Formal Verification of a Lazy Concurrent List-Based Set Algorithm

Robert Colvin[1], Lindsay Groves[2], Victor Luchangco[3], and Mark Moir[3]

[1] ARC Centre for Complex Systems, School of Information Technology and Electrical Engineering, University of Queensland, Australia
[2] School of Mathematics, Statistics and Computer Science, Victoria University of Wellington, New Zealand
[3] Sun Microsystems Laboratories, 1 Network Drive, Burlington, MA 01803, USA

Abstract. We describe a formal verification of a recent concurrent list-based set algorithm due to Heller *et al*. The algorithm is optimistic: the *add* and *remove* operations traverse the list without locking, and lock only the nodes affected by the operation; the *contains* operation uses no locks and is wait-free. These properties make the algorithm challenging to prove correct, much more so than simple coarse-grained locking algorithms. We have proved that the algorithm is linearisable using simulation between input/output automata modelling the behaviour of an abstract set and the implementation. The automata and simulation proof obligations are specified and verified using PVS.

1 Introduction

Concurrent algorithms are notoriously difficult to design correctly, and high performance algorithms that make little or no use of locks even more so. Formal verification of such algorithms is challenging because their correctness often relies on subtle interactions between processes that heavier use of locks would preclude. These proofs are too long and complicated to do (and check) reliably "by hand", so it is important to develop techniques for mechanically performing, or at least checking, such proofs.

In this paper we describe a formal verification of *LazyList*, a recent concurrent list-based set algorithm due to Heller *et al.* [1]. Our proof shows that the algorithm is linearisable to an abstract set object supporting *add*, *remove*, and *contains* methods. Linearisability [2] is the standard correctness condition for concurrent shared data structures. Roughly, it requires that each operation can be assigned a unique *linearisation point* during its execution at which the operation appears to take effect atomically.

The LazyList algorithm is optimistic: *add* and *remove* operations attempt to locate the relevant part of the list without using locks, and only use locks to validate the information read and perform the appropriate insertion or deletion. The *contains* operation uses no locks, and is simple, fast, and wait-free. Heller *et al.* present performance studies showing that this algorithm outperforms well known algorithms in the literature, especially on common workloads in which the *contains* method is invoked significantly more often than *add* and *remove* [1].

The simplicity and efficiency of the *contains* method is achieved by avoiding all checks for interactions with concurrent *add* and *remove* operations. As a result, a *contains* operation can decide that the value it is seeking is not in the set at a moment when in fact the value is in the set. The main challenge in proving that the algorithm is linearisable is to show that this happens only if the sought-after value was absent from the set *at some point* during the execution of the *contains* operation.

We have proved that the algorithm is linearisable using simulation between input/output automata modelling the abstract behaviour of the set and the implementation. Our proof uses a combination of forward and backward simulations, and has the interesting property that a single step of the implementation automaton can correspond to steps by an arbitrary number of different processes in the specification automaton. We modelled the automata and encoded the proof obligations for simulations in the PVS specification language [3], and used the PVS system to check our proofs.

Apart from presenting the first complete and formal verification of an important new algorithm, a contribution of this paper is to describe our ongoing work towards making proof efforts like these easier and more efficient. The proof presented in this paper builds on earlier work in which we proved (and in some cases disproved and/or improved) a number of nonblocking implementations of concurrent stacks, queues and deques [4,5,6,7]. While we still have work to do in this direction, we have made a lot of progress in understanding how to model algorithms and specifications, and how to approach proofs. In this paper, we briefly describe some of the lessons learned. We have made our proof scripts available at `http://www.mcs.vuw.ac.nz/research/SunVUW/`, so that others may examine our work in detail and benefit from our experience.

The rest of the paper is organised as follows. We describe the LazyList algorithm in Section 2, and our verification of it in Section 3. We discuss our experience with using PVS for this project in Section 4, and conclude in Section 5.

2 The LazyList Algorithm

The LazyList algorithm implements a concurrent set supporting three operations:

- *add*(k) adds k to the set and "succeeds" if k is not already in the set.
- *remove*(k) removes k from the set and "succeeds" if k is in the set.
- *contains*(k) "succeeds" if k is in the set.

Each operation returns *true* if it succeeds; otherwise it "fails" and returns *false*.

The algorithm uses a linked-list representation. In addition to *key* and *next* fields, each list node has a *lock* field, used to synchronise *add* and *remove* operations, and a *marked* field, used to logically delete the node's key value (see Figure 1). The list is maintained in ascending key order, and there are two sentinel nodes, *Head* and *Tail*, with keys $-\infty$ and $+\infty$ respectively. We assume that the list methods are invoked only with integer keys k (so that $-\infty < k < +\infty$).

