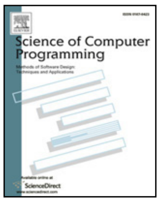
Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico



A general technique for proving lock-freedom

Robert Colvin*, Brijesh Dongol

ARC Centre for Complex Systems, School of Information Technology and Electrical Engineering, The University of Queensland, Australia

ARTICLE INFO

Article history:
Received 14 May 2008
Received in revised form 29 August 2008
Accepted 22 September 2008
Available online 1 October 2008

Keywords:
Lock-free programs
Concurrency
Verification
Temporal logic

ABSTRACT

Lock-freedom is a property of concurrent programs which states that, from any state of the program, eventually some process will complete its operation. Lock-freedom is a weaker property than the usual expectation that eventually *all* processes will complete their operations. By weakening their completion guarantees, lock-free programs increase the potential for parallelism, and hence make more efficient use of multiprocessor architectures than lock-based algorithms. However, lock-free algorithms, and reasoning about them, are considerably more complex.

In this paper we present a technique for proving that a program is lock-free. The technique is designed to be as general as possible and is guided by heuristics that simplify the proofs. We demonstrate our theory by proving lock-freedom of two non-trivial examples from the literature. The proofs have been machine-checked by the PVS theorem prover, and we have developed proof strategies to minimise user interaction.

1. Introduction

Lock-freedom is a progress property of non-blocking concurrent programs which ensures that the system as a whole makes progress, even if some processes never make progress [14,15,8]. Lock-free programs tend to be more efficient than their lock-based counterparts because the potential for parallelism is increased [15]. However, compared to lock-based implementations, lock-free programs are more complex [15,12], and can be error-prone (see, e.g., [3] for discussion on an incorrect published algorithm). Although formal safety proofs for lock-free algorithms exist [7,3,4], formal proofs of progress have mostly been ignored.

Our definition of lock-freedom is based on a literature survey and formal definition given by Dongol [8] and states that from any state of the program, eventually some process executes the final line of code of its operation, i.e., executes a *completing* action. The initial step in proving this property is to identify a set of *progress* actions (which includes the *completing* actions), and prove that if eventually some process executes a progress action, it follows that eventually some process will execute a *completing* action. Having widened the set of actions of interest, we augment the program with an auxiliary variable which detects the execution of progress actions. We use this variable to construct a *well-founded relation* on states of the program, such that every transition of the program results in an improvement with respect to the relation, or is the execution of a progress action.

The steps outlined above are guided by heuristics, and in particular by inspection of the control-flow graphs of the program. The final step involves case analysis on the actions of the system and only requires simple propositional reasoning. The proofs are supported by the theorem prover PVS: we provide a theory for showing that the relations are well-founded, and strategies for minimising user interaction in the case analysis.

Overview. This paper is organised as follows. In Section 2 we present the theoretical background to the rest of the paper. In Section 3 we present the technique and apply it to a running example, the Michael & Scott lock-free queue [15]. In Section 4

* Corresponding author.
E-mail addresses: robert@itee.uq.edu.au (R. Colvin), brijesh@itee.uq.edu.au (B. Dongol).

0167-6423/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2008.09.013

$$\mathcal{P} \;\widehat{=}\; \| \, p{:}\, procs(\mathcal{P}) \bullet \texttt{while true do } \{ \sqcap \, op{:}\, OP(\mathcal{P}) \bullet op \}$$

$$op \;\widehat{=}\; op.preLoop;\; op.Loop;\; op.postLoop$$

Fig. 1. Typical structure of lock-free algorithms.

we apply the technique to a more complex example, a bounded array-based queue [3]. In Section 5 we describe how the proofs can be checked using the PVS theorem prover [18], using a number of strategies for guiding the proofs to minimise user interaction. We conclude and discuss related work in Section 6.

2. Preliminaries

We describe the programming model and the general structure of a lock-free program in Section 2.1; provide the lock-free queue by Michael and Scott as a concrete example in Section 2.2; briefly review well-founded relations in Section 2.3; and describe transition systems, trace-based reasoning and temporal logic in Section 2.4.

2.1. Lock-free programs

The general structure of a lock-free program is summarised by the program, $\mathcal{P}$, in Fig. 1, where $\mathcal{P}$ is formed by a finite set of parallel processes, $procs(\mathcal{P})$, that execute operations from $OP(\mathcal{P})$. Each process p selects some operation op for execution. After op completes, p makes another selection, and so on. This form of $\mathcal{P}$ is an abstraction of a real lock-free program, where processes are likely to perform other functions between calls to operations in $\mathcal{P}$. We assume that functions external to $\mathcal{P}$ do not alter its variables, and hence may be ignored for the purposes of defining $\mathcal{P}$. We say a process is *idle* if the process is not executing any operation, i.e., is between calls to operations. Note that because each process continually chooses new operations for execution, each execution of $\mathcal{P}$ is infinite in length.

Each operation $op \in OP(\mathcal{P})$ is of the form described in Fig. 1. They are sequential non-blocking statements, i.e., no atomic statement of op exhibits blocking behaviour. The main part of the operation occurs in $op.Loop$, which is a potentially infinite retry-loop. Given that $op.Loop$ modifies shared data G, the typical structure of $op.Loop$ is to take a snapshot of G, construct a new value for G locally, then attempt to update G to this new value. If no interference has occurred, i.e., G has not been modified (by a different process) since the snapshot was taken, $op.Loop$ terminates, whereas if interference has occurred $op.Loop$ is retried. Because $op.Loop$ is retried based on interference from external sources, a loop variant, for proving loop termination in the traditional way, cannot be derived.

An operation op may also require some pre- and post-processing tasks to $op.Loop$, as given by $op.preLoop$ and $op.postLoop$. Both $op.preLoop$ and $op.postLoop$ are assumed not to contain potentially infinite loops. Note that $op.preLoop$ and $op.postLoop$ may be empty for some operations. Because $op.Loop$ may contain multiple exit points, there could be multiple control flows within $op.postLoop$, i.e., the structure in Fig. 1 is simplified for descriptive purposes.

Within an operation, each atomic statement has a unique *label*, and each process p has a *program counter*, denoted pc_p, whose value is the label of the next statement p will execute. We assume the existence of a special label idle to identify idle processes. We use $PC(\mathcal{P})$ to denote the set of all labels in the program (including idle).

Lock-free programs are typically implemented using hardware primitives such as Compare-and-Swap (CAS), which combines a test and update of a variable within a single atomic statement. A procedure call CAS(G, ss, n) operates as follows: if (shared) variable G is the same as (snapshot) variable ss, then G is updated to the value of n and *true* is returned; otherwise no update occurs and *false* is returned. CAS instructions are available on many current architectures, including x86.

```
CAS(G, ss, n) ≙ if G = ss
                then G := n ; return true
                else return false
```

CAS-based implementations can suffer from the "ABA problem" [15], which can be manifested in the following way. A process p takes a snapshot, say ss_p, of the shared variable G when G's value is A; then another process modifies G to B, then back again to A. Process p has been interfered with, but the CAS that compares G and ss_p cannot detect the modification to G because $G = ss_p$ holds after the interference has taken place. If value A is a pointer, this is a potentially fatal problem because the contents of the location pointed to by A may have changed. A work-around is to store a *modification counter* with each shared variable, which is incremented whenever the variable is modified. If the modification counter is also atomically compared when executing the CAS, any interference will be detected. As discussed by Michael and Scott [15], modification counters do not constitute a full solution to the ABA problem in a real system because modification counters will be bounded. However, in practice, the likelihood of the ABA problem occurring is reduced to a tolerably small level [17].

