p) = ver(entry(cs)) && (6.177)\\
&\qquad \vee && (6.178)\\
&\quad mynode(cs)(p) = CURRENT\_PTR(cs, myver(cs)(p)))) && (6.179)\\
&\quad \wedge && (6.180)\\
&\quad (pc(cs)(p) = pc\_resp\_ll \Rightarrow data(h(cs))(mynode(cs)(p)) = val(ab)(p)) && (6.181)\\
& && (6.182)\\
&sc\_ok(ab, cs) eqdef && (6.183)\\
&\quad \forall p \bullet (pc(cs)(p) = pc\_sc\_1 \vee pc(cs)(p) = pc\_sc\_2a \vee pc(cs)(p) = pc\_sc\_2b \vee && (6.184)\\
&\quad pc(cs)(p) = pc\_sc\_2c \vee pc(cs)(p) = pc\_sc\_3 \vee pc(cs)(p) = pc\_sc\_4 && (6.185)\\
&\qquad \Rightarrow sc\_newd(cs)(p) = val(ab)(p)) && (6.186)\\
&\quad \wedge && (6.187)\\
&\quad (mods\_new\_node(cs, p) \Rightarrow \\
&\qquad valid\_pointer(h(cs), sc\_new\_nd(cs)(p))) && (6.188)\\
&\quad \wedge && (6.189)\\
&\quad (pc(cs)(p) = pc\_sc\_2b \vee pc(cs)(p) = pc\_sc\_2c \vee pc(cs)(p) = pc\_sc\_3 \vee && (6.190)\\
&\quad pc(cs)(p) = pc\_sc\_4 \Rightarrow \\
&\qquad data(h(cs))(sc\_new\_nd(cs)(p)) = sc\_newd(cs)(p)) && (6.191)\\
&\wedge && (6.192)\\
&(pc(cs)(p) = pc\_sc\_2c \vee pc(cs)(p) = pc\_sc\_3 \vee && (6.193)\\
&pc(cs)(p) = pc\_sc\_4 && (6.194)\\
&\Rightarrow pred(h(cs))(sc\_new\_nd(cs)(p)) = mynode(cs)(p)) && (6.195)\\
&\wedge && (6.196)\\
&(pc(cs)(p) = pc\_sc\_3 \vee pc(cs)(p) = pc\_sc\_4 && (6.197)\\
&\Rightarrow sc\_new\_nd_p \overset{cs}{\rightarrow} status = INIT\_STAT) && (6.198)\\
&\wedge && (6.199)\\
&(pc(cs)(p) = pc\_sc\_7\_9 \Rightarrow ver(sc\_e(cs)(p)) <= ver(entry(cs)) \wedge && (6.200)\\
&myver(cs)(p) <= ver(sc\_e(cs)(p))) && (6.201)
\end{aligned}
$$

$$\mathit{trans\_ok}(\mathit{cs}, \mathit{transferer}) \mathrel{\hat{=}} \tag{6.202}$$

$$\forall\, p \bullet (\mathit{pc}(\mathit{cs})(p) = \mathit{pc\_trans\_2} \lor \mathit{pc}(\mathit{cs})(p) = \mathit{pc\_trans\_3\_4}) \tag{6.203}$$

$$\Rightarrow p = \mathit{transferer}(\mathit{myver}(\mathit{cs})(p)) \land \tag{6.204}$$

$$\mathit{myver}(\mathit{cs})(p) < \mathit{ver}(\mathit{entry}(\mathit{cs})) \land \tag{6.205}$$

$$\neg \mathit{cs}.\mathit{mynode}_p \xrightarrow{\mathit{cs}} \mathit{status}.\mathit{nlC} \tag{6.206}$$

6.6 Concluding Remarks

The main result of this chapter is a lock-free LL/SC implementation that is space-adaptive, population oblivious and pointer clean. We have applied this implementation to the LFRC technique of the previous chapter, to obtain a general pointer-clean and population oblivious lock-free memory-management technique. We believe that this is the first memory-management technique to possess all of these properties.

The LL/SC implementation presented here depends for its correctness on properties of the memory allocator. If the algorithm is used with a memory allocator that is not lock-free, then the algorithm will not be lock-free in that context. [DG02] presents a lock-free memory allocator that can be used with our algorithm, preserving its advantages. However, it exploits certain system properties that are not widely available. [DHLM04] presents a lock-free freelist that is population oblivious, pointer clean and space-adaptive (it can safely release memory back to the system when the memory is no longer required by a client application). Because of these properties, it can be used to manage memory buffers for our LL/SC implementation.

Chapter 7

Conclusions

In this concluding chapter we evaluate the contributions presented in this thesis, and discuss possibilities for future work. The contributions of this thesis are divided into two categories. Part I is concerned with the verification of nonblocking algorithms, and Part II is concerned with their design. Inverting the order of the thesis's parts, and reverting to the order of the title, Section 7.1 evaluates the work of Part II, and Section 7.2 evaluates the work of Part I. Section 7.3 describes a new verification methodology known as *separation logic* that seems very promising. Separation logic has been used in the verification of nonblocking algorithms, and addresses modelling issues that have been ignored in our framework. Separation logic is a member of a family of verification techniques originating with the work of Floyd [Flo67] and Hoare [Hoa69], which we refer to as *axiomatic* techniques, and describe by way of background in Section 7.3. Section 7.4 discusses verification issues raised by *transactional memory*, a technique for constructing concurrent implementations of shared objects that has recently attracted a great deal of interest in the nonblocking algorithms community and beyond.