As explained in more detail below, a successful *add*(k) operation inserts a new node containing k into the list, and a successful *remove*(k) operation logically removes k from the set by marking the node containing k (i.e., setting its *marked* field to *true*), before cleaning up by removing the node from the list. Thus, at any point in time, the abstract set is exactly the set of values stored in the *key* fields of the unmarked nodes in the list.

```
private class Entry {
  int key;
  Entry next;
  boolean marked;
  lock lock;
}
```

Fig. 1. Declaration of list node type

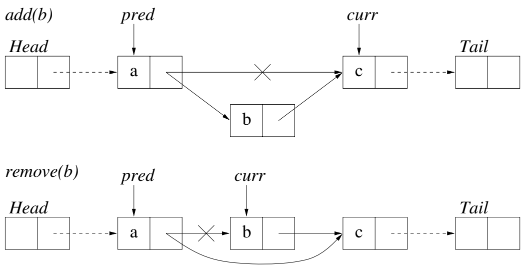

Fig. 2. Inserting and removing list nodes

The *add*(k) and *remove*(k) methods (see Figure 3) use a helper method *locate*(k), which sets *curr* to point to the first node with a key greater than or equal to k and *pred* to point to that node's predecessor in the list—the values in *Head* and *Tail* ensure that these both exist. The *locate* method optimistically searches the list without using locks, and then locks the nodes pointed to by *curr* and *pred*. If both nodes are unmarked (i.e., their *marked* fields are *false*) and *pred.next* is equal to *curr*, the *add* or *remove* operation can proceed; otherwise, the locks are released and the search is restarted.

An *add*(k) operation calls *locate*, then compares *curr.key* to k. If *curr.key* is not equal to k, then k is not in the list, so *add* creates a new node, setting its *key* field to k and its *next* field to point to *curr* (see Figure 2). It then sets the *next* field of *pred* to point to this new node, releases the locks on *curr* and *pred*, and succeeds. If *curr.key* is equal to k, then k is already in the list, so the *add* operation fails.

A *remove*(k) operation also calls *locate*, then compares *curr.key* to k. In this case, if *curr.key* equals k, then *remove* removes the node at *curr* from the list and succeeds; otherwise, k is not in the list, so *remove* fails. A successful removal is done in two stages: first the key is logically removed from the set by setting the *marked* field of *curr*; then it is physically removed by setting the *next* field of its predecessor (*pred*) to its successor (*curr.next*) (see Figure 2). Separating the logical removal of the key from the set and the physical removal of the node from the list is crucial to the simplicity and efficiency of the algorithm: because nodes are not removed before they are marked, observing an unmarked node is sufficient to infer that its key is in the set.

A *contains*(k) operation makes a single pass through the list, starting from *Head*, searching for a node with a key not less than k. If this node contains k and is not marked, the *contains* operation succeeds; otherwise it fails. This operation requires no locks and is wait-free (i.e., it is guaranteed to complete within a finite number of its own steps, even if processes executing other operations are delayed or stop completely).

Linearisation Points. A common way to prove that an algorithm is linearisable is to identify a particular step of each operation as the linearisation point of that operation. With some simple invariants showing there are no duplicate keys in the list, it is straightforward to assign linearisation points in this way for *add* and *remove* operations, and for successful *contains* operations. We can linearise a successful *add* operation when it inserts its node into the list (line 6), a successful *remove* operation at the point at which

```
contains(k) :
 1  curr := Head;
 2  while curr.key < k do
 3    curr := curr.next;
 4  if curr.key = k and
 5       ~curr.marked then
      return true
    else
      return false

locate(k) :
    while true do
 1    pred := Head;
 2    curr := pred.next;
 3    while curr.key < k do
 4      pred := curr;
 5      curr := curr.next;
 6    pred.lock();
 7    curr.lock();
 8    if ~pred.marked and
 9         ~curr.marked and
10         pred.next = curr then
11      return pred, curr
      else
12      pred.unlock();
13      curr.unlock()
```

```
add(k) :
 1  pred, curr := locate(k);
 2  if curr.key != k then
 3    entry := new Entry();
 4    entry.key := k;
 5    entry.next := curr;
 6    pred.next := entry;
 7    res := true
    else
 8    res := false;
 9  pred.unlock();
10  curr.unlock();
    return res

remove(k) :
 1  pred, curr := locate(k);
 2  if curr.key = k then
 3    curr.marked := true;
 4    entry := curr.next;
 5    pred.next := entry;
 6    res := true
    else
 7    res := false;
 8  pred.unlock();
 9  curr.unlock();
    return res
```

Fig. 3. Pseudocode for LazyList algorithm

it marks the node (line 3), and a successful *contains* operation at the point at which it reads the *marked* field of a node containing its key (line 5). An unsuccessful *add* or *remove* can be linearised anywhere between acquiring the lock on *curr* and releasing the lock on *pred*.