2.2. Example: The Michael and Scott queue

To understand how typical lock-free operations are implemented, consider the program $\mathcal{MSQ}$ in Fig. 2, which is the lock-free queue algorithm presented by Michael and Scott [15]. The algorithm is used as the basis of the implementation

```
node ≙ {val: Val; next: ptrmc }
ptrmc ≙ {ptr: ptr_to node; mod: ℕ }
Head, Tail: ptrmc
```

```
enq(v: Val)

E1:  node := new_node();
E2:  node->val := v;
E3:  node->next.ptr = null;
     loop
E5:    tail := Tail;
E6:    nxt := tail.ptr->next;
E7:    if tail = Tail
E8:      if nxt.ptr = null
E9:        if CAS(tail.ptr->next, nxt,
                  <node, nxt.mod+1>)
E10:         break
           fi
         else
E13:       CAS(Tail, tail,
               <nxt.ptr, tail.mod+1>)
         fi
       fi
     endloop;
E17: CAS(Tail, tail,
         <nxt.ptr, tail.mod+1>)
```

```
deq(vp: ptr_to Val)

     loop
D2:    head := Head;
D3:    tail := Tail;
D4:    nxt := head->next;
D5:    if head = Head
D6:      if head = tail
D7:        if nxt.ptr = null
D8:          return FALSE
           fi;
D10:       CAS(Tail,tail,
               <nxt.ptr, tail.mod+1>)
         else
D12:       *vp := nxt.ptr->val;
D13:       if CAS(Head, head,
                  <nxt.ptr, head.mod+1>)
D14:         break
           fi
         fi
       fi
     endloop;
D19: free(head.ptr);
D20: return TRUE
```

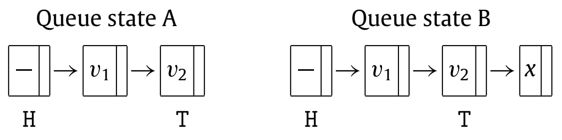
Fig. 2. Michael & Scott's queue.

of the class ConcurrentLinkedQueue in the Java^{TM} Platform, standard edition 6 [19]. $\mathcal{MSQ}$ has a finite set of processes, $procs(\mathcal{MSQ})$, and the operations of $\mathcal{MSQ}$ are enqueue and dequeue, i.e., $OP(\mathcal{MSQ}) = \{\texttt{enq}, \texttt{deq}\}$. $\mathcal{MSQ}$ is a linked-list implementation of a queue, consisting of nodes of type (Val, ptrmc), where Val is the type of the queue elements and ptrmc is the next pointer paired with a modification count. The queue maintains two shared ptrmc variables: Head and Tail. Elements are enqueued at the Tail and dequeued from the Head. Head always points to some dummy element, hence, is never null. Tail points either to the last node in the queue if Tail is up-to-date, or to the second last node if Tail is 'lagging'.

In a CAS-based implementation, one cannot add a new node to the queue and update Tail in a single atomic action. In particular, enq in Fig. 2 requires two updates to Tail in order to complete, namely, the CAS at E9 (which adds a new node onto the queue), and the CAS at E17 (which updates Tail to point to the newly added node). Between executing E9 and E17, the queue is in a 'lagging' state because Tail is not pointing to the real tail of the queue. If Tail is lagging, other processes may 'help' with the progress of the queue by updating Tail. Enqueuers perform helping by updating Tail at E13 while dequeuers do so by updating Tail at D10. Note that it is not possible for Head to lag, and that Tail lags by at most one. An enq must ensure that Tail is accurate (not lagging) before it performs an enqueue, while deq will only 'help' if Tail is lagging and there is exactly one element (i.e., two nodes) in the queue.

For the purposes of proving lock-freedom, analysis is simplified using control-flow graphs. The control flow of $\mathcal{MSQ}$ is presented in Fig. 3. The vertices of the graph is the set $PC(\mathcal{MSQ})$ and the edges correspond to the lines of code. The name of the action at E1 is called *enq*1, and we adopt a similar name for the other actions. If the line of code is a conditional statement it has two edges, annotated with t or f to represent the true and false evaluations of the condition, respectively. For each operation, the labels within *op*.*preLoop*, *op*.*Loop* and *op*.*postLoop* are also identified. It is now easy to see that enq and deq of $\mathcal{MSQ}$ fit the general structure described in Section 2.1. That is, *op*.*preLoop* and *op*.*postLoop* do not contain any potentially infinite loops, while *op*.*Loop* does.

Let us now describe operations enq and deq. Consider first an enq(x) operation executed by process p in isolation. Assume that initially the queue contains two values, v_1 and v_2, and hence is formed from three nodes, as depicted by Queue state A below. For the purposes of description we ignore modification counts.



The first, dummy, node, contains an irrelevant value (—), and points to the node containing value v_1, which in turn points to the node containing v_2. This final node has a next pointer of null. Head points to the dummy node and Tail to the final

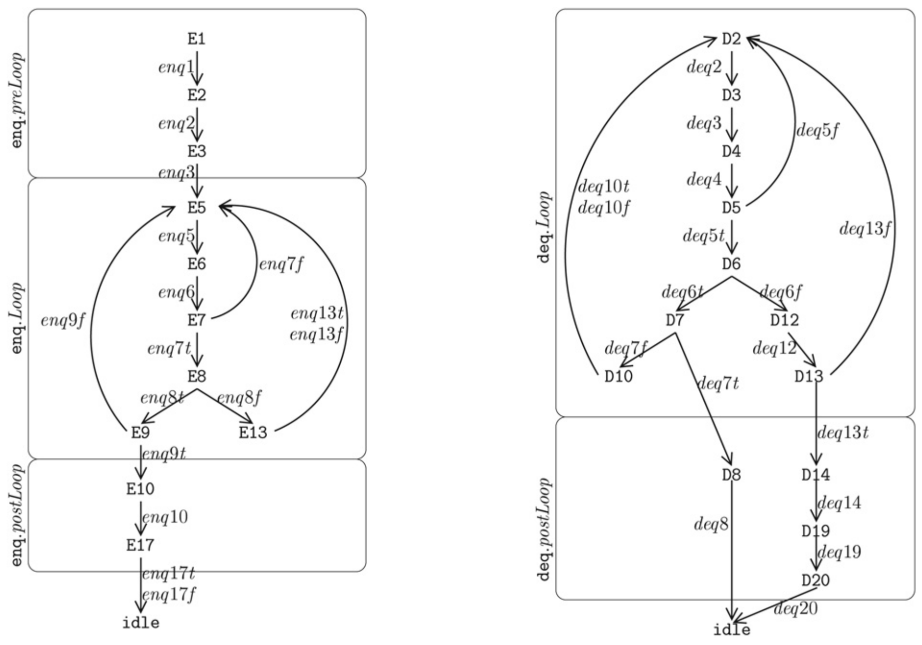


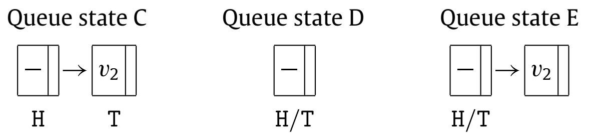
Fig. 3. Control flow of the Michael & Scott queue.

node, represented by H and T in the diagram, respectively. The first three lines of enq (which form enq.*preLoop*) allocate a new node node, set its value field to x, and its next pointer to null (since it will become the last node in the queue). The operation now enters enq.*Loop*. Copies of Tail and Tail.next are taken initially. Since we are assuming no interference, the test at E7 succeeds, and the test at E8 also succeeds, since Tail points to the last element in the queue. The CAS at E9 also succeeds, since tail.next has not changed, and hence the last pointer in the queue is updated to point to the new node. The current state of the queue is depicted in Queue state B above. Note that now Tail does not point to the last node in the list: it is lagging. This is rectified after p exits the loop and executes E17, moving Tail along one link. The effect of the operation is to have added a new node containing value x onto the end of the queue, having modified the next pointer of the previous last element. The shared variable Tail has also been updated to point to the new tail.