7.1 Nonblocking Algorithms

The LL/SC algorithm presented in Chapter 6 is the first nonblocking algorithm that is space-adaptive, population oblivious and pointer-clean. Furthermore, in combination with the LFRC technique of Chapter 5, the LL/SC algorithm can be used to make any garbage collection dependent algorithm space adaptive, population oblivious and pointer-clean. These are important properties for practical algorithms. However, the techniques presented in Part II require at least one atomic modification of a counter value for every operation; some operations require several such modifications. For this reason, it is likely that our LL/SC implementation will perform worse than some other proposals that do not enjoy the same generality. It is very likely that the most efficient existing techniques for obtaining space-adaptive algorithms are the guard based proposals of [HLM02b, HLMM05] and [Mic02, Mic04]. As noted in Chapter 5, these techniques are not themselves space-adaptive, and in some versions, are not popu-

lation oblivious. However, it seems that they could be made space adaptive and population oblivious using the techniques of Part II. It is fairly straightforward to achieve population obliviousness. [HLM03b] and [Mic04] both present techniques for achieving this. The idea is that rather than using an array of guards, which must be allocated with a particular size (as originally proposed in [HLM02b, Mic02]), guards are allocated in a linked-list which can be dynamically expanded. [HLM03b] extends this approach with a reference-counting technique that allows guards to be deallocated when they are no longer needed. As mentioned in Chapter 5, this technique enables the construction of weakly space-adaptive algorithms.

Thus, the goal is to obtain a solution that is strongly space-adaptive, while preserving the performance properties of the guard-based proposals. We can use the techniques of Part II to construct a linked-list from which deallocation is possible, even in the presence of process failures. Traversing this list would be at least as expensive as the technique presented in [HLM03b], but using the guards within application code would be as cheap as the original proposals [HLM02b, Mic04]. However, a traversal of the list of guards must occur whenever any memory is to be freed, so it would be useful to optimise this step. One appealing possibility is to have two linked lists of guards. The first, which we call the *primary* list, is used for application purposes. The second, which we call the *auxiliary* list, is used to safely traverse the primary list, while enabling deallocation from the primary list. Thus, guards in the auxiliary list would be used to protect nodes in the primary list. Because guards in the auxiliary list are only used during operations on one data structure (the primary list), and the operations of this data structure can be implemented using a statically known number of guards,[1] each process needs only a statically determined number of guards in the auxiliary list.

In this scheme, traversal of the primary list would be possible without manipulating counts, and would thus recover much of the efficiency of the array-based approach. Traversal of the auxiliary list would depend on the techniques of Part II, and would thus be significantly slower. However, traversals of the auxiliary list would occur much less frequently under expected loads. This is because the auxiliary list only needs to be traversed when guards from the primary list are destroyed. We would expect these events to be substantially less frequent than the deallocation of memory by the application itself.

One interesting verification issue raised by these algorithms is the question of verifying space-adaptivity. Space-adaptivity is a safety property, like an invariant, and is therefore proved by induction on the length of executions. However, trace inclusion does nothing to capture space adaptivity. The idea would be to introduce a function that measured space usage in each state, and prove that in all reachable states, this function sits within an appropriate bound.

[1] The necessary operations, inserting a new guard, removing a guard, and traversing the list collecting the guards, all require at most three guards.

7.2 Verification

The verifications presented in this thesis are based on labelled-transition systems and simulation relations. Labelled-transition systems are very general structures that have been used to model a broad class of computer systems. This enables the construction of full behavioural verifications of unbounded or infinite state models within a uniform setting. Furthermore, because labelled-transition systems and their properties can be expressed in ordinary mathematical notation, it is straightforward to express the model in the language of a proof checker, and so to construct formal and precise proofs.

Simulation relations themselves provide additional advantages. There is a natural correspondence between the linearisation points of an algorithm, and the step-correspondence of the simulation relation used to verify the algorithm. Furthermore, simulation relations can deal with unusual or complicated linearisation points. For example, the verification in Chapter 6 has executions where the step of one process can be the linearisation point of another process. A more important example is the question of delayed serialisation: backward simulation provides a natural way to treat delayed serialisation. These unusual patterns of linearisation seem to be very important in nonblocking algorithms.

However, there are significant disadvantages to fully deductive verification based on labelled-transition systems and simulation relations. One prominent problem is the issue of coding: the models presented in this thesis are large and complex, and this size and complexity affects the simulation relations as well. This coding is tedious and (in the absence of automation) error prone, and the loss of syntactic structure makes the verification difficult. For example, our models contain no information about the scope of variables, and information about allocation of references that is obvious in the pseudocode is lost.

Another problem is that the human effort and skill needed to construct a formal proof is substantial. As discussed in Sections 2.1 and 3.6, many verification techniques employ automatic methods, which can verify properties of systems without human intervention within a few seconds or hours. The price of the generality of labelled-transition systems is the extra effort required in verifying their properties.

Sections 7.2.1 and 7.2.2 describe further limitations of our approach, and make tentative suggestions as to how they might be addressed. In particular, Section 7.2.1 describes difficulties associated with representing the composition of shared-memory objects in the I/O automaton framework, and Section 7.2.2 outlines the relevance of relaxed consistency models to nonblocking algorithms and their verification.

7.2.1 Compositionality

Currently, we do not make use of the facilities for composing automata provided in the I/O automaton framework. The notion of composition defined in this framework is not well suited to reasoning about collections of objects in shared-memory. Rather, it is designed to model the composition of distributed processes that communicate via message passing. This focus on distributed systems has two important drawbacks for using the same notion of

composition in a shared-memory setting.