Things are not so simple for failed a *contains* operation, however. If the node found by the loop at lines 2 and 3 contains a key greater than k, or it is marked, *contains*(k) returns *false*. But there is no step of the *contains* operation at which k is guaranteed not to be in the set. In particular, when its *key* or *marked* field is checked, the node may have already been removed from the list, and another process may have added a new node with key k, so that k is in the abstract set at that time. Thus the simple approach of proving linearisability by defining a linearisation point for each operation at one of its steps does not work for this algorithm.

The key to proving that LazyList is linearisable is to show that, for any failed *contains*(k) operation, k is absent from the set at *some* point during its execution. Our proof shows that if a *contains*(k) operation fails, then either k is absent from the set when the operation begins or some successful *remove*(k) operation marks a node containing k during the execution of the *contains*(k) operation. Because there may be many *contains*(k) operations executing concurrently, it is sometimes necessary to linearise multiple failed *contains* operations after the same *remove*(k) operation. We found this interesting, because our previous proofs have not required this.

3 Verification

To prove that LazyList is a linearisable implementation of a set supporting *add*, *remove*, and *contains* operations, we define two input/output automata (IOA) [8,9]: a *concrete automaton ConcAut*, which models the behaviour of the LazyList algorithm, and a simple *abstract automaton AbsAut*, which specifies all correct behaviours of a linearisable set. We use *simulation* proof techniques [10] to prove that *ConcAut* implements *AbsAut*.

3.1 I/O Automata and Simulation Proofs

We now informally describe the IOA model and simulation proofs. In our verification, we use a simplified version of IOAs, which is sufficient for this verification. See [8,9,10] for a more detailed and formal discussion.

An IOA consists of a set of states and a set of actions. Each action has a *precondition*, which determines the set of states from which it can be executed, and an *effect*, which determines the next state after the action has been executed. The actions are partitioned into *external* actions, which define the interface of the automaton, and *internal* actions, which represent internal details. An automaton C *implements* an automaton A if for every execution of C, there exists an execution of A with the same external actions, implying that C and A are indistinguishable to an external observer.

One way to prove that C implements A is by *forward simulation*. Given an arbitrary execution of C, we inductively construct an equivalent execution for A by working forwards from the beginning of C's execution, choosing a (possibly empty) sequence of actions of A for each step in C's execution. That is, we start from some initial state of A, choose a sequence of actions of A for the first step of C, execute those actions, resulting in a new state of A, and then choose and execute a sequence of actions corresponding to the next step of C, and so on. In a forward simulation proof, we must choose the action(s) of A for a given step of C based on the step and the state of A thus far in the simulation; we cannot use steps of C later in the execution to make this choice.

The goal is for the constructed execution of A to have the same external behaviour as the execution of C. To guarantee that it does, we insist that the sequence of actions chosen for an internal action of C contains no external actions, and the sequence of actions chosen for an external action of C contains that external action and no other external action. We must also ensure that the sequence of actions we choose for A can be executed from the state of A resulting from the sequence of actions corresponding to the previous step of C.

To ensure that we can successfully carry out this process over the entire execution of C, we must choose actions for A in a way that keeps A "in step" with C. We capture our intuition about what "in step" means by defining a relation, called a *forward simulation*, between states of C and states of A, and we choose actions for A so as to "preserve" the simulation relation. Often the biggest challenge lies in precisely capturing our intuition about why C implements A so that we can define a simulation relation that makes this possible.

For some algorithms and their specifications, however, there is no way to define such a forward simulation because for some action of C, the actions of A that we should choose depend on future actions (i.e., actions that appear later in the execution). As we explain later, LazyList is one such algorithm. In such circumstances, we use a *backward simulation*. Instead of starting from the initial state and working forwards along an execution of C, in a backward simulation we start at the *last* state of an arbitrary execution, and work backwards towards the initial state. (Because we are only concerned with safety properties in this work, it suffices to consider only finite executions.)

Apart from the direction of the induction, there are some additional differences between forward and backward simulations. First, in a forward simulation, the sequence of actions we choose for A (together with the current state of A) uniquely determines the poststate for A (at least for automata with deterministic actions like the ones we use). But in a backward simulation, for a given action of C, we are given a *poststate* of A, and we must choose not only a sequence of actions for A, but also a *prestate* such that executing the chosen action(s) from this prestate brings us to the given poststate *and* the prestate of C's action is related by the simulation relation to the chosen prestate for A.

Second, we must ensure that when we reach the beginning of C's execution, A is in an initial state too. Therefore, a backward simulation relation must ensure that every state of A that is related to an initial state of C is an initial state of A. Forward simulation has no such proof obligation.