Now consider the case where p tries to perform an enqueue operation when the queue is in state B above. Process p will fail the test at E8 since Tail points to the last-but-one element. To deal with this situation, process p attempts to advance Tail itself at E13, before retrying the loop. This is an example of helping, as mentioned above. Note that Tail is lagging exactly when its next pointer is not null.

Now consider the effects of interference on p. If some other process q concurrently executes enq and modifies Tail and/or the next pointer of the last node, then the tests at E7 or E9 may fail, in which case p will retry the loop. If p is at E17 when the interference occurs, this means q has already "helped" p to move Tail along, in which case the CAS fails and p finishes the operation without modifying Tail itself.

Now consider a deq operation performed by process p without interruption, assuming the queue is in state A above. deq.*preLoop* is empty, so the loop is immediately entered. Process p first takes snapshot of Head and Tail, and a copy of the next pointer of Head. Since there is no interference, D5 succeeds, and D6 fails because the queue is non-empty. D12 is therefore executed next, storing the value v_1 in vp. The CAS at D13 succeeds, advancing Head along to its next pointer. The loop is exited and the dequeued node is freed at D19, before the operation completes and returns TRUE at D20. The total effect of the operation is to advance Head along the list, and return the dequeued value, as depicted in state C below. Note the value v_1 is still contained within the new dummy node, although the value is now irrelevant.



Now consider a dequeue on an empty queue, as depicted in Queue state D, by process p. In this case, both Head and Tail point to a dummy node, which has a null next pointer. Process p proceeds as before, but succeeds at the test D6. Because the next pointer is null, D7 also succeeds, and the operation returns FALSE to indicate an empty queue. The more interesting case is when a dequeue is attempted on a queue with exactly one element, v_2, with a lagging Tail, as depicted in Queue state E. The dequeuer must advance the tail pointer before it can dequeue v_2. This case proceeds as above, except that D7 fails. Process p then updates Tail D10, putting the queue into state C, before retrying.

If interference occurs, either through another dequeue or through an enqueue when the queue is empty, the tests at D5, D10 or D13 may fail. In the case of D5 or D13, this forces a retry of the loop, while after D10, the loop is retried regardless.

As argued informally by Michael and Scott [15], the algorithm satisfies the lock-free progress property because the loops are retried only after one of the shared variables has been modified, and if this occurs an infinite number of times then an infinite number of operations must have completed. We will, of course, prove this formally below.

2.3. Well-founded relations

In this section we briefly review well-founded relations, as they are integral to our proofs. A more complete treatment may be found in textbooks such as Dijkstra and Scholten [6] and Gries and Schneider [11].

Definition 1 (*Well-Founded Relation*). Given a set of elements T, and a relation $\prec$ on T, $(\prec, T)$ is a well-founded relation iff every non-empty subset S of T has a minimal element with respect to $\prec$, that is,

$$S \neq \{\} \Leftrightarrow (\exists y: S \bullet (\forall x: S \bullet \neg\ (x \prec y)))$$

An alternative definition is that there are no infinite descending chains of elements of T that are related by $\prec$. That is, if t_i are elements of T, then every chain $\cdots t_a \prec t_b \prec \cdots$ must be finite in length. A well-known example of a well-founded relation is $(<, \mathbb{N})$, where $<$ is “less-than” on numbers. In this paper we make use of the following relation on Boolean values, $(\prec^{\mathbb{B}}, \mathbb{B})$, such that *false* is an improvement on *true*.

$$b1 \prec^{\mathbb{B}} b2 \Leftrightarrow \neg b1 \wedge b2 \tag{2}$$

2.4. Transition systems and trace-based reasoning

In this section we formalise the programs from Fig. 1 and present a logic for reasoning about them.

2.4.1. Programs

We model programs as a type of labelled transition system.

Definition 3 (*Program*). A *program*, $\mathcal{P}$, is defined by:

- $procs(\mathcal{P})$, the finite set of processes of $\mathcal{P}$;
- $PC(\mathcal{P})$, a set of labels for the program counter values;
- $actions(\mathcal{P})$, a set of labels for the possible actions in $\mathcal{P}$;
- $states(\mathcal{P})$, the set of possible states of $\mathcal{P}$, which are mappings from variables to values;
- a non-empty set $start(\mathcal{P}) \subseteq states(\mathcal{P})$, the initial states of $\mathcal{P}$;
- $transitions(\mathcal{P})$: $actions(\mathcal{P}) \to procs(\mathcal{P}) \to states(\mathcal{P}) \nrightarrow states(\mathcal{P})$, the set of possible atomic transitions of $\mathcal{P}$.

The set $procs(\mathcal{P})$ represents the finite set of processes, which in our context are identifiers for distinguishing local state. The set $PC(\mathcal{P})$ contains the pc values that label the statements in the programs, and the set $actions(\mathcal{P})$ represents the atomic actions corresponding to lines of code. For example, $PC(\mathcal{MSQ})$ includes idle, $E1$, $E2$, etc., and $actions(\mathcal{MSQ})$ includes $enq1$, $enq2$, …, etc. Each action α has an initial pc value, $pcpre_\alpha$, and updates the pc of the executing process to some new value, $pcpost_\alpha$.

The variables of a program may either be *shared* or *local*. A local variable of type T is modelled as a function of type $procs(\mathcal{P}) \to T$. For example, $states(\mathcal{MSQ})$ maps shared variable $Head$ to a value of type $node$, and maps local variable $head$ to a value of type $procs(\mathcal{MSQ}) \to node$. The state also includes a program counter pc for each process ($procs(\mathcal{P}) \to PC(\mathcal{P})$).

The start states represent the initial value of the variables. For example, in $start(\mathcal{MSQ})$, $pc_p =$ idle for each process p, and $Head = Tail = Dummy$, where $Dummy$ is some node with a next pointer of null. Although not strictly necessary, we also set $head_p = Head$, $tail_p = Tail$, and $nxt_p =$ null for each process p, to simplify some invariants on the local and shared variables.

We represent the transitions as a function which, given an action, and a process to execute that action, generates a partial function on states. For example, action $enq1$ and process p generate a state transformer, which, assuming $pc_p = E1$ holds, modifies the given state so that $node_p$ is assigned to some new node value and pc_p is updated to $E2$. For presentation purposes, we define *transitions* using guard/effect pairs, which give the domain and updates of the state transformer, respectively. That is, given an action α and process p, $transitions(\alpha)(p)$ is specified as follows:

$$\begin{array}{ll} \alpha_p: & \textbf{grd}: pc_p = pcpre_\alpha \wedge guards \\ & \textbf{eff}: \ updates; pc_p := pcpost_\alpha \end{array} \tag{4}$$

The guard (denoted **grd**) is a predicate that defines the states in which the transition is enabled, and the effect (denoted **eff**) assigns new values to the program variables. Predicate *guards* does not refer to pc_p and *updates* does not assign to pc_p. We note that *guards* may be equivalent to true, in which case α_p only relies on the value of pc_p, and *updates* may be empty, in which case the only effect of α_p is to update pc_p. For example, the statement specification for *enq*5 executed by process p is represented by the following:

$$\begin{array}{ll} enq5_p: & \textbf{grd}: pc_p = E5 \\ & \textbf{eff}: \; tail_p := Tail; pc_p := E6 \end{array}$$

This statement specification is enabled in any state in which process p is ready to execute the code at $E5$, and the effect of this transition is to update local variable $tail_p$ to the value held by the shared variable $Tail$ and to advance pc_p to $E6$. For this action, $pcpre_{enq5}/pcpost_{enq5} = E5/E6$.

