Formal Verification of a Practical Lock-Free Queue Algorithm

Simon Doherty[1], Lindsay Groves[1], Victor Luchangco[2], and Mark Moir[2]

[1] School of Mathematical and Computing Sciences,
Victoria University of Wellington, New Zealand
[2] Sun Microsystems Laboratories, 1 Network Drive,
Burlington, MA 01803, USA

Abstract. We describe a semi-automated verification of a slightly optimised version of Michael and Scott's lock-free FIFO queue implementation. We verify the algorithm with a simulation proof consisting of two stages: a forward simulation from an automaton modelling the algorithm to an intermediate automaton, and a backward simulation from the intermediate automaton to an automaton that models the behaviour of a FIFO queue. These automata are encoded in the input language of the PVS proof system, and the properties needed to show that the algorithm implements the specification are proved using PVS's theorem prover.

1 Introduction

Performance and software engineering problems resulting from the use of locks have motivated researchers to develop *lock-free* algorithms to implement concurrent data structures. However, these algorithms are significantly more complicated than lock-based algorithms, and thus require careful proofs to ensure their correctness. Such proofs typically involve long and tedious case analyses, with few interesting cases. Thus, it is desirable to have a tool that generates and checks all the cases, requiring human guidance only in the few interesting cases.

In this paper, we discuss the verification of a lock-free queue algorithm based on the practical and widely used algorithm of Michael and Scott [1]. which to our knowledge has not been formally verified before. We prove that the algorithm is *linearisable* [2], using a *simulation proof*, which involves constructing a special kind of relation, called a *simulation*, between the states of two automata modelling the algorithm and its specification. We use the PVS verification system [3] to check the proof.

Our verification has three principal points of interest: First, unlike many practical algorithms, which can be verified using only a *forward simulation*, this algorithm also requires a *backward simulation*, which is trickier to verify. Second, the way in which we model a dynamic heap, and use an existentially quantified function to relate objects in the heap with abstract data, avoids many difficulties associated with reasoning about dynamic data structures. Third, we developed various techniques to help PVS automatically dispose of most of the cases in the simulation proofs. Using these techniques, we encountered few cases in which we needed to provide guidance to the prover.

We present the queue algorithm in Sect. 2. In Sect. 3, we introduce I/O automata and show how to model the queue specification and implementation. Sect. 4 describes our verification. Sect. 5 discusses our experience using PVS. We conclude in Sect. 6.

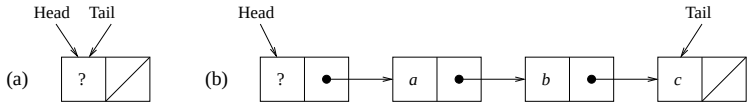

Fig. 1. Basic queue representation

structure pointer_t {*ptr*: **pointer to** node_t, *ver*: **unsigned integer**}
structure node_t {*value*: data type, *next*: pointer_t}
structure queue_t {*Head*: pointer_t, *Tail*: pointer_t}

INITIALISE(Q: **pointer to** queue_t)
node = *new_node*(); node→*next.ptr* = *null*;
Q→*Head* = Q→*Tail* = [node, 0];

Fig. 2. Global declarations and initialisation

2 The Queue Implementation

Our algorithm implements a queue as a linked list of nodes, each having a *value* 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. In quiescent states (i.e., when no operation is in progress), *Tail* points to the last node in the list. Fig. 1 shows an empty queue and a queue containing values *a*, *b* and *c*. The declarations and initialisation are shown in Fig. 2. Pseudocode for the ENQUEUE and DEQUEUE operations is given in Fig. 3.

Shared locations containing pointers (i.e., *Head*, *Tail* and *next*) are updated using *compare-and-swap* (CAS) operations.[3] CAS takes the address of a memory location, an "expected" value, and a "new" value. If the location contains the expected value, the CAS *succeeds*, atomically storing the new value into the location and returning *true*. Otherwise, the CAS *fails*, returning *false* and leaving the memory unchanged.

These shared locations also contain a *version number*, which is incremented atomically every time the location is written.[4] Thus, if such a location contains the same value at two different times, then the location had that value during the entire interval.

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).[5] Between *p* appending its new node and *Tail* being updated, *Tail* *lags* behind the last node in the list (see Fig. 4).

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*, *p*

[3] 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.

[4] In this paper, we treat version numbers as unbounded naturals, so they never "wrap around". This simplification is reasonable as long as enough bits are used for the version number [4].