When a backward simulation is necessary, it is often convenient to develop the proof in two stages by defining an "intermediate" automaton, and proving (i) that the concrete automaton implements the intermediate automaton using a forward simulation and (ii) that the intermediate automaton implements the abstract one using a backward simulation. We took this approach for this verification, as we had used it successfully in previous verifications, e.g. [5].

3.2 The Abstract and Concrete Automata

We now describe informally the abstract and concrete IOAs that we use in this verification; more detailed descriptions of the way we use IOAs to model specifications and implementations can be found in [4,5,6,7].

The abstract automaton *AbsAut* models a set of processes operating on an abstract set, in which each process is either "idle", in which case it can invoke any operation on the set, or is in the midst of executing an operation.

Each operation is modelled in *AbsAut* using four actions: two external actions modelling the operation's invocation and response, and two internal actions modelling successful and unsuccessful application of the operation. For example, for the *add* operation (see Figure 4), the *addInv*(k, p) action models the invocation of the *add*(k) operation by process p, and the *addResp*(r, p) action models the operation returning boolean value r to process p. The precondition of the *doAddT* action requires that k is not in the abstract set, and its effect adds k to the set; the precondition of the *doAddF* action requires that k is in the set, and its effect does not modify the set.

Per-process *program counter* variables constrain the order in which actions are performed, ensuring that each operation consists of an invocation action, a *do* action, and a

Action	Precondition	Effect
addInv(k, p)	pc(p) = idle	pc(p) := pcDoAdd(k)
doAddT(k, p)	pc(p) = pcDoAdd(k) AND NOT member(k, keys)	pc(p) := pcAddResp(true) keys := add(k, keys)
doAddF(k, p)	pc(p) = pcDoAdd(k) AND member(k, keys)	pc(p) := pcAddResp(false)
addResp(r, p)	pc(p) = pcAddResp(r)	pc(p) := idle

Fig. 4. *AbsAut* actions for the *add* operation

response action. These variables also connect the return value of the response action to the *do* action. For example, the *doAddT*(p) action sets process p's program counter to *pcAddResp*(*true*). Thus each operation is guaranteed to return a value consistent with applying the operation atomically at the point at which the *do* action is executed. Because each operation "takes effect" atomically at the execution of its internal *do* action, all executions of *AbsAut* are behaviours of a linearisable set. Thus, proving that *ConcAut* implements *AbsAut* proves that LazyList is a linearisable set implementation.

The concrete automaton *ConcAut* models a set of processes operating on a set implemented by the LazyList algorithm. It has the same external actions as *AbsAut* (i.e invocations and responses for each operation), and has an internal action for each step of the algorithm corresponding to a labelled step in the pseudocode shown in Figure 3, which are assumed to be atomic. In fact, conditional steps in the algorithm have two associated actions, one for each outcome of the step. For example, the precondition of the *cont2T*(p) action (which models an execution by process p of line 2 of the *contains* method when the test succeeds) requires that p's program counter is *pcCont2* and $curr.key_p < k_p$, and its effect sets p's program counter to *pcCont3*. The compound tests in *contains* (lines 4 and 5) and *locate* (lines 8 to 10) are each treated as a sequence of (two and three, respectively) atomic tests. We also assume that allocation of a new node (*add* line 3) is atomic.

3.3 An Intermediate Automaton

As mentioned earlier, we cannot prove that *ConcAut* implements *AbsAut* using a forward simulation proof. The reason is that, to do so, we must identify a point at which each operation "takes effect" and choose the corresponding *do* action in *AbsAut* at that point. However, as explained in Section 2, a failed *contains* operation may not take effect at the point that it determines it has failed: the point at which the sought-after key is absent from the set may be earlier. A correct forward simulation proof would have to choose the *doContF* action at a point that the key is absent from the set. However, at that point, it is still possible that the *contains* operation will return *true*, so choosing *doContF* would make it impossible to complete the proof because when *ConcAut* executes the external *contResp*(*true*) action, the precondition for this action would not hold in the *AbsAut* state (recall that we are required to choose the same action for *AbsAut* when the *ConcAut* action is external).

Thus, we introduce an intermediate automaton *IntAut* that eliminates the need to "know the future" when choosing appropriate actions to prove that *ConcAut* implements

IntAut using a forward simulation, and we show that *IntAut* implements *AbsAut* using a backward simulation. Because backward simulations are generally more difficult than forward ones, we prefer to keep the intermediate automaton as close as possible to the abstract one. We achieved this by modifying *AbsAut* slightly so that the *contains* operation can decide to return *false* if the key it is seeking was absent from the set at some time since its invocation (though it may still return *true* if it finds its key in the set). We now explain how we achieved this.