First, objects in shared memory share their state. At least, they share the same heap and address space. Also, several objects may share the same freelist or other supporting data structures. Because of this, the behaviour of a composition of objects depends on how each object manipulates this shared state. This is quite different to a message passing system, where the state of each object (process) is isolated from the states of all other objects.

Second, the compositionality principle for I/O automata requires that the composed automata be input-enabled. This condition seems unnatural for automata that model objects in shared memory. In a shared-memory context, when a process invokes an operation on some object, that process is guaranteed to do nothing else until the operation is complete. Therefore, the input actions (invocations) are simply not enabled in all states.

Both these issues can be addressed within the I/O automaton framework, but with some cost. The second issue is perhaps more straightforward than the first. It is possible to define the transition relations of automata that represent datatypes so that input (i.e., invocation) actions are input enabled, and require that these actions are invoked by a process only when that process has no pending operations. This amounts to placing a constraint on the environment of the automaton. Such an approach is used in [Lyn96, Section 13.2]. One drawback is that it becomes necessary to prove that whenever an input action occurs on a client automaton, there is no pending operation in the datatype automaton.

A more important issue is that we still have no guidance about how to deal with the fact that different data structures may share the same heap, or other state. Note that the specification of a shared-memory object will frequently not mention heap operations. These are typically not visible in the specification and are hidden by its implementations. (In this thesis, we have achieved this hiding by making actions that correspond to reads, writes and allocations into internal actions.) However, in order to compose one shared memory object with another, it is necessary to have some guarantee about how each object will manipulate the heap. This means that our specifications would need to include such information, and our composition rule would need to exploit this information.

The work discussed in Section 7.3 addresses the issue of compositionality more directly.

7.2.2 Relaxed Consistency Models

One important issue that has not been treated in this thesis is the question of relaxed consistency models. We have assumed that all operations on shared memory are atomic. This assumption is not satisfied by most implementations of shared memory. The impact of operation reordering is more important in nonblocking algorithms than algorithms that use mutual exclusion. This is because primitives that support mutual exclusion, such as locks, typically implement semantics guaranteeing that if every location is only accessed under mutual exclusion, then all read and write operations will appear to be sequentially consistent. Such semantics can be expressed by defining a constraint on code, such that code satisfying the constraint is guaranteed to behave as if the underlying memory were sequentially consistent

[AH90, MPA05].[2] Unfortunately, nonblocking algorithms frequently do not satisfy data-race freedom constraints. Therefore, when implementing a nonblocking algorithm on real hardware, the programmer must be aware of reorderings allowed by the memory model. So the need to deal directly with relaxed consistency models in the verification of nonblocking algorithms is more pressing than in lock-based, shared-memory algorithms.

[CLMT05] uses an I/O automaton model, referred to as a *partial-order machine*, to represent a shared memory system conforming to a specific relaxed consistency model. It would be interesting to use such an automaton as the basis for models of nonblocking algorithms running over shared memory systems exhibiting various flavours of relaxed consistency. However, constructing a plausible model is quite different to completing a proof of correctness. It is unclear whether the method of transition systems and simulation relations would be effective for constructing formal verifications.

We return to the issue of relaxed consistency models in Section 7.3.

7.3 Axiomatic Approaches

This thesis has used labelled-transition systems as models for concurrent shared-memory systems. However, there is a tradition of using *axiomatic semantics* of programming languages as the basis for modelling and verifying shared-memory concurrent systems. Axiomatic semantics, as applied to sequential systems, originates with the work of Floyd [Flo67] and Hoare [Hoa69]. (Notations that use the axiomatic style are sometimes referred to as *Hoare logics*.) There have been several proposals for extending the axiomatic approach to concurrent systems, for example [Hoa73, OG76, LS84], and recently there has been a flurry of work in this area [Bro04, O'H07, VP07] (based on [IO01, Rey02]). In this section, we outline the axiomatic approach and describe recent advances, and we evaluate the advantages and disadvantages of these approaches relative to our transition-system based techniques.

In the axiomatic approach, the effect (or meaning) of a program is described using pre- and postconditions. That is, assertions of the form "if the values of the program variables satisfy P before the program is executed, then they satisfy Q when the program terminates".[3] Such assertions are written formally as

$$\{P\}S\{Q\}$$

where S is the program in question. Here P is referred to as the precondition and Q as the postcondition. Normally, P and Q are written in some version of first-order logic. Such assertions can be used to formalise the semantics of programming languages and also of data structures and their operations. Specifications of data structures typically involve the use of auxiliary variables.

[2] Real memory models tend to provide additional guarantees about the possible behaviour of memory accesses. We ignore that detail here.

[3] We ignore here the possibility that our programs, or their constituent commands, fail to terminate.

One advantage of the axiomatic approach is that the effect of each program is specified in terms of the effect of its component parts, thus exploiting the structure of the syntax. For example, if we know that the program S is such that $\{P\}S\{Q\}$ and the program T is such that $\{Q\}T\{R\}$ then we can conclude that the effect of the *sequential composition* of S and T is $\{P\}S;\ T\{R\}$.

The fact that we are axiomatising the effect of a program using its syntactic structure provides one of the most important advantages of the axiomatic approach over our labelled-transition system techniques. The complicated and error prone encoding into labelled-transition systems is unnecessary. Furthermore, in the axiomatic approach it is possible to exploit the structure of code to generate the properties (or invariants) that are needed to make the proof work.