[5] 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.

```
ENQUEUE(Q: pointer to queue_t,
        value: data type)
E1:  node = new_node()
E2:  node→value= value
E3:  node→next.ptr = null
E4:  loop
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,
                     [node, next.ver+1])
E10:             break
E11:          endif
E12:       else
E13:          CAS(&Q→Tail, tail,
                  [next.ptr, tail.ver+1])
E14:       endif
E15:    endif
E16: endloop
E17: CAS(&Q→Tail, tail,
         [node, tail.ver+1])
```

```
DEQUEUE(Q: pointer to queue_t,
        pvalue: pointer to data type): boolean
D1:  loop
D2:     head = Q→Head
D3:     next = head→next
D4:     if head == Q→Head
D5:        if next.ptr == null
D6:           return false
D7:        else
D8:           *pvalue = next.ptr→value
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]);
                 endif
                 break
D13:          endif
D14:       endif
D15:    endif
D16: endloop
D17: free_node(head.ptr)
D18: return true
```

Fig. 3. Queue operations

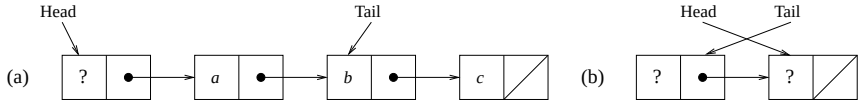

Fig. 4. Queue representation variations

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.

A process *p* executing a DEQUEUE operation checks whether the dummy node 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. Then *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. 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.

Before passing the old dummy node to *free_node*, a dequeuing process checks for the special case shown in Fig. 4(b), where the *Head* and *Tail* have "crossed", because *Tail* 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 [1] 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 [1] 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. Under high load, when operations retry frequently, this change will reduce the number of accesses to shared memory.

3 Modelling the Queue Specification and Implementation

This section briefly introduces the *input/output automaton* (IOA) formalism [5], and shows how we use IOAs to model the queue specification and implementation.

An *input/output automaton* is a labelled transition system, along with a signature partitioning its actions into external and internal actions. Formally, an IOA consists of: a set $states(A)$ of states; a nonempty set $start(A) \subseteq states(A)$ of start states; a set $acts(A)$ of actions; a signature, $sig(A) = (external(A), internal(A))$, which partitions $acts(A)$; and a transition relation, $trans(A) \subseteq states(A) \times acts(A) \times states(A)$.[6]

We describe the states by a collection of state variables, and the transition relation by specifying a *precondition* and *effect* for each action. A precondition is a predicate on states, and an effect is a set of assignments showing only those state variables that change, to be performed as a single atomic action. For states s and s' and action a with precondition pre_a and effect eff_a, the transition (s, a, s') is in $trans(A)$, written $s \xrightarrow{a} s'$, if and only if pre_a holds in s (the *pre-state*) and s' (the *post-state*) is the result of applying eff_a to s. We say that an action a is *enabled* in s if pre_a holds in s. These descriptions are parameterised by process and sometimes by other values, so they actually describe sets of transitions.

A *(finite) execution fragment* of A is a sequence of alternating states and actions of A, $\pi = s_0, a_1, s_1, \ldots s_n$, such that $(s_{k-1}, a_k, s_k) \in trans(A)$ for $k \in [1, n]$. An *execution* is an execution fragment with $s_0 \in start(A)$.[7] A *trace* is the sequence of external actions in some execution. We say that two executions (not necessarily of the same automaton)

[6] The definition in [5] includes additional structure to support fairness and composition, which we do not require for this work.

[7] The full theory of I/O automata also allows infinite executions, which are necessary to reason about liveness, which we do not consider in this paper.

$enq\_inv_p(v)$:	do_enq_p:	enq_resp:
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:** $pc_p := deq\_resp(deq(Q).v)$	**eff:** $pc_p := idle$
	$Q := deq(Q).q$	

Fig. 5. Abstract transitions for process p; v may be any value, and r may be any value or *null*

are *equivalent* if they have the same trace, and we write $traces(A)$ for the set of all traces of A. We also write $trace(\alpha)$ to denote the sequence of external actions in a sequence $\alpha \in acts(A)^*$, where $acts(A)^*$ is the set of finite sequences over $acts(A)$. For $\alpha \in acts(A)^*$, we write $s \xrightarrow{\alpha} s'$ to mean that there is an execution fragment beginning with s, ending with s', and containing exactly the actions of α.

I/O automata can be use to model both specifications and implementations; in both cases, the set of traces represents the possible external behaviours of the automaton. For an "abstract" automaton A, modelling a specification, and a "concrete" automaton C, modelling an implementation, we say that C *implements* A if $traces(C) \subseteq traces(A)$, that is, if all behaviours of the implementation are allowed by the specification.