The state of *IntAut* is the same as that of *AbsAut*, except that we augment each process p with a boolean flag $seen\_out_p$. When process p is executing a *contains*(k) operation, $seen\_out_p$ indicates whether k has been absent from the set at any time since the invocation of the operation.

The transitions of *IntAut* are the same as those of *AbsAut*, except that:

– The *contInv*(p,k) action sets $seen\_out_p$ to *false* if k is in the abstract set, and to *true* otherwise.
– The *doRemT*(q,k) action, in addition to removing k from the set, also sets $seen\_out_p$ for every process p that is executing *contains*(k).
– The precondition of the *doContF*(k,p) action requires $seen\_out_p$ to be true instead of k being absent from the set.

Thus the *doContF*(p) action is enabled if k was not in the set when *contains*(k) was invoked, or if it was removed later by some other process q performing *doRemT*(q,k). Therefore, in this automaton, a *contains*(k) operation can decide to return *false* even when k is in the set, provided k was absent from the set sometime during the operation. This is what we need in order to prove a forward simulation from *ConcAut* to *IntAut*.

3.4 The Backward Simulation

Because *IntAut* is so close to *AbsAut*, the backward simulation is relatively straightforward: it requires that the sets of keys in *IntAut* and *AbsAut* are identical, and that each process p in *AbsAut* stays "in step" with process p in *IntAut*, with one exception. In *AbsAut*, p may have already executed *doContF*, indicating that it will subsequently return *false*, whereas in *IntAut*, p has not yet decided to return *false*. This is allowed only if $seen\_out_p$ is *true*, indicating that either k was absent from the abstract set at the invocation *contInv*(k,p), or was present at the invocation but was subsequently removed before *doContF* is performed. The PVS definition of our backward simulation between *IntAut* state i and *AbsAut* state a is shown below.

```
bsr(i, a): bool = i`keys = a`keys AND
                  FORALL p: (i`pc(p) = a`pc(p) OR
                             (i`pc(p) = pcDoCont AND
                              a`pc(p) = pcContResp(false) AND
                              i`seen_out(p)))
```

Below we briefly describe how we choose an action sequence for *AbsAut* for a given step of *IntAut* in our backward simulation proof. Readers interested in details of how we choose a prestate for *AbsAut*, and how we discharge the various proof obligations for a backward simulation (including proving that our choices result in a valid execution of *AbsAut*) can examine our files and step through the proofs.

In the backward simulation, for most actions in *IntAut*, we choose the same action for *AbsAut*. However, as discussed above, we cannot simply choose *doContF* in *AbsAut* when a *doContF* action occurs in *IntAut* because the sought-after key may be in the set at that point. Instead, we choose *doContF* actions as follows:

- For a *contInv*(k, p) action in *IntAut*, if p is enabled to return *false* in the poststate of *AbsAut*, we choose *contInv*(k, p) followed by *doContF*(k, p).
- For a *doRemT*(k, p) action in *IntAut* (which removes k from the abstract set), we choose a sequence of actions for *AbsAut* consisting of *doRemT*(k, p) followed by a *doContF*(k, p) action for each process p that is executing a *contains*(k) operation and is enabled to return *false* in the poststate of *AbsAut*.

3.5 The Forward Simulation

When defining the relationship between the states of *IntAut* and *ConcAut*, one option is to represent the relationship directly in the forward simulation. However, because in this case the relationship is quite complex, we instead reflect the state of *IntAut* within *ConcAut* by introducing two auxiliary variables, *aux_keys* and *aux_seen_out*. Then, rather than constructing the forward simulation to directly relate the *ConcAut* state and the *IntAut* state, we capture this relation as invariants of *ConcAut*, and simply require, for the forward simulation, that the auxiliary variables equal their counterparts in *IntAut*. This makes it easier to test properties using a model checker before we attempt to prove them.

ConcAut is augmented with the auxiliary variables in a straightforward manner: *aux_keys* is updated when a node is inserted into the list at line 6 of the *add* method, or is marked for deletion at line 3 of the *remove* method; and $aux\_seen\_out_p$ is updated when the *contains* method is invoked by process p, and when another process executes line 3 of the *remove* method to remove the same value p is seeking.

With the addition of the auxiliary variables, the simulation relation is quite simple. Like the backward simulation relation, it has two components, one relating data and one relating program counters of processes. The first component simply requires that the auxiliary variables of *ConcAut* equal their counterparts in *IntAut*. The second component is more complicated than in the backward simulation relation, because we must relate each program counter of *ConcAut* to a program counter value in *IntAut*.

