Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER Electronic Notes in Theoretical Computer Science 137 (2005) 93–110
www.elsevier.com/locate/entcs

Verifying Concurrent Data Structures by Simulation

Robert Colvin[1,2] Simon Doherty[3] Lindsay Groves[4]

School of Mathematics, Statistics and Computer Science
Victoria University of Wellington
Wellington, New Zealand

Abstract

We describe an approach to verifying concurrent data structures based on simulation between two Input/Output Automata (IOAs), modelling the specification and the implementation. We explain how we used this approach in mechanically verifying a simple lock-free stack implementation using forward simulation, and briefly discuss our experience in verifying three other lock-free algorithms which all required the use of backward simulation.

Keywords: Concurrency, Lock-free algorithms, Linearisability, I/O Automata, Simulation

1 Introduction

Concurrent implementations of data structures are designed to allow many processes to execute operations on a data structure at the same time. To prove the correctness of such implementations, we must prove that all possible interleavings of the atomic steps of these operations will produce correct results. *Linearisability* [11] is widely accepted as the appropriate correctness criterion for concurrent data structures, but does not appear to be used widely in mechanical proofs.

[1] We are grateful to Sun Microsystems for financial support, and to Mark Moir for helpful comments on this paper.
[2] Email: Robert.Colvin@mcs.vuw.ac.nz
[3] Email: Simon.Doherty@mcs.vuw.ac.nz
[4] Email: Lindsay.Groves@mcs.vuw.ac.nz

doi:10.1016/j.entcs.2005.04.026

In this paper, we present an approach to proving linearisability of concurrent data structures, based on simulation between two *Input/Output Automata* (IOAs) [14], one modelling the abstract data type specification and one modelling the implementation. We have used this approach in mechanically verifying several *lock-free* data structure implementations using Compare And Swap (CAS) as their only synchronisation mechanism.

In Section 2 we introduce linearisability. In Section 3 we introduce IOAs and define forward and backward simulation. In Sections 4 and 5 we show how to construct an abstract IOA from an abstract data structure specification and how to construct a concrete IOA from the code for a data structure implementation, and illustrate both of these constructions by constructing abstract and concrete IOAs for a concurrent stack. In Section 6 we discuss the verification of the concurrent stack implementation. In Section 7 we briefly discuss our experience in using this approach in verifying (and correcting/improving) three other lock-free algorithms, all of which required the use of backward simulation. In Section 8 we present our conclusions.

2 Linearisability

A concurrent system consists of a finite set of processes, each performing a sequence of operations involving a finite set of shared objects. Informally, a shared object is *linearisable* [11] if each operation on the object can be understood as occurring instantaneously at some point between its invocation and its completion, and its behaviour at that point is consistent with the specification for the corresponding sequential data type. This idea is formalised in [11] by modelling a concurrent system in terms of sequences of events corresponding to the invocations and responses of the operations on shared objects.

We write $X.op_p(args)$ to denote process p invoking operation op with arguments $args$ on object X, and $X.resp_p(res)$ to denote process p returning response $resp$ with result res from an operation on object X. A response *matches* a preceding invocation involving the same process and the same object, provided there are no intervening events involving that process. A *sequential history* is a sequence of matching invocation-response pairs (representing *completed* operations), possibly ending with an unmatched invocation (representing an uncompleted or *pending* operation).

A *sequential specification* for an object X is a prefix-closed set of histories involving only object X. A sequential history H is *legal* if each object subhistory $H|X$, obtained by restricting H to events involving X, belongs to the sequential specification for X.

A history H induces a partial order, $<_H$, on operations such that $a <_H b$

if the response for a occurs in H before the invocation for b. Operations not related by $<_H$ are concurrent. H is sequential iff $<_H$ is a total order.

A history H is *linearisable* if it can be extended, by adding zero or more response events, to give a history H' such that $complete(H')$, the maximal subsequence of H consisting only of invocations and matching responses, is equivalent to some legal sequential history S, called a *linearisation* of H, with $<_H \subseteq <_S$. An object X is *linearisable* if every history for X is linearisable with respect to the sequential specification for X. A concurrent system is *linearisable* if every history is linearisable with respect to the sequential specifications for its shared objects.

2.1 Understanding linearisability

The key idea underlying linearisability is that for any history of a concurrent system we can construct an equivalent sequential history.

The transformation from H to $complete(H')$ is required because some of the pending operations in H might have taken effect although their responses have not yet occurred. By adding responses for these operations and discarding any other pending operations, we only need to consider completed operations.

The relationship between H and S can be explained more clearly by augmenting H with *linearisation points* corresponding to the points at which operations in H are deemed to occur. Let $X.do\_op_p(args)$ denote the linearisation point for an operation with invocation $X.op_p(args)$, which must occur after the invocation, and before any matching response. If H' is an *augmented history* obtained by adding these linearisation points to a history H, we can construct a linearisation of H from H' as follows:

(i) For each completed operation with invocation inv, linearisation point lin and response $resp$, replace lin by $\langle inv, resp\rangle$ and delete inv and $resp$.