3.1 The Abstract Automaton

The standard correctness condition for shared data structures is *linearisability* [2], which requires that every operation appears to take effect atomically at some point between its invocation and its response; this point is called the operation's *linearisation point*. We specify the acceptable behaviours for a set of concurrent processes operating on a shared queue, by defining an abstract automaton *AbsAut* which generates their linearizable traces. The transition relation for *AbsAut* is defined in Fig. 5.

AbsAut has external actions $enq\_inv_p(v)$ and deq_inv_p, representing operation invocations, and enq_resp_p, representing the response from an ENQUEUE, for all processes p and values v. For simplicity, we assume that queue values are pointers, and model DEQUEUE as always returning a pointer, which is *null* when the queue is empty. Thus, *AbsAut* has external actions $deq\_resp_p(r)$, where p is any process and r is any value (i.e., non-*null* pointer) or *null*. *AbsAut* also has internal actions do_enq_p and do_deq_p, for all processes p, representing the operations' linearisation points.

Each process p has a "program counter" pc_p that controls the order in which actions can occur by determining which actions are enabled, and sometimes also encodes the value being enqueued or dequeued. For example, when p is not in the midst of any operation, $pc_p = idle$, so $enq\_inv_p(v)$ and deq_inv_p are both enabled; if an $enq\_inv_p(v)$ action occurs, pc_p is set to $enq(v)$, so then only do_enq_p is enabled.

AbsAut has a global variable Q, which holds the abstract queue. The abstract queue is modelled as a function *seq* from naturals to values, along with *Head* and *Tail* counters that delimit the range corresponding to queue elements. The queue consists of $seq(Head + 1)$ through $seq(Tail)$, inclusive; it is empty if $Head = Tail$. The effects of

do_enq_p and do_deq_p actions are defined in terms of functions *enq* and *deq*: $enq(Q, v)$ returns the queue obtained by incrementing $Q.Tail$ and placing v at the new *Tail* index. When Q is not empty, $deq(Q)$ returns a pair $(deq(Q).q, deq(Q).v)$ consisting of the queue obtained by incrementing $Q.Head$ and the element at the new *Head* index. When Q is empty, $deq(Q) = (Q, null)$.

Each process repeatedly performs either an ENQUEUE or DEQUEUE operation, and each such operation consists of an invocation, a single internal action that atomically updates the abstract queue, and a response. Thus, the trace of any execution of *AbsAut* is consistent with a set of processes operating on a linearisable queue.

3.2 The Concrete Automaton

The concrete automaton *ConcAut* models the queue implementation described in Sect. 2. *ConcAut* has the same external actions as *AbsAut*, and has one internal action for each line of code shown in Fig. 3 that contains a read or a write, and two internal actions for each line of code containing a conditional or a CAS. For example, action e_1_p models a process p executing line E1 of ENQUEUE, and $d\_4\_yes_p$ and $d\_4\_no_p$ model p executing D4 when the condition evaluates to *true* and *false*, respectively.

Each process p has a "program counter" pc_p, ranging over a type that contains one value for each line of code containing a read, write, conditional or CAS, and special values *idle*, *enq_resp* and *deq_resp* that play the same roles as in *AbsAut*.

We model a heap in which every object is a node with two fields *value* and *next*, each of which contains a pointer/version-number pair, whose components are denoted by $pair.ptr$ and $pair.ver$. We write $\mathcal{P}$ for the set of pointers, $\mathcal{H}$ for the set of heaps, and $\mathcal{F}$ for the set of field names (either *value* or *next*). A heap $h \in \mathcal{H}$ is a pair $(h.eval, h.unalloc)$: the function $h.eval$: $\mathcal{P} \times \mathcal{F} \to \mathcal{P} \times \mathbb{N}$ 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 are not allocated in h. Generalising this model to allow multiple object types is straightforward, but this simple model suffices for our purposes.

ConcAut has variables $h \in \mathcal{H}$, $Head, Tail \in \mathcal{P} \times \mathbb{N}$, and $freelist \subseteq \mathcal{P}$, which model the heap, *Head*, *Tail* and the freelist. For each process p, there are variables $head_p, tail_p, next_p \in \mathcal{P} \times \mathbb{N}$, and $node_p \in \mathcal{P}$, which model the local variables in the code, and a local variable $result_p \in \mathcal{P}$ to hold the value that p returns from DEQUEUE.

An assignment $pt{\to}fd := (pt', i)$, which updates field fd in the node pointed to by pt, is modelled using a function *update*: $\mathcal{H} \times \mathcal{P} \times \mathcal{F} \times \mathcal{P} \times \mathbb{N} \to \mathcal{H}$ defined by:[8]

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

Allocation of a new node is modelled with the function *new*: $\mathcal{H} \to \mathcal{H} \times \mathcal{P}$ satisfying the following properties:[9]

[8] $f \oplus \{x \mapsto y\}$ yields a function f' such that $f'(x) = y$ and $f'(z) = f(z)$, for $z \neq x$.