Conditional statements are represented by two transition specifications, appended with t and f, corresponding to the true and false evaluation of the guards, respectively. A CAS is also encoded using two transition specifications; for instance, the CAS at $E13$ is represented by

$$\begin{array}{ll} enq13t_p: & \textbf{grd}: pc_p = E13 \wedge Tail = tail_p \\ & \textbf{eff}: \; Tail := \langle nxt_p.ptr, tail_p.mod + 1\rangle; pc_p := E5 \end{array}$$

$$\begin{array}{ll} enq13f_p: & \textbf{grd}: pc_p = E13 \wedge Tail \neq tail_p \\ & \textbf{eff}: \; pc_p := E5 \end{array}$$

2.4.2. Transitions

An element of $transitions(\mathcal{P})$ is called a *transition*, and is a four-tuple. For a *transition* $t = (\alpha, p, s, s')$, we write $s \xrightarrow{\alpha_p, \mathcal{P}} s'$ as a shorthand for $t \in transitions(\mathcal{P})$ (omitting $\mathcal{P}$ when it is clear from the context). For all transitions $(\alpha, p, s, s') \in transitions(\mathcal{P})$, s must satisfy the **grd** of α_p, and s' is obtained by updating s according to the **eff** of α_p. We write $t.act$ for α, $t.proc$ for p, $t.pre$ for s, and $t.post$ for s'.

We write $t.F$ to indicate property F is satisfied by transition t. If F is a typical predicate on the state, $t.F$ holds if F holds in $t.pre$. In the context of lock-freedom, we are concerned with the actions of a program that are executed, and therefore introduce the property $\mathtt{exec}(A)$, such that $t.\mathtt{exec}(A)$ holds if transition t is an execution of an action in the set of actions A.

$$t.\mathtt{exec}(A) \mathrel{\widehat{=}} t.act \in A$$

2.4.3. Traces

The transitions of program $\mathcal{P}$ generate a set of traces, which represent all possible executions of $\mathcal{P}$. A trace of $\mathcal{P}$ is a sequence of transitions, which begins in $start(\mathcal{P})$, with each succeeding transition's pre-state matching the preceding transition's post-state.

Definition 5 (*Traces*). The *traces* of a program $\mathcal{P}$ is given by the set $traces(\mathcal{P})$ where

$$traces(\mathcal{P}) \mathrel{\widehat{=}} \{tr: \mathbb{N} \to transitions(\mathcal{P}) | tr_0.pre \in start(\mathcal{P}) \wedge (\forall i: \mathbb{N} \bullet tr_i.post = tr_{i+1}.pre)\}$$

Note that, as mentioned earlier, the traces of all programs of the form in Fig. 1 are infinite in length. We assume the presence of *minimal progress* [16], where *some* enabled process is chosen for execution, although this may always be the same process.

Traces are more commonly represented as sequences of states, however for the purposes of this paper it is advantageous to reason directly about the actions that have been executed. Sequences of transitions can be be mapped to sequences of states by taking the pre-state of every transition.

2.4.4. Temporal logic

To describe properties of traces we extend the set of formulas with the temporal operators *always* ($\Box$), *eventually* ($\Diamond$), and *next* ($\bigcirc$). A formula $\Box F$ indicates that F holds in every future state of the system, while $\Diamond F$ indicates that F holds in some future state. A formula $\Box\Diamond F$ therefore indicates that there is always some future state in which F holds; for an infinite-length trace, this means that F occurs infinitely often. A formula $\bigcirc F$ indicates that F holds in the immediate next state.

We write $(tr, i) \vdash F$ if F is satisfied by the transition at position i in tr. The meaning of the temporal operators follows those of Manna and Pnueli [13].

$$\begin{array}{lll} (tr, i) \vdash \Box F & \Leftrightarrow & (\forall j: \mathbb{N} \bullet j \geq i \Rightarrow (tr, j) \vdash F) \\ (tr, i) \vdash \Diamond F & \Leftrightarrow & (\exists j: \mathbb{N} \bullet j \geq i \wedge (tr, j) \vdash F) \\ (tr, i) \vdash \bigcirc F & \Leftrightarrow & (tr, i+1) \vdash F \end{array}$$

A trace tr satisfies temporal formula F, written $tr \vdash F$, if $(tr, 0) \vdash F$, i.e., F holds in the initial state. For a program $\mathcal{P}$ and a temporal formula F, $\mathcal{P}$ satisfies F, written $\mathcal{P} \models F$, iff $(\forall tr: traces(\mathcal{P}) \bullet tr \vdash F)$.

Below are some properties of infinite traces from temporal logic, which we use for reasoning about formulas of the form $\Box\Diamond F$.

$$\Box\Diamond(A \wedge B) \Rightarrow \Box\Diamond A \wedge \Box\Diamond B \tag{6}$$

$$\Box\Diamond\left(\bigvee_{i:I} A_i\right) \Leftrightarrow \bigvee_{i:I}(\Box\Diamond A_i) \quad \text{where } I \text{ is finite} \tag{7}$$

$$\neg\,\Box\Diamond A \Leftrightarrow \Diamond\Box\neg\, A \tag{8}$$

$$\Box\Diamond\bigcirc F \Leftrightarrow \Box\Diamond F \tag{9}$$

$$\Box\Diamond(F \wedge \bigcirc\neg F) \Leftrightarrow \Box\Diamond(\neg F \wedge \bigcirc F) \tag{10}$$

Properties (6)–(8) have been proved by Manna and Pnueli [13]. Property (9) states that there is always some future state in which F holds iff there is always some future state where F holds in the next state. The proof of this property in the $\Rightarrow$ direction is trivial from definition. In the $\Leftarrow$ direction, for every non-zero index i such that $(tr, i) \vdash F$, then $tr_{i-1} \vdash \bigcirc F$. Because there are infinite number of such indices i, there are an infinite number of indices such that the next state satisfies F.

Property (10) states that if there is always a transition from a state in which F holds to one in which $\neg F$ holds, then there must always be a converse transition from $\neg F$ to F. We prove (10) in the $\Rightarrow$ direction; the $\Leftarrow$ direction is symmetric. By (6) and (9), $\Box\Diamond F$ and $\Box\Diamond\neg F$. Hence, from any state where $\neg F$ holds there is a future state where F holds. Since $F \vee \neg F$ holds in every state, there must be consecutive states in which $\neg F$ and F hold.

To prove properties of the form $\Box\Diamond F$ we identify a well-founded relation on program states and show that each transition in the system either reduces the value of the state according to the well-founded relation or establishes F. Using a well-founded relation ensures a finite bound on the number of intermediate states before F is established. A similar approach is used for developing methods for proving *leads-to* properties by Chandry and Misra [1] and *until* properties by Fix and Grumberg [9].