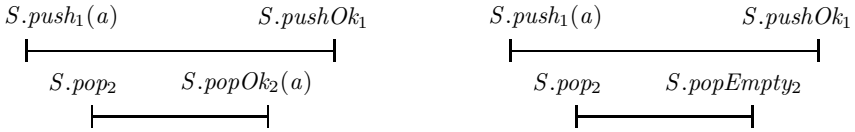
(ii) For each pending operation with invocation inv and linearisation point lin, replace lin by $\langle inv, resp\rangle$ for some legal response $resp$ and delete inv.

(iii) Delete any invocation inv with no linearisation point.

It is easy to see that this construction produces a sequential history that satisfies the conditions for linearisability given above, and that for any linearisation of a history H there is at least one corresponding augmented history. There may be several different linearisations, reflecting different choices made in extending H to H', and the fact that concurrent operations in H (and thus H') can be understood as occurring sequentially in many different orders.

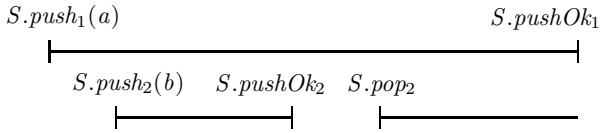
For example, consider a concurrent stack with invocations $push(v)$ and pop, and responses $pushOk$, $popOk(v)$ and $popEmpty$, where v is any value

in the component type and *popEmpty* is the response when *pop* is applied to an empty stack. We can depict possible histories for a system with a shared stack S using diagrams with time lines showing the invocations and responses of operations. For example, consider the following figure:



The diagram on the left depicts the history $\langle S.push_1(a),\ S.pop_2,\ S.popOk_2(a),\ S.pushOk_1\rangle$, which is equivalent to the legal sequential history $\langle S.push_1(a),\ S.pushOk_1,\ S.pop_2,\ S.popOk_2(a)\rangle$ obtained by linearising the push before the pop. If the pop returned b, there would be no equivalent legal sequential history, so an implementation producing this behaviour would not be linearisable. The diagram on the right depicts the history $\langle S.push_1(a),\ S.pop_2,\ S.popEmpty_2,\ S.pushOk_1\rangle$, which is equivalent to the legal sequential history $\langle S.pop_2,\ S.popEmpty_2,\ S.push_1(a),\ S.pushOk_1\rangle$ obtained by linearising the pop before the push.

Now consider the history depicted by the diagram:



The push by process 2 precedes the pop, but neither is ordered with respect to the push by process 1, and the pop is pending. This history can be linearised in five different ways by either leaving the pop incomplete or completing the pop by adding either $S.popOk_2(a)$ or $S.popOk_2(b)$, and then linearising the push operations appropriately.

3 I/O Automata and Simulation

In order to mechanically verify linearisability of concurrent data structures, we first recast the definition of linearisability in terms of Input/Output Automata (IOAs). IOAs are able to model operation invocation and response events, as well as the complex data structures used in implementations, and provides systematic proof methods which are amenable to mechanisation.

3.1 I/O Automata

An *Input/Output Automaton* (IOA) [14] is a labelled transition system, along with a signature partitioning its actions into external (input and output) and

internal actions. Formally, an IOA, A, 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, $sig(A) = (input(A), output(A), internal(A))$, which partitions $acts(A)$; and a transition relation, $trans(A) \subseteq states(A) \times acts(A) \times states(A)$.[5] We define $external(A) = input(A) \cup output(A)$.

A *(finite) execution fragment* of A is a sequence of alternating states and actions, $\pi = s_0, a_1, s_1, \ldots s_n$, such that $(s_{k-1}, a_k, s_k) \in trans(A)$ for $k \in [1, n]$. An *execution* is an execution fragment with $s_0 \in start(A)$.[6] A *trace* is the sequence of external actions in some execution. We say that two executions (not necessarily of the same automaton) are *equivalent* if they have the same trace, and we write $traces(A)$ for the set of all traces of A. We also write $trace(\beta)$ to denote the sequence of external actions in a sequence $\beta \in acts(A)^*$, 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 $(s, \alpha, s') \in trans(A)$, and for $\beta \in acts(A)^*$, we write $s \stackrel{\beta}{\Longrightarrow} s'$ to mean that there is an execution fragment beginning with s, ending with s', and containing exactly the actions of β.

In practice, 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 (the *pre-state*) and s' (the *post-state*) and action α with precondition pre_α and effect eff_α, the transition (s, α, s') is in $trans(A)$, i.e. $s \xrightarrow{\alpha} s'$, if and only if pre_α holds in s and s' is the result of applying eff_α to s. We say that an action α is *enabled* in s if pre_α holds in s. These descriptions are parameterised by processes and sometimes by other values, so they actually describe sets of transitions.

3.2 Simulation

Let A and C be IOAs with $external(C) = external(A)$. We prove $traces(C) \subseteq traces(A)$ by showing, for any execution of C, how to construct an execution of A with the same trace. The required abstract execution may be constructed either by working forwards from the start of the concrete execution, or by working backwards from the end of the (finite) execution. The key to each construction is defining a *simulation relation* between states of C and A that holds at corresponding steps in the executions of the two IOAs.