Perhaps the most important early attempt at applying the axiomatic approach to concurrent programs is the work of Owicki and Gries [OG76]. Unfortunately, their techniques, along with others (e.g., [Hoa73]), did not address the issue of pointers, and the possibility of aliasing that they introduce. Furthermore, they did not deal directly with dynamic memory allocation. However, a new approach known as *separation logic* [Rey02, O'H07] attempts to address these issues.

Syntactically, the key aspect of separation logic is its use of a connective known as *separating conjunction* that behaves like ordinary conjunction in propositional logic, except that each of its arguments refers to (is about) disjoint parts of the heap. This interpretation is enforced by the proof rules governing separating conjunction. Separating conjunction allows the statement and proof of heap properties, without needing to explicitly state properties about aliasing relationships or reachability.

Separation logic has been used to verify a version of the Treiber stack [PBO07], as well as several other sequential and concurrent algorithms ([O'H07] presents examples, and provides pointers to other verifications in the literature).

One appealing aspect of this work is that a form of composition can be directly achieved using separating conjunction, even in the presence of shared heap state. Because each argument of a separating conjunction is about disjoint parts of the heap, each separating conjunction guarantees that the heap operations of two objects that satisfy the specification cannot interfere.

Mechanical assistance for theorem proving in separation logic is at a preliminary stage. An application called *Smallfoot* [BCO06, SmF] can be used to verify both sequential and concurrent code using separation logic. However, as yet Smallfoot does not provide any facilities for human interaction: proofs must be found automatically by the application or not at all. Furthermore, there are limited definitional facilities. Predicates describing singly- and doubly-linked lists are hardwired into the application, so as to allow nontrivial verifications. Inductive definitions cannot be directly expressed in Smallfoot's specification language. It is likely that these limitations can be overcome, given sufficient attention. However, it seems likely that useful mechanical assistance for proofs of significant size will not be available until the work on Smallfoot has advanced substantially.

7.4 Transactional Memory

Much of the recent work in the field of nonblocking algorithms is concerned with the development and use of *transactional memory* [HM93]. Much of this work has focussed on hardware implementations of transactional memory [AAK+05, MBM+06, BMV+07], but there are several software-only implementations [ST95, HLM02a, HF03, DSS06, SATH+06], and implementations that use hardware transactional-memory primitives, augmented by software [DFL+06, MTC+07]. The implementations that involve a software aspect are frequently as complicated as nonblocking implementations of standard data structures. Thus, they present interesting and important verification challenges.

However, there are important questions about the semantics of transactional memory, among them:

- How do transactional operations interact with nontransactional operations? For example, can a transaction observe writes executed by a process not executing a transaction?
- How should exceptions thrown during a transaction be propagated?
- What progress guarantees are desirable, and in what situations? Must the system guarantee progress to each transaction? Or is progress on a system wide basis acceptable (as with lock-freedom)? Are probabilistic guarantees acceptable? Must transactions be able to survive across page-faults or descheduling of the process executing the transaction?

These issues, which are the subject of recent work (e.g., [MG08, ABHI08]) are of particular relevance to the specification of transactional memory systems. [MG08, ABHI08] do not use transition systems (at least, not of the kind used in this thesis) in the specification of transactional memory. Rather, they apply techniques developed to provide operational semantics for programming languages. Therefore, using labelled-transition systems such as I/O automata to describe the semantics of transactional memory might be interesting in itself. Another possibility is to adapt the techniques of operational semantics to define I/O automata representing transactional memory. In any case, the development of techniques for specifying transactional memory systems such that their implementations can be naturally verified is an important goal.

The growth in popularity of transactional memory may increase the need for nonblocking memory management solutions of the kind discussed in Part II of the thesis, and Section 7.1 of this chapter. Most proposed transactional memory implementations that depend on software (whether software only or hybrid) depend for their correctness on the system not releasing memory that may be accessed by a delayed transaction ([DSS06] is one exception). For this reason, many such proposals assume the presence of a garbage collector. However, if software or hybrid transactional memory is to be applicable outside of garbage collected environments, efficient and correct concurrent memory management techniques must be found.

Bibliography

[AAK+05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[ABHI08] Martín Abadi, Andrew Birrell, Tim Harris, and Michael Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 63–74, New York, NY, USA, 2008. ACM.

[Abr96] Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, October 1996.

[Abr03] Jean-Raymond Abrial. Event based sequential program development: Application to constructing a pointer program. In *Formal Methods Europe*, Lecture Notes in Computer Science. Sringer-Verlag, September 2003.

[AC05] Jean-Raymond Abrial and Dominique Cansell. Formal construction of a non-blocking concurrent queue algorithm (a case study in atomicity). *Journal of Universal Computer Science*, May 2005.

[ACM03] Jean-Raymond Abrial, Dominique Cansell, and D. Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3):215–227, April 2003.

[ADF+00] Ole Agesen, David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir Shavit, and Guy L. Steele Jr. DCAS-based concurrent deques. In *ACM Symposium on Parallel Algorithms and Architectures*, pages 137–146. ACM Press, 2000.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering—a new definition. *SIGARCH Comput. Archit. News*, 18(3a):2–14, 1990.

[AHR00] Myla Archer, Constance Heitmeyer, and Elvinia Riccobene. Using TAME to prove invariants of automata models: Two case studies. In *Formal Methods in Software Practice*, pages 25–36. ACM Press, 2000.

[AKW88] A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK Programming Language*. Addison Wesley, 1988.

[AL91] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82, May 1991.

[AM95] James H. Anderson and Mark Moir. Universal constructions for multi-object operations. In *Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 184–193. ACM Press, 1995.