The proof for the forward simulation is also quite straightforward: almost all of the cases of the proof were dispatched automatically using PVS strategies. The only proofs that required user interaction were those to show that whenever we choose a *do* action for *IntAut* for a given *ConcAut* action, that action's precondition holds in *IntAut*. These proofs required the introduction of high-level invariants of the concrete automaton—one for each action corresponding to a *do* action in the intermediate automaton—that show that at the point that we choose these actions in the simulation, their preconditions hold in *IntAut*.

For one interesting example, we must show that when we choose *doContF*(k, p) as the action for *IntAut*, $seen\_out_p$ is *true*. More specifically, we show that $aux\_seen\_out_p$ is *true* in the *ConcAut* state, and then use the simulation relation's requirement that *aux_seen_out* and *seen_out* are equal to infer that the *IntAut* action is enabled.

3.6 Invariants

To prove the invariants required to show that an *IntAut* action's precondition holds when we choose it in the simulation, we needed over a hundred supporting invariants and lemmas. Proving these properties was the bulk of the proof effort. We do not have room to discuss all these properties, but we invite the interested reader to consult our PVS theory files and proof scripts.

Many of these properties are "obvious": In an informal proof, we might say that some follow "by inspection" (e.g., a local variable of a process is changed only by an action of that process; the *key* field of a node is changed only by line 4 of the *add* method). Others follow immediately given that nodes are not modified unless they are locked or newly allocated (e.g., the *pred* node of a process at lines 2–9 of the *add* method is not marked). One "obvious" invariant that was more difficult to prove than we expected is that a node pointed to by *curr* is *public*, which means that it was placed into the list at some time in the past (i.e., it was allocated by some process that is not still at lines 4–6 of the *add* method with that node pointed to by *entry*). Proving this required jointly proving that *pred* is public, that the successor of any public node is public, that *entry* is public for a process at line 5 of the *remove* method, and that $entry.next = curr$ for a process at line 6 of the *add* method. These five properties mutually depend on each other, and are expressed in the `public_nodes` invariant.

The `locate_works` invariant captures properties that are guaranteed by the *locate* method because it locks the nodes and then validates the desired properties before returning to *add* or *remove*. Specifically, `locate_works` says that after returning from a call to *locate*, the process has locks on adjacent *live nodes* (i.e., public and unmarked nodes) such that the key of the first node is less than the sought-after key, and the key of the second is greater or equal. The `locate_works` invariant ensures, for example, that, before the *next* field of the *pred* node is set at line 5 of the *remove* method, that field still points to the *curr* node. The `entry_unchanged_in_rem` invariant ensures that the value written to that field is the value in the *next* field of the *curr* node, so that exactly one node (the *curr* node) is removed.

The `aux_keys_accurate` invariant states that *aux_keys* is exactly the set of keys for which there is a live node. The proof of this property is mostly straightforward because a key is inserted into *aux_keys* each time a new node containing that key is inserted into the list (thus becoming live), and removed from *aux_keys* each time a node containing the key is marked (thus ceasing to be live). The difficult case in this proof is showing that marking a node with a certain key ensures that no live node with that key exists. This case uses several additional invariants: `live_nodes_in_list` says that all live nodes are reachable from *Head*; `one_from_other` says that if two different nodes are both reachable from *Head* then one of them is reachable from the other; and `later_nodes_greater_and_public` says that if one node is reachable from another, then it has a higher key. Together these three properties imply that there is at most one live node with a given key at any time, and therefore marking one falsifies the existence of any such node, as required.

Given the `aux_keys_accurate` invariant, and the simulation relation's requirement that *ConcAut*'s *aux_keys* equals *IntAut*'s *keys*, it is easy to see that we can choose the *doContT* action when a *contains*(k) operation finds an unmarked node with key k, and therefore decides to return *true*. A *contains*(k) operation that returns *false* is more interesting. First, note that if *seen_out*$_p$ is set to *true* during a *contains*(k) operation of process p, then it remains *true* for the duration of the operation, so we are justified in choosing the *doContF* action for *IntAut* if *ConcAut* decides to return *false*.

Otherwise, assume *seen_out*$_p$ is set to *false* by *contInv*(k, p) and remains *false* throughout the operation until it decides to return *false*. This implies that there is a live node containing k when *contInv*(k, p) is executed. Because nodes are marked before being removed from the list, the node remains live throughout the operation unless it is marked. If it is marked, the action sequence chosen for *IntAut* corresponding to the marking action in *ConcAut* sets *seen_out*$_p$ to *true*, contradicting the assumption. Therefore, the node remains live throughout the operation, implying that, immediately after the *cont1* action reads *Head* into *curr*$_p$, the live node is reachable from the node indicated by *curr*$_p$. As explained below, this remains true as p walks down the list towards the live node unless the node is marked (again, this contradicts the assumption). Thus, the *contains*(k) operation finds this live node, and does not return *false* as assumed.