[9] Michael and Scott do not specify what happens if ENQUEUE is unable to allocate a new node. In our model, if *new* returns a *null* pointer, *ConcAut* loops until space becomes available. A practical implementation would trap this error.

e_3_p:
pre: $pc_p = \text{e\_3}$
eff: $node_p \rightarrow next.ptr := null$
$pc_p := \text{e\_5}$

d_2_p:
pre: $pc_p = \text{d\_2}$
eff: $head_p := Head$
$pc_p := \text{d\_3}$

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

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

Fig. 6. Part of the transition relation of *ConcAut*

$$new(h) = (h', null) \Rightarrow h.unalloc = \varnothing \wedge h' = h$$
$$new(h) = (h', p) \wedge p \neq null \Rightarrow$$
$$p \in h.unalloc \wedge h'.eval = h.eval \wedge h'.unalloc = h.unalloc \setminus \{p\}$$

The preconditions and effects of some representative actions of the concrete automaton are shown in Fig. 6. Transitions for the other actions are defined similarly.

In subsequent sections, we write $pt \xrightarrow{cs} fd$ for $cs.h.eval(pt, fd)$, and $cs.free?(pt)$ for $pt \in cs.unalloc \cup cs.freelist$, where cs is a state of *ConcAut*.

4 Verification

To verify our queue implementation, we use a simulation proof [6], which shows how to construct, from any execution of the concrete automaton, an equivalent execution of the abstract automaton, proving that *ConcAut* implements *AbsAut*.

Simulation proofs can often be done using a *forward simulation* (see Fig. 7), 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. Therefore, we need to use a *backward simulation* (see Fig. 8), showing how to construct an abstract execution by working from the last step of a concrete execution back to the beginning.

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 Sect. 4.2), and a forward simulation from *ConcAut* to *IntAut* (see Sect. 4.3).

4.1 The Intermediate Automaton

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

$$(\forall\, cs_0 \bullet (\exists\, as_0 \bullet R(cs, as))) \quad (1)$$

$$(\forall\, cs, cs', as, a \bullet \\ R(cs, as) \wedge cs \xrightarrow{a} cs' \Rightarrow \\ (\exists\, as', b \bullet \\ R(cs', as') \wedge as \xrightarrow{b} as' \wedge \\ trace(a) = trace(b))) \quad (2)$$

Fig. 7. A relation $R \subseteq states(C) \times states(A)$ is a *forward simulation* from C to A if C and A have the same external actions and these conditions hold, where $cs_0 : start(C)$, $as_0 : start(A)$ $cs, cs' : states(C)$, $as, as' : states(A)$, $a : acts(C)$, $b : acts(C)$

$$(\forall\, cs \bullet (\exists\, as \bullet R(cs, as))) \quad (3)$$

$$(\forall\, cs_0 : start(C), as \bullet R(cs, as) \Rightarrow \\ as \in start(A)) \quad (4)$$

$$(\forall\, cs, cs', as', a \bullet \\ R(cs', as') \wedge cs \xrightarrow{a} cs' \Rightarrow \\ (\exists\, as, b \bullet \\ (cs, as) \wedge as \xrightarrow{b} as' \wedge \\ trace(a) = trace(b))) \quad (5)$$

Fig. 8. A relation $R \subseteq states(C) \times states(A)$ is a *forward simulation* from C to A if C and A have the same external actions and these conditions hold

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.

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 DEQUEUE operation is linearised to one such point.

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 Sect. 4.3.

It is easy to see that *IntAut* captures the behaviour of a set of processes accessing a linearisable FIFO queue; we describe a formal proof in the following section.

4.2 Backward Simulation Proof

In this section we define a relation *BSR* (see Fig. 9), and show that it is a backward simulation from *IntAut* to *AbsAut*. Given states *as* of *AbsAut* and *is* of *IntAut*, the third

$$BSR(as, is) \stackrel{def}{=} basic\_ok(as, is) \wedge dequeuer\_ok(as, is) \wedge is.Q = as.Q$$

$$basic\_ok(is, as) \stackrel{def}{=}$$
$$\forall p \bullet is.pc_p \neq deq \Rightarrow is.pc_p = as.pc_p$$

$$dequeuer\_ok(as, is) \stackrel{def}{=}$$
$$\forall p \bullet is.pc_p = deq \Rightarrow (as.pc_p = deq \vee (as.pc_p = deq\_resp(null) \wedge is.empty\_ok_p))$$

Fig. 9. The backward simulation relation *BSR*

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 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.