Theorem 11. *Let $\mathcal{P}$ be a program from Definition 3; let $(\prec, states(\mathcal{P}))$ be a well-founded relation; and let F be some property of transitions. If*

$$(\forall t\colon transitions(\mathcal{P}) \bullet (t.post \prec t.pre) \vee t.F) \tag{12}$$

then

$$\mathcal{P} \models \Box\Diamond F \tag{13}$$

Proof. We choose an arbitrary trace tr of $\mathcal{P}$ such that (12) holds. From the definition of traces, for all $i \in \mathbb{N}$ we have $tr_{i+1}.pre \prec tr_i.pre$ or $tr_i.F$. We now prove by contradiction: we show that $tr \vdash \neg\,\Box\Diamond F$ is false. By (8), this is equivalent to showing $tr \vdash \Diamond\Box\neg F$, that is, there is some index i in tr such that $\neg F$ holds for all tr_j where $j \geq i$. Thus, by (12) for all $j \geq i$, $tr_{j+1}.pre \prec tr_j.pre$ must hold. However, this results in an infinite descending chain of $states(\mathcal{P})$, which, because $(\prec, states(\mathcal{P}))$ is well-founded, is impossible. $\Box$

2.5. Lock-freedom

Lock-freedom is a system-wide property which guarantees that eventually *some* process will complete an operation. That is, in Fig. 1, some processes may execute *op.Loop* forever, as long as there is always at least one other process which is successfully completing its operations. A survey and formal definition of lock-freedom are given by Dongol [8]. However, because the definition is designed to be as general as possible, it is not easily amenable to proof. Our definition of lock-freedom is based on Dongol's, but specialised for the class of programs described in Section 2.1. We first define the set of completing actions of a program $\mathcal{P}$ as those that return a process to the idle state.

$$complete_{\mathcal{P}} \mathrel{\widehat{=}} \{\alpha\colon actions(\mathcal{P}) | pcpost_\alpha = \mathsf{idle}\} \tag{14}$$

Definition 15 (*Lock-Freedom*). Program $\mathcal{P}$ is *lock-free* iff it satisfies the following property

$$\mathcal{P} \models \Box\Diamond\,\mathsf{exec}(complete_{\mathcal{P}}) \tag{16}$$

That is, a program $\mathcal{P}$ is lock-free iff from any state, there is a point in the future where some process completes its operation. To prove lock-freedom, we instantiate formula F in Theorem 11 with $\mathsf{exec}(complete_{\mathcal{P}})$. This reduces a proof of lock-freedom to the construction of a well-founded relation followed by case analysis on all actions in the program.

3. The proof strategy

In this section we describe our strategy for proving lock-freedom, which is based on the observation that retries of *op.Loop* by process *p* are acceptable as long as they are only caused by interference from other processes which indicates, directly or indirectly, that some other process has completed its operation.

Definition 15 (lock-freedom) requires some process to execute an action in $complete_{\mathcal{P}}$. However, rather than proving this condition directly, we identify a more general set of *progress* actions, $progress_{\mathcal{P}}$, that satisfy the property that if always eventually a progress action is executed, then always eventually a completing action is executed. The bulk of the lock-freedom proof then consists of showing the weaker requirement that always eventually a progress action will be executed. The intuition behind the proof technique is that if progress occurs (i.e., an action in $progress_{\mathcal{P}}$ is executed) after some process *p* starts *op.Loop*, then *p* may retry *op.Loop*. However, if progress does not occur after *p* starts *op.Loop*, then *p* must proceed to making progress itself by executing an action in $progress_{\mathcal{P}}$. In summary, the proof strategy consists of the following steps:

(i) (a) Identify the set of actions which indicate progress of program $\mathcal{P}$ as a whole, $progress_{\mathcal{P}}$.
  (b) Prove that if always eventually an action in $progress_{\mathcal{P}}$ is executed, then always eventually an action in $complete_{\mathcal{P}}$ is executed.

(ii) Augment the program with an auxiliary progress-detection variable π of type $procs(\mathcal{P}) \to \mathbb{B}$, such that $\pi_p = true$ if some process has executed a progress action.
  (a) All actions in $progress_{\mathcal{P}}$ must set π to $(\lambda p{:}\, procs(\mathcal{P}) \bullet true)$ to indicate that progress has been made.
  (b) For each *op*, the action at the start of *op.Loop* must set π_p to *false* to indicate that progress has not been made since *p* (re)started *op.Loop*.

(iii) Define a well-founded relation on $states(\mathcal{P})$.
  (a) Using the control-flow graphs as a guide, instantiate two well-founded relations, $(\prec^{\times}, PC(\mathcal{P}))$ and $(\prec^{\checkmark}, PC(\mathcal{P}))$. These give the sequences of *pc* values a process *p* will have in the cases $\neg\pi_p$ and π_p, respectively. The two relations are combined to define a generic relation $(\prec, states(\mathcal{P}))$.
  (b) Prove that $(\prec^{\times}, PC(\mathcal{P}))$ and $(\prec^{\checkmark}, PC(\mathcal{P}))$ are well-founded.

(iv) By case analysis, show that each action not in $progress_{\mathcal{P}}$ reduces the value of the state according to $(\prec, states(\mathcal{P}))$.

We explain these steps in more detail below, using $\mathcal{MSQ}$ as a running example. We note that insight is only required in steps (i) a, (i) b and (iii) a. Once step (i) is complete, step (ii) is a purely mechanical procedure. Furthermore, after instantiating $(\prec^{\times}, PC(\mathcal{P}))$ and $(\prec^{\checkmark}, PC(\mathcal{P}))$, one must merely plug them into a template for the relation $(\prec, states(\mathcal{P}))$. Steps (iii) b and (iv) can be proved using a theorem proving tool with minimal user interaction.

3.1. Progress actions

In order to prove lock-freedom (16), we are required to determine the set of actions corresponding to completing an operation, $complete_{\mathcal{P}}$. By (14), we have:

$$complete_{\mathcal{MSQ}} = \{enq17t, enq17f, deq8, deq20\}$$

That is, the true and false cases of *enq*17 are the last actions in an enqueue, and *deq*8 and *deq*20 are the last actions in the empty and non-empty dequeue cases, respectively.

However, proving lock-freedom by showing (16) holds directly is not straightforward because a process may be continually interfered with while in *op.Loop*, which prevents the process from executing a $complete_{\mathcal{P}}$ action. Furthermore, the $complete_{\mathcal{P}}$ actions usually do not correspond to actions that cause interference (and hence cause other processes to retry *op.Loop*). Typically it is processes that exit *op.Loop* (and enter *op.postLoop*) that cause interference, hence, we define a second set of actions $exit_{\mathcal{P}}$ below. The set *pcPostLoop* contains $pcpre_{\alpha}$ for each action α in *op.postLoop*.

$$exit_{\mathcal{P}} \mathrel{\widehat{=}} \{\alpha{:}\, actions(\mathcal{P}) | pcpre_{\alpha} \notin pcPostLoop_{\mathcal{P}} \wedge pcpost_{\alpha} \in pcPostLoop_{\mathcal{P}}\} \tag{17}$$

The set $exit_{\mathcal{P}}$ is therefore formed from the set of actions that "cross the border" between *op.Loop* and *op.postLoop* in the control-flow graph. For $\mathcal{MSQ}$ we have:

$$exit_{\mathcal{MSQ}} = \{enq9t, deq7t, deq13t\}$$

We require that the following property holds to guarantee that an execution of an $exit_{\mathcal{P}}$ action leads to an execution of a $complete_{\mathcal{P}}$ action:

$$\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(exit_{\mathcal{P}}) \Rightarrow \Box\Diamond\, \mathsf{exec}(complete_{\mathcal{P}}) \tag{18}$$

That is, if there are an infinite number of executions of actions in $exit_{\mathcal{P}}$, then there must be an infinite number of executions of an action in $complete_{\mathcal{P}}$.