[AM99] James Anderson and Mark Moir. Universal constructions for large objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317–1332, 1999.

[ARR+07a] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In *Proceedings of the 19th Conference on Computer Aided Verification (CAV 2007)*, 2007.

[ARR+07b] Daphna Amit, Noam Rinetzky, Thomas Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *Computer Aided Verification (CAV)*, Lecture Notes in Computer Science. Springer-Verlag, August 2007.

[Bac98] Jean Bacon. *Concurrent Systems: Operating Systems, Database and Distributed Systems—an integrated approach*. Addison-Wesley, 2nd edition, 1998.

[BAM06] S. Burckhardt, R. Alur, and M. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In *Proceedings of the 18th International Conference of Computer Aided Verification*, 2006.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Checkfence: Checking consistency of concurrent datatypes on relaxed memory models. In *Proceedings of the ACM COnference on Programming Language Design and Implementation (PLDI)*, 2007.

[BCDR04] Thomas Ball, Byron Cook, Satyaki Das, and Sriram K. Rajamani. Refining approximations in software predicate abstraction. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2004.

[BCO06] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Proceedings of the 5th International Symposium on Formal Methods for Components and Objects*, 2006.

[BGLR01] Elizabeth Borowsky, Eli Gafni, Nancy Lynch, and Sergio Rajsbaum. The BG distributed simulation algorithm. *Distributed Computing*, 14(3):127–146, July 2001.

[Blo88] B. Bloom. Constructing two-writer atomic registers. *IEEE Transactions on Computers*, 37(12):1506–1514, 1988.

[BMV+07] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. Performance pathologies in hardware transactional memory. *SIGARCH Comput. Archit. News*, 35(2):81–91, 2007.

[Bro04] S. Brookes. A semantics for concurrent separation logic. In *Proceedings of the 15th CONCUR*, August 2004.

[But97] David R. Butenhof. *Programming with POSIX threads*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1997.

[CBHLL92] J. Chase, M. Baker-Harvey, H. Levy, and E. Lazowska. Opal: A single address space system for 64-bit architectures (abstract). *Operating Systems Review*, 26(2), 1992.

[CDG05] Robert Colvin, Simon Doherty, and Lindsay Groves. Verifying concurrent data structures by simulation. In Eerke Boiten and John Derrick, editors, *Proceedings of the RefineNet Workshop 2005 (REFINE 2005), Guildford, UK*, Electronic Notes in Theoretical Computer Science, April 2005.

[CE82] E.M. Clarke and E.A. Emerson. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Trans. Program. Lang. Syst.*, 8(2):244–263, 1986.

[CG05] Robert Colvin and Lindsay Groves. Formal verification of an array based nonblocking queue. In *Proceedings of the 10th International Conference on Engineering of Complex Computer Systems,(ICECCS'05)*, June 2005.

[Chr84] T. W. Christopher. Reference count garbage collection. *Software — Practice and Experience*, 14(6):503–508, 1984.

[CLMT05] Gregory Chockler, Nancy Lynch, Sayan Mitra, and Joshua Tauber. Proving atomicity: An assertional approach. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'2005)*, pages 152–168, 2005. Published by Springer-Verlag GmbH in Lecture Notes in Computer Science, Volume 3724, Oct 2005, Pages 152–168.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation*. Adison Wesley, 1988.

[Col60] G. E. Collins. A method for overlapping and erasure of lists. *Communications of the ACM*, 3(12):655–657, December 1960.

[COR+95] Judy Crow, Sam Owre, John Rushby, Natarajan Shankar, and Mandayam Srivas. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, April 1995.

[DB76] L. P. Deutsch and D. G. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM*, 19(9):522–526, 1976.

[DD02] Satyaki Das and Devid L. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, 2002.

[DDG+04] Simon Doherty, David Detlefs, Lindsay Groves, Christine Flood, Victor Luchangco, Paul Martin, Mark Moir, Nir Shavit, and G. L. Steele Jr. DCAS is not a silver bullet for nonblocking synchronization. In *Proceedings of the Sixteenth ACM Symposium on Parallelism in Algorithm s and Architectures*. ACM Press, June 2004.

[DeT90] John DeTreville. Experience with garbage collection for modula-2+ in the topaz environment. In OOPSLA/ECOOP '90 Workshop on Garbage Collection in Object-Oriented Systems, October 1990.

[DFG+00] David Detlefs, Christine H. Flood, Alex Garthwaite, Paul Martin, Nir N. Shavit, and G. L. Steele Jr. Even better DCAS-based concurrent deques. In *Proceedings of the 14th International Conference on Distributed Computing*, pages 59–73. IEEE Computer Society Press, 2000.

[DFL+06] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, 2006.

[DG02] Dave Dice and Alex Garthwaite. Mostly lock-free malloc. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*. ACM Press, 2002.

[DGLM04] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Formal verification of a practical lock-free queue algorithm. In *LNCS 3235: Formal Techniques for Networked and Distributed Systems - FORTE 2004*. Springer Verlag, September 2004.

[DHLM04] Simon Doherty, Maurice Herlihy, Victor Luchangco, and Mark Moir. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the*

23rd Annual ACM Symposium on Principles of Distributed Computing. ACM Press, July 2004.

[DMMm01] David Detlefs, Paul Martin, Mark Moir, and Guy L. Steele, Jr. Lock-free reference counting. In *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, August 2001.

[Doh03] Simon Doherty. Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, School of Mathematical and Computing Sciences, Victoria University, Wellington, New Zealand, 2003.