Conditions (3) and (4) of Fig. 8 are trivial, because related states of *AbsAut* and *IntAut* are almost identical. Condition (5) requires that, for every transition $is \xrightarrow{a} is'$ of *IntAut*, if $BSR(is', as')$ holds, then there is some abstract state as and some sequence b of abstract actions, containing exactly the same external actions as a, such that executing each action b, starting from as, takes the abstract automaton into state as'.

To aid in the automation of our proof, we define a function that calculates as given is, is', as' and a. Similarly, we define a *step-correspondence* function [7], that determines the action sequence to choose for the abstract automaton given an action of the intermediate automaton (in our proof, this sequence always consists of either zero or one action). Specifying these functions allows us to avoid manually instantiating the existentially quantified abstract state and abstract action required by the proof obligation: instead we simply use the two functions to calculate them directly.

These functions are defined as follows. For every intermediate action a except *observe_empty*, *deq_empty* and *deq_nonempty*, we choose the same action a for *AbsAut*; for *deq_nonempty*, we choose *do_deq*; and for *deq_empty*, we choose the empty action sequence. 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.

Given the abstract action chosen for a particular intermediate transition, it is generally easy to construct a pre-state as from the post-state 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.

Having chosen an abstract action b, it is usually straightforward to prove $as \xrightarrow{b} as'$, since the construction of as ensures that the precondition for b holds and applying the effect of b to as yields as'. It is slightly trickier in one 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 pc_p in as' 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 $observe\_empty_p$. It is easy to see that $BSR(is, as')$ holds in these cases because the only possible difference between states *is* and is' is that $empty\_ok_p$ is true; the value of this variable affects the truth of $BSR(is, as')$ only if pc_p in as' is $deq\_resp(null)$.

4.3 Forward Simulation Proof

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 Sect. 4.2. We do not have space here to describe the whole simulation relation or the whole proof; instead we present a detailed overview of the most interesting parts.

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

$$FSR(cs, is) \stackrel{def}{=} \exists f : rel(is, cs, f)$$

where f is a function from naturals to pointers called the *representation function*; we explain the purpose of f below. Fig. 10 defines *rel*. Fig. 11 defines *obj_ok*, and Fig. 12 defines some of the other predicates used in defining *rel*.

The most important part of *rel* is the predicate *obj_ok*, 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 a state *is* of *IntAut* contains a

$$rel(is, cs, f) \stackrel{def}{=} enqueue\_ok(is, cs, f) \wedge dequeue\_ok(is, cs, f) \wedge obj\_ok(is, cs, f) \wedge nds\_ok(is, cs, f) \wedge distinctness\_ok(is, cs, f) \wedge procs\_ok(is, cs, f) \wedge injective\_ok(is, cs, f) \wedge access\_safety\_ok(is, cs, f)$$

Fig. 10. The *rel* predicate

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

Fig. 11. The *obj_ok* predicate

queue variable Q, represented by a sequence and *Head* and *Tail* variables indicating which indexes are relevant in the current queue state. If $obj\_ok(is, cs, f)$ holds, then f indicates which node corresponds to each relevant position in $is.Q.seq$; i.e., for each $i \in [is.Q.Head + 1 \cdots is.Q.Tail]$, $f(i)$ is the queue node in cs containing the value $is.Q.seq[i]$, and $f(is.Q.Head)$ indicates which queue node in cs is the dummy node pointed to by $cs.Head.ptr$. The latter is stated by Conjunct (1) of *obj_ok*. 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 *Tail.ptr* points to the next-to-last node in the queue, and several other properties that help the proof to go through hold (3b). Conjunct (4) expresses the properties of the nodes in the concrete queue: the pointer value of the *next* field of each node points to the node corresponding to the next index (4a); the value in each relevant node is the value in the corresponding position in $is.Q.seq$ (4b); none of the relevant nodes is unallocated or in the freelist (4c); and none of the relevant nodes is *null* (4d).