Theorem 19. *Property (18) holds for any program of the form in Fig. 1.*

Proof. We note that from the definition of *op* in Fig. 1, a transition from inside *op.postLoop* to outside *op.postLoop* must be a transition to idle. This is given by the following property.

$$(\exists p\colon procs(\mathcal{P}) \bullet pc_p \in pcPostLoop_{\mathcal{P}} \wedge \bigcirc pc_p \notin pcPostLoop_{\mathcal{P}}) \Rightarrow \mathsf{exec}(complete_{\mathcal{P}}) \tag{20}$$

$$\begin{array}{ll}
& \Box\Diamond\, \mathsf{exec}(exit_{\mathcal{P}}) \\
\Rightarrow & \{(17)\} \\
& \Box\Diamond\,(\exists p\colon procs(\mathcal{P}) \bullet pc_p \notin pcPostLoop_{\mathcal{P}} \wedge \bigcirc pc_p \in pcPostLoop_{\mathcal{P}}) \\
\Leftrightarrow & \{\text{By } (7),\ procs(\mathcal{P}) \text{ is finite}\} \\
& (\exists p\colon procs(\mathcal{P}) \bullet \Box\Diamond\,(pc_p \notin pcPostLoop_{\mathcal{P}} \wedge \bigcirc pc_p \in pcPostLoop_{\mathcal{P}})) \\
\Leftrightarrow & \{\text{By } (10)\} \\
& (\exists p\colon procs(\mathcal{P}) \bullet \Box\Diamond\,(pc_p \in pcPostLoop_{\mathcal{P}} \wedge \bigcirc pc_p \notin pcPostLoop_{\mathcal{P}})) \\
\Leftrightarrow & \{\text{By } (7),\ procs(\mathcal{P}) \text{ is finite}\} \\
& \Box\Diamond\,(\exists p\colon procs(\mathcal{P}) \bullet pc_p \in pcPostLoop_{\mathcal{P}} \wedge \bigcirc pc_p \notin pcPostLoop_{\mathcal{P}}) \\
\Rightarrow & \{(20)\} \\
& \Box\Diamond\, \mathsf{exec}(complete_{\mathcal{P}}) \quad \Box
\end{array}$$

Many operations require updates to more than one shared variable. Because each CAS can only update a single variable at a time, these updates must be spread across a number of CAS statements. For instance, in $\mathcal{MSQ}$, `enq` updates the shared pointer `tail.ptr->next` at E9, then updates `Tail` at E17. The effect of operation `enq` is 'felt' after the first CAS (in the sense of linearisability [7]), after which the state of the program is 'lagging'. Lock-free algorithms are designed so that from a lagging state, any process may 'help' bring the state of the program up-to-date. However, because helping actions may cause interference with other processes, it is necessary to define a third set of actions, $help_{\mathcal{P}}$. Unlike $complete_{\mathcal{P}}$ and $exit_{\mathcal{P}}$, which could be determined by static analysis of the control flow, determining the set $help_{\mathcal{P}}$ requires some knowledge of the algorithm at hand. Typically, helping actions modify some shared variables to facilitate the execution of the $exit_{\mathcal{P}}$ actions. Formally, $help_{\mathcal{P}}$ must satisfy the following property:

$$\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(help_{\mathcal{P}}) \Rightarrow \Box\Diamond\, \mathsf{exec}(exit_{\mathcal{P}}) \tag{21}$$

That is, the shared data structure can only be "helped" a finite number of times before some process exits *op.Loop*.

For $\mathcal{MSQ}$ we choose $help_{\mathcal{MSQ}}$ as the the set of actions which *successfully* update a lagging `Tail`.

$$help_{\mathcal{MSQ}} = \{enq13t, enq17t, deq10t\}$$

Let us define *TL* as the property that *Tail* is lagging (its next pointer is not *null*), that is, $TL \mathrel{\widehat{=}} (Tail.ptr \rightarrow next).ptr \neq null$. As shown by Doherty et al., [7], the set $help_{\mathcal{MSQ}}$ is exactly the set of actions which update the state so that *TL* holds initially and $\neg TL$ holds afterwards. The enqueue action *enq*9*t* is the converse action: $\neg TL$ must hold initially but afterwards *TL* holds. This is given formally by the properties below.

$$\mathsf{exec}(help_{\mathcal{MSQ}}) \Leftrightarrow TL \wedge \bigcirc\neg TL \tag{22}$$

$$\mathsf{exec}(\{enq9t\}) \Leftrightarrow \neg TL \wedge \bigcirc TL \tag{23}$$

Theorem 24. *Property* (21) *holds for* $\mathcal{MSQ}$.

Proof.

$$\begin{array}{ll}
& \Box\Diamond\, \mathsf{exec}(help_{\mathcal{MSQ}}) \\
\Leftrightarrow & \{\text{by } (22)\} \\
& \Box\Diamond\,(TL \wedge \bigcirc\neg TL) \\
\Leftrightarrow & \{\text{from } (10)\} \\
& \Box\Diamond\,(\neg TL \wedge \bigcirc TL) \\
\Leftrightarrow & \{\text{by } (23)\} \\
& \Box\Diamond\, \mathsf{exec}(\{enq9t\}) \\
\Rightarrow & \{\text{by definition}\} \\
& \Box\Diamond\, \mathsf{exec}(exit_{\mathcal{MSQ}}) \quad \Box
\end{array}$$

Finally, we define the set of progress actions as the union of the three sets.

$$progress_{\mathcal{P}} = complete_{\mathcal{P}} \cup exit_{\mathcal{P}} \cup help_{\mathcal{P}} \tag{25}$$

Theorem 26. *For a program* $\mathcal{P}$ *and sets* $complete_{\mathcal{P}}$, $exit_{\mathcal{P}}$ *and* $help_{\mathcal{P}}$ *that satisfy* (18) *and* (21), *if*

$$\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(progress_{\mathcal{P}}) \tag{27}$$

then $\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(complete_{\mathcal{P}})$, *and hence* $\mathcal{P}$ *is lock-free.*

Proof. $\Box\Diamond\ \mathsf{exec}(progress_{\mathcal{P}})$

$\Leftrightarrow$ {from (7) (twice), $progress_{\mathcal{P}}$ is finite}

$\Box\Diamond\ \mathsf{exec}(help_{\mathcal{P}}) \vee \Box\Diamond\ \mathsf{exec}(exit_{\mathcal{P}}) \vee \Box\Diamond\ \mathsf{exec}(complete_{\mathcal{P}})$

$\Leftrightarrow$ {(21), (18), logic}

$\Box\Diamond\ \mathsf{exec}(complete_{\mathcal{P}})$ $\Box$

3.2. Augment with progress detection

Having identified the progress transitions in the code, we augment the program with an auxiliary *progress-detection* variable, π, of type $procs(\mathcal{P}) \to \mathbb{B}$, which is initially *true* for all processes. For each process p, the value of π_p is set to false at the start of each iteration of *op.Loop*. Setting π to $(\lambda p{:}\, procs(\mathcal{P}) \bullet true)$ as part of the effect of each action in $progress_{\mathcal{P}}$ records the fact that progress has been made. As long as $\neg\pi_p$ holds, no $progress_{\mathcal{P}}$ actions have been executed, and hence, snapshots of the shared variables modified only by $progress_{\mathcal{P}}$ actions remain accurate. Within the loop, if $\neg\pi_p$ holds, then p must proceed to making progress by executing an action in $progress_{\mathcal{P}}$, whereas if π_p holds, p is allowed to retry *op.Loop* (to refresh its snapshot of the global variables). Note that π_p does not necessarily mean that p has been interfered with, hence, even if π_p holds, p may execute an action in $progress_{\mathcal{P}}$.