[dREB98] Willem Paul de Roever, Kal Engelhardt, and Karl Heinz Buth. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*, volume 47 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.

[dSp] Website of dSpin: a model checker for programs, http://www-verimag.imag.fr/ iosif/dspin/.

[DSS06] Dave Dice, Ori Shalev, and Nir Shavit. Transactional locking II. *Distributed Computing*, 4167/2006:194–208, 2006.

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J.T Schwartz, editor, *Proceedings of a Symposium on Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[FOL05] Dominique Fober, Yann Orlarey, and Stephan Letz. Optimized lock-free fifo queue continued. Technical Report TR-050523, Laboratoire de Recherche en Informatique Musicale, 2005.

[Fra03] Keir Fraserback. *Practical Lock-Freedom*. PhD thesis, King's College, 2003.

[Gao05] Hui Gao. *Design and Verification of Lock-free Parallel Algorithms*. PhD thesis, University of Groningen, Netherlands, 2005.

[GGH05a] Hui Gao, J F Groote, and W H Hesselink. Lock-free parallel garbage collection. In *Proceedings of the Third International Symposium of Parallel and Distributed Processing and Applications, LNCS 3758*. Springer, 2005.

[GGH05b] Hui Gao, J.F Groote, and W. H. Hesselink. Lock-free resizable hash-tables with open addressing. *Distributed Computing*, 18(1), July 2005.

[GL00] Stephen J. Garland and Nancy A. Lynch. Using I/O automata for developing distributed systems. In Gary T. Leavens and Murali Sitaraman, editors, *Foundations of Component-Based Systems*, pages 285–312. Cambridge University Press, 2000.

[GLV01] Stephen J. Garland, Nancy A. Lynch, and Mandana Vaziri. *IOA: A Language for Specifying, Programming and Validating Distributed Systems*. MIT Laboratory for Computer Science, October 2001.

[Gre96] Michael Greenwald. The synergy between non-blocking synchronization and system design. In *Proceedings of the Second USENIX Symposium on Operating System Design and Implementation*, pages 123–136. ACM Press, 1996.

[Gre99] Michael Barry Greenwald. *Non-Blocking Synchronisation and System Design*. PhD thesis, Stanford University, August 1999.

[Gre02] Michael Greenwald. Two-handed emulation: How to build non-blocking implementations of complex data-structures using DCAS. In *Proceedings of the twenty-first annual symposium on Principles of Distributed Computing*. ACM Press, 2002.

[Gro08] Lindsay Groves. Verifying Michael and Scott's lock-free queue algorithm using trace reduction. In *CATS '08: Proceedings of the fourteenth symposium on Computing: the Australasian Theory Symposium*, pages 133–142, Darlinghurst, Australia, Australia, 2008. Australian Computer Society, Inc.

[GS97] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In *Proceedings of the Ninth annual Conference of Computer Aided Verification*, pages 72–83. Springer-Verlag, 1997.

[Har01] T. Harris. A pragmatic implementation of non-blocking linked lists. In *Proceedings of the 15th Annual Symposium on Distributed Computing*, pages 300–314. IEEE Computer Society Press, October 2001.

[Hei91] Joe Heinrich. *MIPS R4000 Microprocessor User's Manual*. MIPS Computer Systems, Inc., 1991.

[Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[Hes02a] Wim H. Hesselink. An assertional criterion for atomicity. *Acta Informatica*, 38:342–366, December 2002.

[Hes02b] Wim H. Hesselink. Eternity variables to simulate specifications. In *Mathematics of Program Construction 2002, volume 2386 of LNCS*, pages 117–130. Springer, 2002.

[Hes05] Wim H. Hesselink. Eternity variables to prove simulation of specifications. *ACM Transactions on Computational Logic (TOCL)*, 6(1):175–201, January 2005.

[HF03] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-oriented Programming, Systems, Languages and Applications*, pages 388–402. ACM Press, October 2003.

[HHL+06] Steve Heller, Maurice Herlihy, Victor Luchangco, Mark Moir, William N. Scherer, and Nir Shavit. A lazy concurrent list-based set algorithm. In *Principles of Distributed Systems*, 2006.

[HHS86] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In *ESOP '86: Proceedings of the European Symposium on Programming*, pages 187–196, London, UK, 1986. Springer-Verlag.

[HLM02a] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free software NCAS and transactional memory. Unpublished manuscript, Sun Microsystems Laboratories, Burlington, Massachusetts, 2002.

[HLM02b] M.P. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of 16th International Symposium on Distributed Computing*, October 2002.

[HLM03a] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003.

[HLM03b] Maurice Herlihy, Victor Luchangco, and Mark Moir. Space- and time-adaptive nonblocking algorithms. In *Proceedings of Computing: The Australasian Theory Symposium*, Electronic Notes in Computer Science. Elsevier Science, 2003.

[HLMM02] Maurice Herlihy, Victor Luchangco, Paul Martin, and Mark Moir. Dynamic-sized lock-free data structures. In *ACM Symposium on Principles of Distributed Computing*, page 131. ACM Press, 2002.

[HLMM05] Maurice Herlihy, Victor Luchangco, Paul Marting, and Mark Moir. Nonblocking memory management support for dynamic sized data structures. *ACM Transactions of Computer Systems*, 23, May 2005.

[HLMS03] M. Herlihy, V. Luchangco, M. Moir, and W.N. Scherer. Software transactional memory for dynamic-sized data structures. In *Twenty-Second Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, July 2003.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969.