Predicates *enqueue_ok* and *dequeue_ok* (Fig. 12) play the same role as *basic_ok* and *dequeuer_ok* in the backward simulation. The other predicates capture properties needed to support the proof of the other properties. $nds\_ok(is, cs, f)$ expresses properties of a node as it gets initialised (Fig. 12). The *distinctness_ok* predicate expresses that various values are distinct, for example, that nodes being initialised by two different processes are different. The *procs_ok* predicate expresses several properties about the private variables of processes. Some of its subpredicates are shown in Fig. 12. For example, *procs_ok_15* says that if a process p is executing ENQUEUE and pc_p is e_9, then the pointer component of $next_p$ is *null*. The *injective_ok* predicate ensures that each node corresponds to only one index (in the relevant range), so that modifications to a node corresponding to one index do not destroy properties required of nodes corresponding to other indexes. The *access_safety_ok* predicate says that the implementation

$$
\begin{array}{l}
\mathit{enqueue\_ok}(is, cs, f) \stackrel{def}{=} \\
\quad \forall p \bullet (cs.pc_p = \mathit{idle} \Rightarrow is.pc_p = \mathit{idle}) \wedge \\
\qquad (\mathit{pc\_e\_1\_9}(cs, p) \vee cs.pc_p = \text{e\_13} \Rightarrow is.pc_p = \mathit{enqueuing}(cs.\mathit{value}_p)) \wedge \\
\qquad (cs.pc_p = \text{e\_17} \vee cs.pc_p = \mathit{enq\_resp} \Rightarrow is.pc_p = \mathit{enq\_resp})
\end{array}
$$

$$
\begin{array}{l}
\mathit{nds\_ok}(is, cs, f) \stackrel{def}{=} \forall p \bullet (\mathit{pc\_e\_2\_13}(cs, p) \Rightarrow \neg cs.\mathit{free?}(cs.\mathit{node}_p) \wedge cs.\mathit{node}_p \neq \mathit{null}) \wedge \\
\qquad\qquad\qquad\qquad (\mathit{pc\_e\_3\_13}(cs, p) \Rightarrow cs.\mathit{node}_p \xrightarrow{cs} \mathit{value.ptr} = cs.\mathit{value}_p) \wedge \\
\qquad\qquad\qquad\qquad (\mathit{pc\_e\_4\_13}(cs, p) \Rightarrow cs.\mathit{node}_p \xrightarrow{cs} \mathit{next.ptr} = \mathit{null})
\end{array}
$$

$$
\begin{array}{l}
\mathit{procs\_ok\_5}(is, cs, f) \stackrel{def}{=} \\
\quad \forall p \bullet \mathit{pc\_e\_8\_9}(cs, p) \wedge cs.\mathit{next}_p.\mathit{ptr} = \mathit{null} \Rightarrow \\
\qquad\qquad cs.\mathit{next}_p.\mathit{ver} < cs.\mathit{tail}_p.\mathit{ptr} \xrightarrow{cs} \mathit{next.ver} \vee (cs.\mathit{next}_p = cs.\mathit{tail}_p.\mathit{ptr} \xrightarrow{cs} \mathit{next} \wedge \\
\qquad\qquad cs.\mathit{tail}_p = cs.\mathit{Tail} \wedge cs.\mathit{tail}_p.\mathit{ptr} = f(is.Q.\mathit{Tail}))
\end{array}
$$

$$
\mathit{procs\_ok\_15}(is, cs, f) \stackrel{def}{=} \forall p \bullet cs.pc_p = \text{e\_9} \Rightarrow cs.\mathit{next}_p.\mathit{ptr} = \mathit{null}
$$

$$
\mathit{procs\_ok\_16}(is, cs, f) \stackrel{def}{=} \forall p \bullet \mathit{pc\_e\_6\_13}(cs, p) \Rightarrow cs.\mathit{node}_p.\mathit{ptr} \neq cs.\mathit{tail}_p.\mathit{ptr}
$$

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

Fig. 12. Some predicates used in *FSR*. A predicate of the form $\mathit{pc\_e\_m\_n}(cs, p)$, where m, n are integers, holds when $cs.pc_p = \text{e\_}i$ for some $i \in [m, n]$.

never dereferences *null* or accesses a node that is in *unalloc*, which is important for correct interaction with a memory allocator.

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: $\mathit{e\_9\_yes}_p$, which models a successful CAS at line E9, is mapped to $\mathit{do\_enq}_p$; $\mathit{d\_9\_yes}_p$ is mapped to $\mathit{deq\_nonempty}_p$; $\mathit{d\_3}_p$ is mapped to $\mathit{observe\_empty}_p$; and $\mathit{d\_5\_yes}_p$ is mapped to $\mathit{deq\_empty}_p$.

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. However, we specify a *witness function* that shows how to choose the new f so that *FSR* holds between the concrete and intermediate post-states. For a representation function f, concrete action a, concrete state cs and intermediate state is, the witness function returns the function $f' = f \oplus \{is.Q.\mathit{Tail}+1 \mapsto cs.\mathit{node}_p\}$.

We now present a careful manual proof that *obj_ok* is preserved across transitions that represent the execution of line E9 by some process, 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.