The above reasoning is captured in part by the `cont_val_still_in` invariant, which states that as p executes the loop at lines 2 and 3 of the *contains*(k) method, either *aux_seen_out*$_p$ is *true*, or there is a path from *curr*$_p$ to a live node m containing k. The path property is expressed as `leadsfrom`(*curr*$_p$, m), which states that there is a non-zero-length path from *curr*$_p$ to m. `leadsfrom` is defined using `leadsfromsteps` as follows to allow us to prove to PVS that inductive proofs over it are finite.

```
leadsfromsteps(c, n, m, w): INDUCTIVE bool =
  n /= Tail AND ((w=1 AND c`nextf(n) = m) OR
                 (w>1 AND c`nextf(n) /= m AND
                          leadsfromsteps(c,c`nextf(n),m,w-1)))
leadsfrom(c, n, m): bool = EXISTS w: leadsfromsteps(c, n, m, w)
```

A challenging part of the proof is proving that `leadsfrom`(*curr*$_p$, m) is falsified only from a state in which node m is marked, and therefore `cont_val_still_in` is not falsified by `leadsfrom`(*curr*$_p$, m) becoming false (because the invariant implies that m is unmarked). The intuition for this property is clear: changes outside the path between *curr*$_p$ and m have no effect, and inserting or removing a node from the list between *curr*$_p$ and m preserves the `leadsfrom`(*curr*$_p$, m) property. Thus, only removing a link to m can falsify `leadsfrom`(*curr*$_p$, m), and it is easy to prove that this occurs only after m has been marked. While the intuition for this property is straightforward, the proof is somewhat involved because it requires various inductions over the `leadsfrom` property to capture the effects of inserting and removing nodes in the middle of the list. Our decision to represent the path property using the recursive `leadsfrom` predicates was driven by our desire to minimise the size of the state space to accommodate model checking work by our colleagues (see below). Explicitly encoding the paths in auxiliary variables would simplify the proof considerably.

4 Experience with PVS

We used PVS [3] to verify all the proofs discussed in this paper. As we are not experts in the use of PVS, and we had no special support, our experience may be relevant both to others considering using PVS to verify similar proofs, and as a comparison to others' experience with different formal tools. In addition to our work with PVS, we collaborated with David Friggens and Ray Nickson, who developed models for this algorithm for use in the model checkers Spin and SAL [11,12]. As well as model checking the entire algorithm for small numbers of threads and small bounds on the queue size, which gave us some confidence that our proof attempt would eventually be successful, they used these models to test some of the putative invariants we used in our proofs before we actually proved them.

In approaching this verification, we worked mostly "top-down", starting with the simulation proofs and then proceeding with the invariants. We did not develop the basic proof using PVS; rather, we figured out the top-level invariants informally, and prepared a fairly detailed proof sketch of these invariants, and some of their supporting lemmas and invariants, before formalizing them in PVS. We did not, however, work out all the low-level lemmas and invariants that we knew would be helpful for the proofs, leaving many of them to be stated and proved as necessary.

As mentioned earlier, introducing auxiliary variables in the concrete automaton pushed the bulk of the work for this proof into the verification of invariants. The complete verification contains 165 PVS proofs: 32 typecheck constraints, 30 lemmas for the simulation proofs, and 103 invariants and supporting lemmas. Pushing most of the work into the invariants reduced the state that had to be managed within a PVS proof, because invariants are about a single automaton, while simulations are relations between two automata. Also, unlike simulation relations, invariants are straightforward to check using model checkers, so this reduced the gap between our work and that of our colleagues working with Spin and SAL.

PVS includes support for proof management, tracking which proofs have been done, and marking lemmas as proven but "incomplete" if they depend on earlier lemmas that have not yet been proved completely. This support helped us to work independently on different lemmas, which was especially helpful as the authors were spread over three countries. However, PVS manages changes at the file level—any change in a file invalidates all proofs for lemmas in that file—so we often had to rerun proofs that were unchanged. Finer-grained dependency tracking would have saved us considerable time.

PVS supports the creation of user-defined rules, called *strategies*, by combining built-in rules. These strategies can be saved and used in other proofs (or used again in a single proof). We used strategies extensively, for example, to set up the beginning of invariant proofs, which almost always begin assuming the invariant holds on an arbitrary reachable state and setting up as a goal that it holds after executing any action; to extract parts of the precondition or effect of an action; and to handle the many "trivial" cases in the forward simulation proof for concrete actions that did not correspond to any action in the intermediate automaton.

As useful as strategies are, we found that in many cases it was better to define a lemma that captured a desired property than to design a strategy to prove it. There are two advantages: First, PVS doesn't have to do the proof each time—it just uses the lemma. Second, often the way you state a property makes a significant difference in how you are able to use it. With a lemma, you can easily control how a property is stated.