[Hoa73] C. A. R. Hoare. Parallel programming: an axiomatic approach. Technical Report CS-73-394, Stanford University, Stanford, CA, USA, 1973.

[Hol97] Gerard J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[HW87] Maurice P. Herlihy and Jeannette M. Wing. Axioms for concurrent objects. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 13–26. ACM Press, January 1987.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, November 1990.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, 2001.

[isa] The Isabelle Theorem Proving Environment, http://www.cl.cam.ac.uk/research/hvg/isabelle/.

[Jav] The ConcurrentLinkedQueue class of java.util.concurrent, http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.

[Jay05] Prasad Jayanti. Efficiently implementing a large number of ll/sc objects. Technical Report TR2005-554, Dartmouth College Computer Science Department, 2005.

[JL96] Richard Jones and Radal Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons Ltd., 1996.

[JP03] Prasad Jayanti and S. Petrovic. Efficient and practical constructions of LL/SC variables. In *Twenty-second Annual ACM Symposium on Principles of Distributed Computing*, pages 285–294. ACM Press, July 2003.

[JP05] Prasad Jayanti and Srdjan Petrovic. Efficiently implementing LL/SC objects shared by an unknown number of processes. In *Distributed Computing IWDC 2005*. Springer Berlin/Heidelberg, 2005.

[JP07] Prasad Jayanti and Srdjan Petrovic. Efficiently implementing a large number of LL/SC objects. In *Twenty-Sixth Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*. Springer Berlin/Heidelberg, 2007.

[JSR] Java Specification Request for Concurrent Utilities (JSR 166). http://jcp.org.

[JTL] The ThreadLocal class of the java.lang package.

[Lam94] Leslie Lamport. The temporal logic of actions. In *ACM Transactions of Programming Languages and Systems*, pages 872–923. ACM Press, May 1994.

[Lea00] Doug Lea. *Concurrent Programming in Java: design principles and patterns*. Addison-Weslyer, 2nd edition, 2000.

[Lee07] Jeremy Lee. Practical multiwriter lock-free queues for ”hard real-time” systems without CAS. Availabel from arxi.org at http://arxiv.org/abs/0709.4558v1, September 2007.

[LMS03a] Victor Luchangco, Mark Moir, and Nir Shavit. Nonblocking k-compare-single-swap. In *Fifteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 314–323. ACM Press, 2003.

[LMS03b] Victor Luchangco, Mark Moir, and Nir Shavit. On the uncontended complexity of consensus. In *DISC*, 2003.

[LP] LP, the Larch Prover, http://nms.lcs.mit.edu/larch.

[LP01] Yossi Levanoni and Erez Petrank. An on-the-fly reference-counting garbage collector for Java. In *the ACM Conference on Object-Oriented Programming Languages and Systems (OOPSLA)*, October 2001.

[LS84] Leslie Lamport and Fred B. Schneider. The ”Hoare Logic” of CSP and all that. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 6(2):281–296, 1984.

[LT87] N. Lynch and M. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Distributed Computing*, pages 137–151. ACM Press, August 1987.

[LV93] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations – part I: untimed systems. Technical Report CS-R9313, Centrum voor Wiskunde en Informatica (CWI), 1993.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., 1996.

[MBM+06] K. E. Moore, J. Bobba, M. J. Moravan, M.D. Hill, and D. A. Wood. LogTM: log-based transactional memory. In *Proceedings of the Twelfth International Symposium on High-performance Computer Architecture*, pages 254–265, February 2006.

[met] The MetaPRL proof assistant and logical programming environment, http://metaprl.org/default.html.

[MG08] Katherine F. Moore and Dan Grossman. High-level small-step operational semantics for transactions. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 51–62, New York, NY, USA, 2008. ACM.

[Mic02] Maged Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, July 2002.

[Mic04] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems*, 15(6), June 2004.

[MMm02] Paul A. Martin, Mark Moir, and Guy L. Steele, Jr. Better still DCAS-based concurrent deques. Technical Report TR-2002-111, Sun Microsystems Laboratories, 2002.

[MNSS05] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. In *Proceedings of the 17th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 253–262. ACM Press, 2005.

[Moi97] Mark Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, Santa Barbara, CA.*, August 1997.

[Moi00] Mark Moir. Laziness pays! Using lazy synchronization mechanisms to improve non-blocking constructions. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 61–70. ACM Press, 2000.

[Mot93] Motorola Inc. *PowerPC 601 RISC Microprocessor User's Manual*, 1993.

[MP91] H. Massalin and C. Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-01, Computer Science Department, Columbia University, October 1991.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. *ACM SIGPLAN Notices*, 40(1):378–391, 2005.

[MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, 1995.

[MS96a] M. M. Michael and M. L. Scott. Executable code - fast concurrent queue algorithms, 1996. Code for the M&S

queue. Available from ftp://ftp.cs.rochester.edu/pub/packages/sched_conscious_synch/concurrent_queues.tar.gz.

[MS96b] M. M. Michael and M. L. Scott. Simple, fast and practical nonblocking and blocking concurrent queue algorithms. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 267–275. ACM Press, 1996.

[MS98a] M. M. Michael and M. L. Scott. Nonblocking algorithms and preemption safe locking on multiprogrammed shared memory multiprocessors. *Journal of Parallel and Distributed Computing*, 51:1–26, 1998.

[MS98b] Maged Michael and Michael Scott. Relative performance of preemption-safe locking and non-blocking synchronization on multiprogrammed shared memory multiprocessors. In *Proceedings of the 11th International Symposium on Parallel Processing*. IEEE Computer Society, 1998.