For $\mathcal{MSQ}$, we set π to false for process p at $enq5$ and $deq2$, the start of `enq`.*Loop* and `deq`.*Loop*, respectively, and set π to true for all processes q when an action from $progress_{\mathcal{MSQ}}$ is executed. For instance, the augmented transition specifications for $enq5$ and $enq13t$ are:

$$\begin{array}{ll} enq5_p{:} & \textbf{grd}{:}\ pc_p = E5 \\ & \textbf{eff}{:}\ \ tail_p := Tail;\ \pi_p := false;\ pc_p := E6 \end{array}$$

$$\begin{array}{ll} enq13t_p{:} & \textbf{grd}{:}\ pc_p = E13 \wedge Tail = tail_p \\ & \textbf{eff}{:}\ \ Tail := \langle nxt_p.ptr,\ tail_p.ref + 1\rangle; \\ & \qquad \pi := (\lambda p{:}\, procs(\mathcal{P}) \bullet true); \\ & \qquad pc_p := E5 \end{array}$$

After augmenting $\mathcal{MSQ}$ with π, we obtain the following invariants that relate π to the local snapshots of the shared variables.

$$(\forall p{:}\, procs(\mathcal{MSQ}) \bullet \neg\pi_p \Rightarrow$$

$$(pc_p \neq D3 \Rightarrow tail_p = Tail)\ \wedge \tag{28}$$

$$(pc_p \in PC(\mathtt{deq}) \Rightarrow head_p = Head)\ \wedge \tag{29}$$

$$(pc_p \in PC(\mathtt{enq}) \setminus \{E6\} \Rightarrow nxt_p = tail_p.ptr \to next)) \tag{30}$$

For any process p, given that progress has not been made since p executed $enq5_p$ or $deq2_p$, invariant (28) states that the values of $tail_p$ and $Tail$ are equal if p is not about to execute $deq3_p$. Invariant (28) holds trivially because the actions that modify $Tail$ ($enq13t$, $enq17t$ and $deq10t$) are in $progress_{\mathcal{MSQ}}$ and hence set π_p to true for all p. Similarly, invariant (29) states that any process performing a dequeue can be certain that no process has modified $Head$ while $\neg\pi_p$ holds, which follows because all actions that modify $Head$ are in $progress_{\mathcal{MSQ}}$. Invariant (30) is similar, stating that if $\neg\pi_p$ and p is executing `enq`.*Loop*, after refreshing nxt_p at $E6$, the *next* pointer of $tail_p$ has not changed since it was read. There are two actions that modify *next* pointers: $enq9t$ and $enq3$. The former case holds trivially because $enq9t$ also sets π_p to true (which falsifies the antecedent). The latter case holds because a newly allocated node at $E3$ cannot appear as a node currently within the queue [7].

3.3. The well-founded relation

In this section, we give a generic definition of the relation $(\prec, states(\mathcal{P}))$ required for Theorem 11, and describe a technique for constructing the relation. Because lock-freedom is a property of the program as a whole, the relation $\prec$ is defined over $states(\mathcal{P})$. We construct $(\prec, states(\mathcal{P}))$ so that it is sufficient to consider the execution of a single process in order to determine lock-freedom. Interleavings of two or more processes do not need to be considered, which keeps the overall technique scalable.

The control-flow graph of each operation in $\mathcal{P}$ is a directed graph. Treating each edge $\langle pcpre, pcpost\rangle$ to mean $pcpost < pcpre$, we may obtain a relation on pc values from the control-flow graph. Due to *op.Loop*, however, the relation will contain cycles, which means it cannot be well-founded. However, by removing particular edges from the graph, we can obtain a well-founded relation. For example, if we remove edges $\langle E5, E6\rangle$ and $\langle D2, D3\rangle$ from the graph in Fig. 3, then the graphs for `enq` and `deq` both represent well-founded relations (directed acyclic graphs) since the loops have been cut. Similarly, if we remove all edges of the form $\langle pcpre, E5\rangle$ and $\langle pcpre, D2\rangle$, i.e., all "upward pointing" edges, we have another well-founded relation. We note that $\langle E5, E6\rangle$ and $\langle D2, D3\rangle$ correspond to actions that falsify π_p, and that the "upward pointing" edges correspond to actions that are only taken if interference has occurred.

3.3.1. *Relation on processes*

The relevant information in the state of a process p is whether or not a $progress_{\mathcal{P}}$ action has been executed, and the value of pc_p. Thus, we define type *ProcInfo* as a pair formed from a Boolean and a program counter value, i.e., $ProcInfo \mathrel{\widehat{=}} \mathbb{B} \times PC(\mathcal{P})$, and define a function δ that maps each process to its corresponding *ProcInfo* value, i.e.

$$\delta\colon states(\mathcal{P}) \to procs(\mathcal{P}) \to ProcInfo$$
$$\delta(s)(p) = (s.\pi_p, s.pc_p)$$

For convenience, we will write $\delta_p(s)$ for $\delta(s)(p)$.

We first motivate a relation $(\lll, ProcInfo)$, which is used to compare successive states of an individual process p in a trace. If s and s' are successive states and $\delta_p(s') \lll \delta_p(s)$ for some p, then p has improved its position with respect to making progress. To record whether or not process p is on track to executing an action in $progress_{\mathcal{P}}$, the relation $(\lll, ProcInfo)$ must take pc_p into account. Furthermore, because a process has two distinct goals depending on the value of π_p, we use two different relations on pc_p.

- If $\neg\pi_p$ continues to hold, process p must eventually execute an action in $progress_{\mathcal{P}}$. The well-founded relation on pc_p for this case is $(\prec^{\times}, PC(\mathcal{P}))$, which includes all edges in the control graph except those that fail due to interference.
- If π_p, then some process must have executed an action in $progress_{\mathcal{P}}$ since p started *op*.*Loop*, hence, p is allowed to retry *op*.*Loop* (although it need not do so). The well-founded relation on pc_p for this case is $(\prec^{\checkmark}, PC(\mathcal{P}))$, and includes all actions except the first action in *op*.*Loop*, i.e., the actions that falsify π_p.

Definition 31 (*Relation on ProcInfo*). For any $(b', pc'), (b, pc) \in ProcInfo$

$$\begin{aligned}(b', pc') \lll (b, pc) \Leftrightarrow\ & b' \prec^{\mathbb{B}} b \vee \\ & (b' = b \wedge \\ & \quad ((\neg b \wedge pc' \prec^{\times} pc) \vee (b \wedge pc' \prec^{\checkmark} pc)))\end{aligned} \tag{32}$$

Thus $\delta_p(s') \lll \delta_p(s)$ holds if either
- $s'.b_p \prec^{\mathbb{B}} s.b_p$ holds, i.e., π_p is falsified (recall (2)); or
- $s'.b_p = s.b_p$ holds, i.e., π_p is unchanged and
  . if $\neg\pi_p$ holds, p advances with respect to $(\prec^{\times}, PC(\mathcal{P}))$
  . if π_p holds, p advances with respect to $(\prec^{\checkmark}, PC(\mathcal{P}))$.