One challenge in this verification was making proofs that PVS could check quickly. In particular, in an invariant proof, we typically show that the property is preserved by every action. Usually, only a few actions affect any of the variables mentioned by the property, and only those actions need to be considered; the rest obviously preserve the property. However, PVS must check all of those actions, and even a couple of seconds for each action turns into minutes for 52 actions. Thus, we stated and proved several “does not modify” lemmas, one for each variable, stating which actions actually modified that variable, and we used those lemmas extensively to avoid having PVS consider each of the other actions separately.

We also found it helpful to define functions to describe things that we wanted to refer to frequently, and especially that we might want to use in a strategy. For example, in the forward simulation proof, we defined the *action_corr* function to return, for any transition of the concrete automaton, the corresponding sequence of actions of the intermediate automaton. We also defined *pcin* and *pcout* to return, for each action, the program counters corresponding to the prestate and poststate respectively.

5 Concluding Remarks

We have developed the first complete and formal correctness proof for the LazyList algorithm of Heller *et al.* [1]. We model the algorithm and specification as I/O Automata in the PVS specification language, and proved that the algorithm implements the specification using simulation proofs developed in and checked by the PVS system.

As in previous algorithms we have verified [4,5,6,7], we found that the outcome of an operation cannot always be determined before it takes effect. Our proof uses a combination of backward and forward simulations to deal with this problem. An interesting aspect of the proof, which we have not encountered in our previous proofs, is the need for an arbitrary number of operations to be linearised after an action of a different operation.

In a related manual verification effort, Vafeiadis *et al.* [13] also present a verification of the LazyList algorithm. They add the abstract set as an auxiliary variable and add actions that perform the abstract operation and record the abstract result at the linearisation point of each operation. They then use the Rely-Guarantee proof method [14,15] to show that the implementation and the abstract set behave the same way in every execution of this augmented algorithm. Because it is impossible to correctly linearise a failed *contains* operation without knowledge of the future, this approach cannot be used to prove the linearisability of failed *contains* operations, and [13] only considers this case informally. It is noteworthy that this case accounted for a large proportion of the complexity and the effort involved in our completely machine-checked proof.

We have made our proof scripts available so that others may benefit from our experience (see `http://www.mcs.vuw.ac.nz/research/SunVUW/`). We also plan to test our hypothesis that substantial parts of our proof can be reused to verify several optimised versions of LazyList. In the longer term, we plan to continue refining our proof methodology to make it easier and more efficient to develop fully machine-checked proofs for concurrent algorithms.

Acknowledgements. We are grateful to David Friggens and Ray Nickson for useful conversations and for model checking various proposed properties.

References

1. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W., Shavit, N.: A lazy concurrent list-based set algorithm. In: 9th International Conference on Principles of Distributed Systems (OPODIS). (2005)
2. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. TOPLAS **12**(3) (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. Doherty, S.: Modelling and verifying non-blocking algorithms that use dynamically allocated memory. Master's thesis, School of Mathematical and Computing Sciences, Victoria University of Wellington (2003)
5. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In de Frutos-Escrig, D., Núñez, M., eds.: Formal Techniques for Networked and Distributed Systems — FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings. Volume 3235 of Lecture Notes in Computer Science., Springer (2004) 97–114
6. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. In Boiten, E., Derrick, J., eds.: Proc. Refinement Workshop 2005 (REFINE 2005). Volume 137(2) of Electronic Notes in Theoretical Computer Science., Guildford, UK, Elsevier (2005)
7. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: ICECCS 2005: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, Shanghai, Chin (2005) 507–516
8. Lynch, N., Tuttle, M.: An Introduction to Input/Output automata. CWI-Quarterly **2**(3) (1989) 219–246
9. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann (1996)
10. Lynch, N.A., Vaandrager, F.W.: Forward and backward simulations – part I: untimed systems. Information and Computation **121**(2) (1995) 214 – 233
11. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5) (1997) 279–295
12. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In Alur, R., Peled, D., eds.: Computer-Aided Verification, CAV 2004. Volume 3114 of Lecture Notes in Computer Science., Boston, MA, Springer-Verlag (2004) 496–500
13. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP '06: Proc. 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, New York, NY, USA, ACM Press (2006) 129–136
14. Jones, C.B.: Specification and design of (parallel) programs. In: 9th IFIP World Computer Congress (Information Processing 83). Volume 9 of FIP Congress Series., IFIP, North-Holland (1983) 321–332
15. Xu, Q., de Roever, W.P., He, J.: The rely-guarantee method for verifying shared variable concurrent programs. Formal Aspects of Computing **9**(2) (1997) 149–174