[MTC+07] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 69–80, New York, NY, USA, 2007. ACM.

[Mur] The Murphi description language and protocol verifier, http://verify.stanford.edu/dill/murphi.html.

[MWL90] A. D. Martinez, R. Wachenchauzer, and R. D. Lins. Cyclic reference counting with local mark-scan. *Information Processing Letters*, 34(1):31–35, 1990.

[MYRS05] Roman Manevich, Eran Yahav, G. Ramalingam, and Mooly Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In Radhia Cousot, editor, *Proceedings of the 6th International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI 2005*, Lecture Notes in Computer Science. Springer, January 2005.

[OG76] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta informatica*, 6(4):319–340, 1976.

[O'H07] Peter W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, 2007.

[PBK+05] Harel Paz, David F. Bacon, Elliot K. Kolodner, Erez Petrank, and V. T. Rajan. Complete on-the-fly cycle detection. In *Proceedings of the 14th International Conference on Compiler Construction (CC'05)*, April 2005.

[PBO07] Matthew Parkinson, Richard Bornat, and Peter O'Hearn. Modular verification of a non-blocking stack. In *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 2007. New York, NY, USA.

[PLJ94] Sundeep Prakash, Yann Hang Lee, and Theodore Johnson. A nonblocking algorithm for shared queues using compare-and-swap. *IEEE Transactions on Computers*, 43(5), May 1994.

[PVS] The PVS Specification and Verification System, http://pvs.csl.sri.com/.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, London, UK, 1982. Springer-Verlag.

[Rei04] William K. Reinholtz. Atomic reference counting pointers. Dr. Dobb's Journal. Availiable from http://www.ddj.com/cpp/184401888, November 2004.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS '02: Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, Washington, DC, USA, 2002. IEEE Computer Society.

[RR00] J. Antonio Ramírez-Robredo. Paired simulation of I/O automata. Master's thesis, Massachusetts Institute of Technology, September 2000.

[SAGG+93] Jørgen F. Søgaard-Andersen, Stephen J. Garland, John V. Guttag, Nancy A. Lynch, and Anya Pogosyants. Computer-assisted simulation proofs. In Costas Courcoubetis, editor, *Computer-Aided Verification, Fifth International Conference, CAV '93, Elounda, Greece, Lecture Notes in Computer Science 697*, pages 305–319. Springer-Verlag, 1993.

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[Sha93] A. Udaya Shankar. An introduction to assertional reasoning for concurrent systems. *ACM Computing Surveys*, 25(3), September 1993.

[Sit92] Richard L. Sites. *Alpha Architecture Reference Manual*, 1992.

[SmF] Website for the Smallfoot proof assistant, http://www.dcs.qmul.ac.uk/research/logic/theory/projects/smallfoot/.

[Smi96] M. A. S. Smith. Formal verification of communication protocols. In *FORTE*, 1996.

[Smi97] Mark Anthony Shawn Smith. *Formal Verifcation of TCP and T/TCP*. PhD thesis, Massachusetts Institute of Technology. Dept. of Electrical Engineering and Computer Science, 1997.

[SMV] Model checking at CMU, http://www-2.cs.cmu.edu/ modelcheck/.

[Spi] Model checking with Spin, http://spinroot.com/spin/whatispin.html.

[SS03] Ori Shalev and Nir Shavit. Split-ordered lists: lock-free extensible hash tables. In *Twenty-second Annual Symposium on Principles of Distributed Computing*, pages 102–111. ACM Press, 2003.

[ST95] Nir Shavit and Dan Touitou. Software transactional memory. In *ACM Symposium on Principles of Distributed Computing*, pages 204–213. ACM Press, 1995.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center, April 1986.

[TSP92] John Turek, Dennis Shasha, and Sundeep Prakash. Locking without blocking: making lock based concurrent data structure algorithms nonblocking. In *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 212–222, 1992.

[TZ01a] Philippas Tsigas and Yi Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*. ACM Press, 2001.

[TZ01b] Philippas Tsigas and Yi Zhang. A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 134–143. ACM Press, 2001.

[Val94] John Valois. Implementing lock-free queues. In *Proceedings of the 7th International Conference on Parallel and Distributed Computing Systems*, October 1994.

[Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, August 1995.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of Concur 2007 – Concurrency Theory*, volume 4703/2007 of *Lecture Notes in Computer Science*, pages 256–271. Springer, Heidelberg, 2007.

[WD96] Jim Woodcock and Jim Davies. *Using Z. Specification, Refinement and Proof.* Prentice Hall International Series in Computer Science. Prentice Hall, 1996.

[WG94] D. Weaver and T. Germond. *The SPARC Architecture Manual Version 9*, 1994.

[Wis93] D. S. Wise. Stop-and-copy and one-bit reference counting. *Information Processing Letters*, 45(5):243–249, July 1993.

[WS91] L. Wall and R. L. Schwarz. *Programming Perl*. O'Reilly and Associates Inc., 1991.

[WS02] Farn Wang and K. Schmidt. Symmetric symbolic safety-analysis of concurrent software with pointer data structures. In *IFIP FORTE, LNCS 2529*. Springer-Verlag, November 2002.

[WS05] Liqiang Wang and Scott D. Stoller. 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, 2005. ACM.

[Yah01] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3):27–40, March 2001.

[YML99] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications, 1999.

[YS03] Eran Yahav and Shmuel Sagiv. Automatically verifying concurrent queue algorithms. *Electr. Notes Theor. Comput. Sci.*, 89(3), 2003.