[5] The definition in [15] includes additional structure to support fairness and composition, which we do not require for this work.

[6] The full theory of I/O automata also allows infinite executions, which are necessary to reason about liveness. In this paper we are only concerned with proving linearisability, which is a safety property, so we consider only finite executions.

A *forward simulation* is a relation $R \subseteq states(C) \times states(A)$ satisfying: [7]

(i) $(\forall\, c : start(C) \bullet (\exists\, a : start(A) \bullet R(c, a)))$

(ii) $(\forall\, c, c' : states(C), a : states(A), \alpha : acts(C) \bullet$
$R(c, a) \wedge c \xrightarrow{\alpha} c' \Rightarrow$
$(\exists\, a' : states(A), \beta : acts(A)^* \bullet$
$R(c', a') \wedge a \overset{\beta}{\Longrightarrow} a' \wedge trace(\alpha) = trace(\beta)))$

A *backward simulation* is a relation $R \subseteq states(C) \times states(A)$ satisfying:

(i) $(\forall\, c : states(C) \bullet (\exists\, a : states(A) \bullet R(c, a)))$

(ii) $(\forall\, c : start(C);\ a : states(A) \bullet R(c, a) \Rightarrow a \in start(A))$

(iii) $(\forall\, c, c' : states(C);\ a' : states(A);\ \alpha : acts(C) \bullet$
$R(c', a') \wedge c \xrightarrow{\alpha} c' \Rightarrow$
$(\exists\, a : states(A);\ \beta : acts^*(A) \bullet$
$R(c, a) \wedge a \overset{\beta}{\Longrightarrow} a' \wedge trace(\alpha) = trace(\beta)))$

In our verifications, β is typically either a singleton (when α is an external action or a linearisation point) or empty (otherwise).

The existence of a forward or backward simulation from C to A guarantees that $traces(C) \subseteq traces(A)$. Trace inclusion cannot always be proved using either forward or backward simulation; some cases need a forward simulation from C to an intermediate IOA, B, and a backward simulation from B to A [15]. This is similar to the results given in [13] and [1].

4 Abstract Stack Automaton

Given a data type $\mathcal{D}$, we construct an "abstract" IOA which generates exactly the linearisable histories for a set of processes operating on a shared object of type $\mathcal{D}$. This construction is based on the canonical atomic object automaton construction described in [14, Chapter 13], extended to apply to arbitrary objects and simplified due to our assumption that the subhistory for each process is sequential. A more detailed description of the construction is given in [7].

[7] These conditions are essentially as given in [14, p225], but stated more formally, and rearranged slightly. Likewise for backward simulation.

4.1 Specifying data types

We specify a data type by giving its set of values, an initial value, sets of invocations and responses for the data type operations, and an update function mapping an invocation and a given value to a response and a new value.

For example, the type of stacks with component type T is defined as $Stack_T = (V, v_0, I, R, f)$ where:

- V is seq T, the set of sequences over T
- v_0 is the empty sequence, $\langle\rangle$
- $I = \{push(v) \mid v \in T\} \cup \{pop\}$
- $R = \{pushOK, popEmpty\} \cup \{popOk(v) \mid v \in T\}$
- f is defined by the following equations:

$$f(push(v), s) = (pushOk, \langle v\rangle \frown s)$$

$$f(pop, s) = \begin{cases} (popOk(head(s)), tail(s)), \text{ if } s \neq \langle\rangle \\ (popEmpty, \langle\rangle), \text{ if } s = \langle\rangle \end{cases}$$

4.2 Canonical abstract automaton

The abstract IOA, *AbStack*, for a finite set of processes, $PROC$, acting on a shared stack, has external actions corresponding to the invocations in I and the responses in R, which simply update the program counter for the relevant process to ensure that actions occur in the correct order. It also has internal actions $doPush(v)$, for all $v \in T$, and $doPop$, corresponding to linearisation points, where these operations take effect, and *doPopEmpty* corresponding to linearisation point for pop on an empty stack. Thus, $doPush(v)$ pushes v onto the stack, and $doPop$ pops the stack provided it is non-empty.

The state space is $(\text{seq } T) \times \Pi_{p \in PROC} AbsPCVals$, where $AbsPCVals = \bigcup_{v:Val}\{idle, DoPush(v), PushOk, DoPop, PopOk(v), PopEmpty\}$ is the set of abstract counter values. The program counter for a process is *idle* if the process is idle, and thus ready to perform an invocation action. The other values indicate that the process is ready to perform the indicated internal or response action. Initially, all processes are idle and the stack is empty: $start(AbStack) = \{a : states(AbStack) \mid (\forall\, p : PROC \bullet a.pc_p = idle) \wedge a.Stack = \langle\rangle\}$. The transitions are defined as shown in Fig. 1.