Consider a concrete transition $cs \xrightarrow{a} cs'$, where $a = \mathit{e\_9\_yes}_p$ for some p, intermediate state is and representation function f, and let as' and f' be respectively the interme-

diate state and function determined by the step-correspondence and witness functions. When we say that part of the simulation relation *holds in the pre-state* (resp. *holds in the post-state*), we mean that it is true for cs, is and f (resp. cs', is', f').

The step-correspondence associates $e\_9\_yes_p$ with $do\_enq_p(cs.value_p)$, so we need to show that if the precondition of $e\_9\_yes_p$ holds in the pre-state (see Fig. 6) and $rel(is, cs, f)$ then $obj\_ok(is', cs', f')$.

First, we make some observations about the transition:

$$cs.Tail.ptr = cs.tail_p.ptr = f(is.Q.Tail) \quad \text{(i)}$$
$$f'(is'.Q.Tail) = cs.node_p \quad \text{(ii)}$$

Claim (i) is shown using $procs\_ok\_15$ to yield that $cs.next_p.ptr = null$, and then using $procs\_ok\_5$ to yield that $cs.Tail.ptr = cs.tail_p.ptr = f(is.Q.Tail)$. Claim (ii) follows immediately from the construction of f' and the effect of do_enq_p.

(1) of obj_ok is preserved because $is'.Q.Head = is.Q.Head$, but $is.Q.Head < is.Q.Tail+1$ (this is a simple invariant of *IntAut*). Therefore $is'.Q.Head \neq is.Q.Tail+1$, so by construction of f' and because obj_ok holds in the pre-state, $f'(as'.Q.Head) = f(is.Q.Head) = cs.Head.ptr = cs'.Head.ptr$.

For (2), by construction of f' and the effect of do_enq_p, $f'(is'.Q.Tail) = f'(is.Q.Tail+1) = cs.node_p$. Moreover, by 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 (3b) holds in the post-state, arguing each sub-conjunct in turn.

$$\begin{aligned}
f'(is'.Q.Tail) &= cs.node_p && \text{by (ii) above}\\
&= cs.tail_p.ptr \xrightarrow{cs'} next.ptr && \text{by construction of } cs'\\
&= cs.Tail.ptr \xrightarrow{cs'} next.ptr && \text{by (i) above}\\
&= cs'.Tail.ptr \xrightarrow{cs'} next.ptr && \text{because } cs'.Tail = cs.Tail
\end{aligned}$$

$$\begin{aligned}
cs'.free?(cs'.Tail.ptr) &= cs.free?(cs'.Tail.ptr) && \text{because } cs'.free? = cs.free?\\
&= cs.free?(cs.Tail.ptr) && \text{because } cs'.Tail = cs.Tail\\
&= cs.free?(f(is.Q.Tail)) && \text{by (i) above}\\
&= false && \text{conjunct 4c with } i = is.Q.Tail
\end{aligned}$$

Now by claim (i), $cs.Tail.ptr = f(is.Q.Tail)$, 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

$$\begin{aligned}
f'(is'.Q.Tail) &= cs.node_p && \text{by (ii) above}\\
&\neq cs.tail_p.ptr && \text{by } procs_ok_16\\
&= cs.Tail.ptr && \text{by (i) above}\\
&= cs'.Tail.ptr
\end{aligned}$$

We prove (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 (4a). For (4b) we have

$$
\begin{aligned}
is'.Q.seq(i) &= cs.value_p && \text{by effect of } do\_enq_p \text{ and } enqueue\_ok \\
&= cs.node_p \xrightarrow{cs} value.ptr && \text{by } nds\_ok \\
&= cs.node_p \xrightarrow{cs'} value.ptr && \text{by effect of } e\_9\_yes w_p \\
&= f'(i) \xrightarrow{cs'} value.ptr && \text{by (ii) above}
\end{aligned}
$$

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

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

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

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

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 *value* fields, nor *free?* are modified by the transition.

5 Experience with PVS

In this section we describe our experience using PVS to prove that the relations presented in the previous sections are in fact simulations. We focus on the forward simulation from *ConcAut* to *IntAut* because of its greater complexity. The techniques used to verify the backward simulation are similar.

The PVS system [3] provides a specification language, which we used to define the notions of backward and forward simulation. Using techniques adapted from [8], we also encoded the three automata, *AbsAut*, *IntAut* and *ConcAut*, as well as the simulation relations, *BSR* and *FSR*.

One of the goals of our verification effort was to construct the proof without requiring the human prover to attend to the tedious and uninformative aspects. We achieved this using two techniques: using the step-correspondence and witness functions, and dividing the forward simulation proof into many small, manageable parts. As noted in

Sect. 4.2, using predefined functions to instantiate existentially quantified variables relieves the user of needing to manually instantiate these variables during proofs. Also, as described below, dividing the proof into many small parts allowed us to quickly isolate the parts of the proof that required human insight.