For each process p, *AbStack* generates a sequence of actions consisting of a concatenation of subsequences of the form $\langle push_p(v), doPush_p(v), pushOk_p\rangle$, $\langle pop_p, doPop_p, popOk(v)_p\rangle$ or $\langle pop_p, doPopEmpty_p, popEmpty_p\rangle$. Thus, every trace when restricted to a single process is a concatenation of sequences of the form $\langle push_p(v), pushOk_p\rangle$, $\langle pop_p, popOk(v)_p\rangle$ or $\langle pop_p, popEmpty_p\rangle$, and is

$push_p(v)$: **pre:** $pc_p = idle$
eff: $pc_p \coloneqq DoPush(v)$

$doPush_p(v)$: **pre:** $pc_p = DoPush(v)$
eff: $Stack \coloneqq \langle v \rangle \frown Stack$,
$pc_p \coloneqq PushOk$

$pushOk_p$: **pre:** $pc_p = PushOk$
eff: $pc_p \coloneqq idle$

pop_p: **pre:** $pc_p = idle$
eff: $pc_p \coloneqq DoPop$

$doPop_p$: **pre:** $pc_p = DoPop \wedge Stack \neq \langle\rangle$
eff: $Stack \coloneqq tail(Stack)$,
$pc_p \coloneqq PopOk(head(Stack))$

$doPopEmpty_p$: **pre:** $pc_p = DoPop \wedge Stack = \langle\rangle$
eff: $pc_p \coloneqq PopEmpty$

$popOk_p(v)$: **pre:** $pc_p = PopOk(v)$
eff: $pc_p \coloneqq idle$

$popEmpty_p$: **pre:** $pc_p = PopEmpty$
eff: $pc_p \coloneqq idle$

Fig. 1. Abstract Automaton Transitions

therefore a sequential history for a stack object.

The actions from various processes can be interleaved in all possible ways that are legal with respect to the stack specification. Thus, the executions produced by the stack IOA are precisely the augmented histories described in Section 2, and are linearisable for the reasons given there. We can show that the canonical automata construction described above produces an IOA that generates exactly the linearisable histories for a given data type; see [7] for details.

The construction of canonical automata presented here differs from the construction presented in [14, Section 13.2] in certain respects. Firstly, in [14] indices on invocations and responses are interpreted as representing *ports*, rather than processes, which is appropriate in modelling distributed systems. Secondly, we treat all external actions as outputs. This is done to side-step the requirement that IOAs be *input-enabled*, i.e. that every input action is always enabled. Input-enabled is not required in our context, since a process cannot begin a new stack operation while it is already performing an operation on the stack.

5 Concrete Stack Automaton

We now show how to construct a "concrete" IOA to model a set of processes operating on a concurrent data structure from the code for the implementation.

This IOA has the same external actions as the abstract IOA, and internal actions corresponding to atomic steps in the implementation. There is typically (provided they involve at most one shared variable access) one internal action for each assignment statement, and one each for the true and false branches of each test. The state of the concrete IOA has shared variables corresponding to the shared variables used in the code, and, for each process,

```
type  Node = {val : T; next :↑ Node};
shared Head :  ↑ Node := null;

push(v : T) ≙                          pop() : T ≙
1 n ≔ new(Node);                       1 repeat
2 n.val ≔ v;                           2     ss ≔ Head;
3 repeat                               3     if ss = null then
4     ss ≔ Head;                       4         return empty;
5     n.next ≔ ss;                     5     ssn ≔ ss.next;
6 until CAS(Head, ss, n)               6     lv ≔ ss.val;
                                       7 until CAS(Head, ss, ssn);
                                       8 return lv
```

Fig. 2. Code for stack implementation

a program counter and any local variables used in the code for the operations. Sometimes additional variables are added to aid in describing the effects of actions or to simplify proofs.

5.1 A lock-free stack implementation

We will illustrate the construction by constructing a concrete IOA for a simple lock-free stack implementation,[8] which is a simplified version of the one given in [18]. The implementation, expressed in Pascal-like pseudo-code form, is show in Fig. 2.

The stack is stored as a standard linked list of nodes with fields *val* and *next*, with a shared variable *Head* pointing to the head of the list. *Head* is *null* when the stack is empty (so is initialised to *null*), and points to the node containing the top of the stack when the stack is non-empty.

The *push* operation creates a new node and stores the value to be pushed in its *val* field. It then attempts to link the new node into the list. It takes a snapshot of *Head* in the local variable *ss*, sets the *next* field of the new node to this value, then attempts to make *Head* point to the new node. This will produce the correct result only if *Head* has not changed since the snapshot was taken, so a CAS[9] is used to atomically test whether *Head* is still the same as *ss* and if so change it to *n*, in which case the operation is complete. If *Head* has changed, the operation loops back to line 3 and tries again.

The *pop* operation starts by taking a snapshot of *Head*. It then tests whether this value is *null* and if so returns a special value *empty* indicating

[8] An implementation is *lock-free* if, whenever an operation takes infinitely many steps attempting to complete an operation, infinitely many other operations complete. Some authors call this property *nonblocking*; others use nonblocking as a more general term encompassing other progress conditions such as wait-freedom and obstruction-freedom [10].

[9] A 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.

that the stack was empty. Otherwise, *pop* copies the *next* and *val* fields from the node pointed to by *ss*, and attempts to update *Head* to point to the value in the *next* field of the node it points to. This will only produce the correct result if *Head* has not changed since the snapshot was taken, so a CAS is used to atomically test whether *Head* is still the same as *ss* and if so change it to *ssn*, in which case the operation is complete and *lv* can be returned as the popped value. If *Head* has changed, the operation loops back to line 1 and tries again.

Since the only synchronisation is through CAS instructions, which are supported by most modern multiprocessors, there is no danger of a process dying while holding a lock or of two processes endlessly waiting upon each other. Instead, if the actions of another process interfere with the operation a process is performing, this is detected by *Head* having changed so the process simply retries its operation.

A subtle point, not apparent in the code, is that the correctness of the algorithm relies upon the fact that a successful pop operation does not free the memory used by the popped node, since other processes might also be trying to pop that node. The algorithm would also be correct if memory was recycled by a garbage collection mechanism that only reclaims storage when there are no pointers to it, but then to claim that the implementation is lock-free, we would have to show that the garbage collector was lock-free (see [6]). Other ways of recycling memory use version numbers to detect when storage has been recycled (e.g. see [16]).

5.2 Constructing the concrete automaton

We now construct a concrete automaton for the above stack implementation. The state of the concrete IOA has variables to represent the heap and, for each process, a program counter and local variables n, lv, ss and ssn, used in the push and pop code.

We model the heap using three sets, *Val* of values, *Loc* of locations and $Ptr \mathrel{\widehat{=}} Loc \cup \{\mathsf{null}\}$ (where $\mathsf{null} \notin Loc$), and shared variables, *Head* of type *Loc*, *val* a function from *Loc* to *Val*, *next* a function from *Loc* to *Ptr*, and $free \subseteq Loc$ a set of unallocated locations. To simplify the verification, we also add an auxiliary variable, *list*, which holds a list of the pointers to the nodes in the linked list. Initially, all processes are idle, *Head* is null, *next* and *val* are empty functions, $free = Loc$ and $list = \langle\rangle$.

As in the abstract automaton, a process's program counter is *idle* if the process is idle and thus ready to perform any invocation action. Otherwise, it has a value corresponding to a line number in the code or a response it is ready to return.

$push_p(v)$: **pre:** $pc_p = idle$
eff: $lv_p \coloneqq v$,
$pc_p \coloneqq Push1$

$push1_p$: **pre:** $pc_p = Push1$
eff: $n_p \coloneqq nextfree(free)$,
$free \coloneqq free - \{nextfree(free)\}$,
$pc_p \coloneqq Push2$

$push2_p$: **pre:** $pc_p = Push2$
eff: $val \coloneqq val \oplus \{n_p \mapsto lv_p\}$,
$pc_p \coloneqq Push4$

$push4_p$: **pre:** $pc_p = Push4$
eff: $ss_p \coloneqq Head$,
$pc_p \coloneqq Push5$

$push5_p$: **pre:** $pc_p = Push5$
eff: $next \coloneqq next \oplus \{n_p \mapsto ss_p\}$,
$pc_p \coloneqq Push6$

$push6t_p$: **pre:** $pc_p = Push6 \wedge Head = ss_p$
eff: $Head \coloneqq n_p$,
$list \coloneqq \langle n_p \rangle \frown list$,
$pc_p \coloneqq PushOk$

$push6f_p$: **pre:** $pc_p = Push6 \wedge Head \neq ss_p$
eff: $pc_p \coloneqq Push4$

pop_p: **pre:** $pc_p = idle$
eff: $pc_p \coloneqq Pop2$

$pop2_p$: **pre:** $pc_p = Pop2$
eff: $ss_p \coloneqq Head$,
$pc_p \coloneqq Pop3$

$pop3t_p$: **pre:** $pc_p = Pop3 \wedge ss_p = \mathsf{null}$
eff: $pc_p = PopEmpty$

$pop3f_p$: **pre:** $pc_p = Pop3 \wedge ss_p \neq \mathsf{null}$
eff: $pc_p = Pop5$

$pop5_p$: **pre:** $pc_p = Pop5$
eff: $ssn_p \coloneqq next(ss_p)$,
$pc_p \coloneqq Pop6$

$pop6_p$: **pre:** $pc_p = Pop6$
eff: $lv_p \coloneqq val(ss_p)$,
$pc_p \coloneqq Pop7$

$pop7t_p$: **pre:** $pc_p = Pop7 \wedge Head = ss_p$
eff: $Head \coloneqq ssn_p$,
$list \coloneqq tail(list)$,
$pc_p \coloneqq PopOk(lv_p)$

$pop7f_p$: **pre:** $pc_p = Pop7 \wedge Head \neq ss_p$
eff: $pc_p = Pop2$

Fig. 3. Concrete Automaton Transitions

The main transitions are shown in Fig. 3 (transitions $pushOk_p$, $popEmpty_p$ and $popOk_p(v)$ are the same as in Fig. 1). Storage is allocated using the partial function $nextfree : \mathbb{P}\, Loc \nrightarrow \mathbb{P}\, Loc$, which satisfies the following property:

$$(\forall\, s : \mathbb{P}\, Loc \bullet s \neq \varnothing \Rightarrow nextfree(s) \in s)$$

In our verification, we assume that the system does not run out of memory, so *nextfree* is never applied to the empty set.