We divided the forward simulation verification condition into over 1000 lemmas. One lemma covers condition 1 of Fig. 7; for each concrete action associated by the step-correspondence with a nonempty intermediate action sequence, there is a lemma stating that if the concrete precondition holds, then the intermediate precondition holds in all related states; and finally, more than 900 *preservation lemmas*, each asserting that a part of the simulation relation is preserved across some transition. We used the mechanical proof facilities of PVS to prove a large proportion of these lemmas automatically.

Constructing proofs for the preservation lemmas constituted by far the bulk of the proof effort, and so we describe the techniques used to achieve this here. The conjuncts of the simulation relation can be divided into a small number of classes, depending on the presence and structure of the top level quantification: for example, *enqueue_ok* and all the subpredicates of *procs_ok* are universally quantified over a single process, so fall into the same class. For each of these classes, we developed a simple strategy that set up a proof, to be continued by a user or automated strategy. All these strategies begin by executing a strategy called `Begin-Simstep`, which evaluates the step-correspondence and witness functions, and expands the definition of *rel* and the definitions on which it depends, resulting in a set of subformulae each making assertions about *is*, *cs* and f. `Begin-SimStep` then labels each subformulae, allowing strategies applied later to refer to each subformula by name. Because *rel* is too complex to be analysed by PVS's automated strategies, `Begin-SimStep` *hides* the subformulae of *rel*. In PVS, each subgoal of a proof is associated with a set of formulae that are *hidden*; that is, they are not visible to any strategies, unless they are first *revealed*.

After `Begin-SimStep` has completed, one or more strategies are applied, each of which applies proof steps that are always needed to prove a conjunct of a particular form. For example, the `SimStep-obj-ok` strategy, which is applied at the beginning of preservation proofs involving *obj_ok* (which has no top-level quantifier), expands *obj_ok* in the consequent, and generates a set of new subgoals, where each conjunct must be shown to hold in the post-state. Once this strategy is completed, either an automatic strategy is applied to attempt to complete the proof without user intervention, if possible, or PVS waits for a command to be invoked interactively.

Now we have a situation in which the user is presented with a set of subgoals. Using primitive PVS proof commands and the labels defined by `Begin-SimStep`, the user reveals antecedent formulae that assert facts about the pre-state that are relevant to the subgoal at hand and instantiates any universally quantified variables. Once the relevant formulae have been revealed and instantiated, it remains to invoke the PVS automated strategies on the subgoal. These strategies apply boolean decision procedures, rewrite rules, and sometimes heuristic instantiations to attempt to complete the goal.

The limited form of interaction with the theorem prover not only reduces user-effort, but also improves the robustness of the proof. As the project progressed, we often made small modifications to the simulation relation and even the automata. Because we used proof commands that did not depend on fine aspects of the formulae being proved, we

were able to successfully re-run most proofs after a modification, without changing the proofs themselves.

6 Concluding Remarks

We have presented a variation on the practical lock-free FIFO queue algorithm of Michael and Scott, and described a semi-automated proof of its linearisability we developed using the PVS system. The algorithm and specification are both modelled using I/O automata, and the proof is based on a combination of forward and backward simulation proofs. Our work illustrates some techniques for modelling and reasoning about dynamically allocated memory, and also some techniques for fully automating the easy parts of proofs, allowing the human prover to focus on aspects of the proof that require human insight. Future work includes refining our techniques to increase automation and applicability, as well as applying them to other problems. We expect that our efforts to automate the easy parts of the proof will enable us to tackle larger and more complicated problems in the future.

References

1. Michael, M., Scott, M.: Nonblocking algorithms and preemption safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing **51** (1998) 1–26
2. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS **12** (1990) 463–492
3. Crow, J., Owre, S., Rushby, J., Shankar, N., Srivas, M.: A tutorial introduction to PVS. In: Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida (1995)
4. Moir, M.: Practical implementations of non-blocking synchronization primitives. In: Proceedings of the 15th Annual ACM Symposium on the Principles of Distributed Computing, Santa Barbara, CA. (1997)
5. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
6. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations – Part I: Untimed systems. Information and Computation **121** (1995) 214–233
7. Ramírez-Robredo, J.A.: Paired simulation of I/O automata. Master's thesis, Massachusetts Institute of Technology (2000)
8. Devillers, M.: Translating IOA automata to PVS. Technical Report CSI-R9903, Computing Science Institute, University of Nijmegen, the Netherlands (1999)