We claim that this IOA generates exactly the executions that can be produced by a set of processes executing the push and pop operations for the above stack implementation.

6 Verification

To show that the stack implementation is linearisable, we define a forward simulation, R, showing that every history for the stack implementation (i.e. every trace of the concrete IOA) is also a history for the stack specification (i.e.

a trace of the abstract IOA). R is defined in two parts: an *abstraction relation* relating the abstract and concrete stack values, and a *step correspondence* relating the abstract and concrete program counter values:

$$R(c, a) \mathrel{\widehat{=}} abs(c, a) \wedge steps(c, a)$$

The abstraction relation is quite simple due to the inclusion of the auxiliary variable *list*. We can construct the abstract stack by applying the *val* function to each of the pointers in *list*.

$$abs(c, a) \mathrel{\widehat{=}} map(c.val, c.list) = a.Stack$$

The step correspondence is defined in terms of a function *step* giving the program counter a process must have in the abstract state, given its program counter in the concrete state.

$$steps(c, a) \mathrel{\widehat{=}} (\forall\, p : PROC \bullet a.pc_p = step(p, c))$$

where:

$$step(p, c) = \begin{cases} c.pc_p & \text{if } c.pc_p \in \bigcup_{v:Val}\{idle, PushOk, PopOk(v), PopEmpty\} \\ DoPush(c.lv_p) & \text{if } c.pc_p \in \{Push1, Push2, Push4, Push5, Push6\} \\ DoPop & \text{if } c.pc_p \in \{Pop2, Pop5, Pop6, Pop7\} \\ DoPop & \text{if } c.pc_p = Pop3 \wedge c.ss_p \neq \mathsf{null} \\ PopEmpty & \text{if } c.pc_p = Pop3 \wedge c.ss_p = \mathsf{null} \end{cases}$$

When p is ready to perform an external action in concrete state c, p must be ready to perform the same action in abstract state a. When p is at line 1 to 6 of a push operation in c, in a, p is ready to perform a $DoPush(v)$, where the value to be pushed is given by p's variable lv in c. When p is at line 2, 5, 6 or 7 of a pop operation in c, in a, p is ready to perform a $DoPop(v)$ in a. The same is true if p is at line 3 in c and p's variable ss is not null. However, when p is at line 3 in c and p's variable ss is null, p is ready to perform a $PopEmpty$ in a.

In this example, we can determine exactly the program counter a process will have in the abstract state from its program counter in the concrete state. For more complex algorithms, *step* returns a set of possible abstract program counter values, and $steps(c, a)$ is defined as $(\forall\, p : PROC \bullet a.pc_p \in step(p, c))$.

In defining *step*, it is helpful to understand the relationship between actions of the concrete IOA and those of the abstract IOA. In the proof, this relationship is used to provide a witness for the existentially quantified variable β in the definition of forward simulation. In defining this relationship, we note:

- When the concrete IOA performs an external action, the abstract IOA must perform the same action (so that the traces of the two IOAs are the same).
- When the concrete IOA performs an internal action corresponding to a linearisation point, the abstract IOA must perform the corresponding abstract linearisation point action.
- When the concrete IOA performs any other internal action, the abstract IOA does not perform any action (these are often called stuttering steps).

The key to defining this *action correspondence* is identifying the internal actions that correspond to linearisation points. This is quite straightforward for a push or a successful pop — the linearisation point is the CAS that updates the shared variable *Head*, i.e. $push6t_p$ or $pop7t_p$. It is more subtle for an unsuccessful pop — the linearisation point is the step where an empty stack was observed, which in this case is $pop2_p$, but only when the value of *Head* read is *null*.

For the stack algorithm, we can determine the abstract action sequence from the concrete action and the concrete state, so we can define the action correspondence using the following function:

$$h(\alpha, c) \mathrel{\widehat{=}} \begin{cases} \langle \alpha \rangle & \text{if } \alpha \in external(ConcStack) \\ \langle doPush_p(c.lv_p) \rangle & \text{if } \alpha = push6t_p \\ \langle doPop_p \rangle & \text{if } \alpha = pop7t_p \\ \langle doPopEmpty_p \rangle & \text{if } \alpha = pop2_p \wedge c.Head = \mathsf{null} \\ \langle \rangle & \text{otherwise} \end{cases}$$

With the simulation relation defined above, we are able to prove the conditions for forward simulation, using h to provide witnesses for β and using the semantics of β to calculate the corresponding witness for a' (since all of our actions are deterministic), and thus show that the stack implementation is linearisable. The proof relies on a number of invariants capturing key properties of the variables in the concrete IOA, including values of local variables, integrity properties of the dynamic storage structure, and the value of the auxiliary variable *list*.

7 Experience with other Verifications

We have verified several lock-free data structure implementations using the approach described above. The proofs have been mechanised using the PVS theorem prover [4] and share a number of common theories, most notably the IOA theory which embodies the definitions of IOAs and simulation. We have developed a suite of PVS strategies which allow most of the proof obligations in these proofs to be discharged automatically, leaving only a small proportion requiring user direction. We describe three of these verifications below.

7.1 Michael and Scott's queue

Michael and Scott's lock-free queue [16] is one of the most well-know and most widely used lock-free algorithms. It represents a queue using a linked list with a header node and uses CAS as its synchronisation primitive. An enqueue operation has to update two shared variables: the tail pointer and the *next* field of the last node (the header node ensures that the last node always exists and the tail pointer is never null). Since a CAS can only atomically update one location, the tail pointer is allowed to lag by up to one node. Enqueue operations, and certain dequeue operations, check to see whether the tail pointer is lagging (which indicates that another enqueue has appended a new node but not yet updated the tail) and if so assists the pending operation by advancing the tail pointer.

Our verification [9] used a backward simulation between the abstract IOA and an intermediate IOA, and a forward simulation between the intermediate IOA and the concrete IOA. The intermediate IOA differs from the abstract IOA only in its handling of a dequeue operation returning *empty* (analogous to a pop operation on an empty stack returning *empty*). The forward simulation deals with the linked list implementation of the queue, and is similar to the one described in this paper. The backward simulation is a little different. In a backward simulation, states are matched moving backwards through an execution, starting from some (reachable) state. This is typically a less intuitive process than forward simulation, and its use appears to be much less common. In our verification of the Michael and Scott queue, backward simulation is required because the point in the dequeue operation where an empty queue is observed does not necessarily lead to a response of empty, that is, the *empty* queue linearisation point is not always a linearisation point; sometimes it is a stuttering step. Using forward simulation, at the point where an empty queue is observed it is impossible to determine whether or not the operation will eventually respond with empty — the nondeterminism cannot be resolved and the verification cannot proceed. However, using backward simulation, one

can work back from the concrete state immediately after an empty queue is observed. Then the program counter value of the corresponding abstract state will indicate if the abstract IOA took the *empty* response transition. It is interesting that the verification of the Michael and Scott queue, and of two other algorithms discussed in this section, requires the less intuitive notion of backward simulation.

The verification effort revealed a small optimisation which reduces the number of shared reads required in performing a dequeue operation.

7.2 Shann et al's queue

Shann, Huang and Chen [17] describe an array implementation of a bounded queue, also using CAS. The algorithm allows both head and tail pointers to lag behind the actual front and back of the queue, and operations check whether the relevant pointer is correct, and update it if necessary, before performing their modification. Enqueues on a full queue and dequeues on an empty queue wait until the operation can be performed, so lock-free behaviour is not guaranteed in these cases. The algorithm thus avoids the issues associated with boundary cases, which necessitated the use of backward simulation in verifying Michael and Scott's queue implementation.

Our verification effort [3] revealed two symmetrical bugs which meant that sometimes the implementation did not behave correctly when attempting to update an inaccurate front or tail pointer. Using the failed verification to produce a counter-example, we were able to understand the cause of the bugs and provide a fix for them. The fixed algorithm was then verified using a forward simulation similar to that described in this paper. We then modified the algorithm to provide lock-free behaviour in all cases, by returning special values to indicate that an enqueue was performed on a full queue or a dequeue on an empty queue. This verification required a backward simulation for the reasons discussed in Section 7.1. In this case, however, since the way we modelled the array was similar to the way we modelled the queue, we performed the backward simulation directly between the abstract and concrete IOAs without using an intermediate IOA.

7.3 Detlefs et al's deque

Detlefs *et al* [5] describe a doubly linked list implementation of a concurrent deque, using DCAS as its synchronisation primitive. [10] This algorithm, known

[10] DCAS is like CAS, except that it atomically tests and updates two locations at once. DCAS is not widely available on current architectures, and this algorithm was developed as part of an attempt to determine whether it would be worth implementing — see [8].

as Snark, makes clever use of sentinels at both ends of the doubly linked list to reduce the number of special cases required, and the number of accesses to shared variables.

Our initial verification effort [7] revealed a very subtle bug which meant that two processes performing pops from opposite ends of the deque could both succeed in popping the same element. We fixed the algorithm by allowing both processes to proceed as though their pop was successful and decide later which process would actually get to return the value. Although this correction was conceptually relatively simple, it required a major revision of the verification. The revised version required a backward simulation, again for reasons similar to those discussed in Section 7.1, with the intermediate IOA representing the deque in the same way as the abstract IOA, and the forward simulation between the intermediate and concrete IOAs dealing with the dynamic data structure. In this case, however, the intermediate IOA also introduced a data structure to register invocations of operations, to be used in resolving conflicts between operations attempting to pop the same value from the deque. Also, in some cases the linearisation point for one operation turned out to be a step performed by another process. Our initial verification and an informal description of the correction is given in [7]. The corrected version and its verification are discussed briefly in [8]; a more detailed description is in preparation.

8 Conclusions

We have presented an approach to verifying concurrent data structures by simulation between Input/Output Automata and illustrated the approach by showing how it is applied to the verification of a simple lock-free stack algorithm. We have also discussed our experience in using this approach in verifying three other published algorithms; in each case our verification effort revealed either optimisations that could be applied or bugs that we then corrected. This experience demonstrates the effectiveness of our approach, and highlights some interesting aspects of the application of familiar theoretical results to realistic applications, including the fact that most of the verifications required backward simulations.

Although our approach has been shown to be effective, each verification is still a significant undertaking, and we are continuing to investigate ways of improving and/or complementing our existing techniques. We are continuing to develop strategies for automation of our proofs, as we gain experience with more proofs and see greater opportunities for generalisation and reuse. Having twice expended considerable effort in attempting to verify published al-

gorithms that turned out to contain bugs, we now use the Spin model checker [12] to test any algorithms we attempt to verify. We are also investigating the possibility of constructing finite abstractions of lock-free algorithms that would allow exhaustive model checking (cf. [19]).

We are also interested in finding ways in which we can simplify our proofs by using constructive approaches based on refinement. This might involve identifying common steps in the construction of nonblocking algorithms, and perhaps using a different formalism such as action systems. A similar endeavour has been undertaken by Abrial and Cansell, using Event B (see [2]).

References

[1] Abadi, M. and L. Lamport, *The existence of refinement mappings*, Theoretical Computer Science **82** (1991), pp. 253–284.

[2] Abrial, J.-R. and D. Cansell, *Formal construction of a non-blocking concurrent queue algorithm*, Seminar on Proving Pointer Programs, LORIA (2004).
URL http://www.loria.fr/~cansell/qsl/qsl26-02-2004.html

[3] Colvin, R. and L. Groves, *Formal verification of an array-based nonblocking queue*, in: *ICECCS 2005: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems*, 2004, to appear.

[4] Crow, J., S. Owre, J. Rushby, N. Shankar and M. Srivas, *A tutorial introduction to PVS*, in: *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, 1995.
URL http://www.csl.sri.com/papers/wift-tutorial/

[5] Detlefs, D., C. H. Flood, A. Garthwaite, P. Martin, N. N. Shavit and G. L. Steele, Jr., *Even better DCAS-based concurrent deques*, in: *In Proceedings of the 14th International Conference on Distributed Computing* (2000), pp. 59–73.

[6] Detlefs, D. L., P. A. Martin, M. Moir and G. L. Steele, Jr., *Lock-free reference counting*, Distrib. Comput. **15** (2002), pp. 255–271.

[7] 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).

[8] Doherty, S., D. Detlefs, L. Groves, C. H. Flood, V. Luchangco, P. A. Martin, M. Moir, N. Shavit and G. L. Steele, Jr., *DCAS is not a silver bullet for nonblocking algorithm design*, in: P. B. Gibbons and M. Adler, editors, *SPAA 2004: Proceedings of the Sixteenth Annual ACM Symposium on Parallel Algorithms, June 27-30, 2004, Barcelona, Spain* (2004), pp. 216–224.

[9] Doherty, S., L. Groves, V. Luchangco and M. Moir, *Formal verification of a practical lock-free queue algorithm.*, in: D. de Frutos-Escrig and M. Núñez, editors, *Formal Techniques for Networked and Distributed Systems — FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings*, Lecture Notes in Computer Science **3235** (2004), pp. 97–114.

[10] Herlihy, M., V. Luchangco and M. Moir, *Obstruction-free synchronization: Double-ended queues as an example*, in: *ICDCS '03: Proceedings of the 23rd International Conference on Distributed Computing Systems* (2003), p. 522.

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

[12] Holzmann, G. J., *The model checker SPIN*, IEEE Trans. Softw. Eng. **23** (1997), pp. 279–295.

[13] Jifeng, H., C. Hoare and J. Sanders, *Data refinement refined*, in: *ESOP 86*, Lecture Notes in Computer Science **213**, Springer-Verlag, 1986 pp. 187–196.

[14] Lynch, N. A., "Distributed Algorithms," Morgan Kaufmann, 1996.

[15] Lynch, N. A. and F. W. Vaandrager, *Forward and backward simulations – part I: untimed systems.*, in: *135*, Centrum voor Wiskunde en Informatica (CWI), ISSN 0169-118X, 1993 p. 35.

[16] Michael, M. M. and M. L. Scott, *Simple, fast, and practical non-blocking and blocking concurrent queue algorithms*, in: *Symposium on Principles of Distributed Computing*, 1996, pp. 267–275.

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

[18] Treiber, R. K., *Systems programming: Coping with parallelism*, Technical Report RJ 5118, IBM Almaden Research Center (1986).

[19] Yahav, E. and S. Sagiv, *Automatically verifying concurrent queue algorithms*, SoftMC 2003: Workshop on Software Model Checking, Electronic Notes in Theoretical Computer Science **89** (2003).