Clearly, when π is set to $(\lambda p\colon procs(\mathcal{P}) \bullet true)$, then all processes are in a worse position with respect to $(\prec^{\mathbb{B}}, \mathbb{B})$, and hence with respect to $(\lll, ProcInfo)$. However, this is not a problem, because π is only set to $(\lambda p\colon procs(\mathcal{P}) \bullet true)$ by actions in $progress_{\mathcal{P}}$ and execution of an action in $progress_{\mathcal{P}}$ indicates progress of the program. We must ensure that the relation $(\prec^{\times}, PC(\mathcal{P}))$ represents the sequence of pc values taken by a process in the lead-up to executing an action in $progress_{\mathcal{P}}$, and $(\prec^{\checkmark}, PC(\mathcal{P}))$ represents the sequence of pc values taken by a process that has possibly been interfered with (i.e., ends in an action that falsifies π_p). The link between the two relations is the action that falsifies π_p; it is the base of $(\prec^{\checkmark}, PC(\mathcal{P}))$ and executing it results in an improvement according to $(\lll, ProcInfo)$.

Theorem 33. *Assuming* $(\prec^{\mathbb{B}}, \mathbb{B})$, $(\prec^{\times}, PC(\mathcal{P}))$ *and* $(\prec^{\checkmark}, PC(\mathcal{P}))$ *are well-founded,* $(\lll, ProcInfo)$ *as defined by* (32) *is well-founded.*

Proof. We prove that every non-empty set of *ProcInfo* has a minimal element (Definition 1). Choose S to be an arbitrary non-empty set of *ProcInfo*. Define *FS* as the subset of S containing all elements with the first element being *false*, and *TS* the subset with all first elements being *true*, that is

$$FS \mathrel{\widehat{=}} \{(b, pc)\colon S \mid \neg b\} \qquad TS \mathrel{\widehat{=}} \{(b, pc)\colon S \mid b\}$$

Sets *FS* and *TS* are mutually exclusive and exhaustive of S and furthermore, by the definition of $(\lll, ProcInfo)$,

$$(\forall fs\colon FS \bullet (\forall ts\colon TS \bullet fs \lll ts)). \tag{34}$$

Now select the set of pc values in *FS* and *TS*.

$$pcFS \mathrel{\widehat{=}} \{pc\colon PC(\mathcal{P}) \mid (false, pc) \in FS\} \qquad pcTS \mathrel{\widehat{=}} \{pc\colon PC(\mathcal{P}) \mid (true, pc) \in TS\}$$

If *pcFS* is non-empty, because $(\prec^{\times}, PC(\mathcal{P}))$ is well-founded, there exists a minimal element, say $pcFSmin \in pcFS$. Similarly, if *pcTS* is non-empty, there exists a minimal element $pcTSmin \in pcTS$. We now perform case analysis on whether or not *FS* is empty. If *FS* is non-empty, (*false*, *pcFSmin*) is in *FS*, which is the minimal element of *FS*. By (34) *pcFSmin* is smaller than every element of *TS*, hence, (*false*, *pcFSmin*) is a minimal element of S. If *FS* is empty, there are no elements in S with first parameter *false*, hence $TS = S$. By our assumptions S is non-empty, hence (*true*, *pcTSmin*) is in *TS*, which by definition is the minimal element of *TS*, and thus S. □

Importantly, execution of an action not in $progress_{\mathcal{P}}$ does not affect π_p or pc_p of any other process, hence, we obtain the following property which follows from the definition of δ.

$$s \xrightarrow{\alpha_p} s' \wedge \alpha \notin progress_{\mathcal{P}} \Rightarrow (\forall q\colon procs(\mathcal{P}) \bullet p \neq q \Rightarrow \delta_q(s') = \delta_q(s)) \tag{35}$$

We use this property later to simplify some proofs.

3.3.2. Relation on states

We now define the well-founded relation $(\prec, states(\mathcal{P}))$.

Definition 36 (*Relation on Program States*). For any $s, s' \in states(\mathcal{P})$,

$$s' \prec s \Leftrightarrow (\exists p : procs(\mathcal{P}) \bullet \delta_p(s') \lll \delta_p(s) \wedge (\forall q : procs(\mathcal{P}) \bullet p \neq q \Rightarrow \delta_q(s') = \delta_q(s))) \tag{37}$$

That is, s' is an improvement on s with respect to $(\prec, states(\mathcal{P}))$ if exactly one process is closer to executing a $progress_{\mathcal{P}}$ action and no other processes have changed.[1]

Theorem 38. *Assuming $(\lll, ProcInfo)$ is well-founded, the relation $(\prec, states(\mathcal{P}))$ as defined in (37) is well-founded.*

Proof. We show that any descending chain of $states(\mathcal{P})$ must be finite. Pick an arbitrary chain c. For any pair of consecutive states c_i, c_{i+1}, exactly one process reduces its value within $(\lll, ProcInfo)$, and all other processes remain unchanged. Because there are only a finite number of processes, and because only a finite number of steps are possible for each process within $(\lll, ProcInfo)$, the chain c must be finite. □

We have given a generic well-founded relation on states of programs $\mathcal{P}$. However, completing the definition of Definition 36 relies on the instantiation of the relations $(\prec^{\times}, PC(\mathcal{P}))$ and $(\prec^{\checkmark}, PC(\mathcal{P}))$, which are determined by the actions corresponding to the "no-interference" and "interference" execution sequences of each operation, respectively. For program $\mathcal{P}$, let us call these actions $\mathcal{P}_{\times}$ and $\mathcal{P}_{\checkmark}$. Then the relations are defined below.

$$pc' \prec^{\times} pc \;\Leftrightarrow (\exists \alpha : \mathcal{P}_{\times} \bullet (pc', pc) = (pcpost_{\alpha}, pcpre_{\alpha})) \tag{39}$$

$$pc' \prec^{\checkmark} pc \;\Leftrightarrow (\exists \alpha : \mathcal{P}_{\checkmark} \bullet (pc', pc) = (pcpost_{\alpha}, pcpre_{\alpha})) \tag{40}$$

From this we may state that improving with respect to $(\prec^{\times}, PC(\mathcal{P}))$ is equivalent to executing an action in $\mathcal{P}_{\times}$ (and similarly for $(\prec^{\checkmark}, PC(\mathcal{P}))$).

$$s \xrightarrow{\alpha_p} s' \Rightarrow (s'.pc_p \prec^{\times} s.pc_p \Leftrightarrow \alpha \in \mathcal{P}_{\times}) \tag{41}$$

$$s \xrightarrow{\alpha_p} s' \Rightarrow (s'.pc_p \prec^{\checkmark} s.pc_p \Leftrightarrow \alpha \in \mathcal{P}_{\checkmark}) \tag{42}$$

For any lock-free program we require the sets $\mathcal{P}_{\times}$ and $\mathcal{P}_{\checkmark}$ to be selected so that the corresponding relations are well-founded. If the control-flow graphs corresponding to $\mathcal{P}_{\times}$ and $\mathcal{P}_{\checkmark}$ are acyclic then well-foundedness is trivial.

We now return to our example and instantiate the relations $(\prec^{\times}, PC(\mathcal{MSQ}))$ and $(\prec^{\checkmark}, PC(\mathcal{MSQ}))$ by instantiating the sets $\mathcal{MSQ}_{\times}$ and $\mathcal{MSQ}_{\checkmark}$, respectively. Sets $\mathcal{MSQ}_{\times}$ and $\mathcal{MSQ}_{\checkmark}$ share the majority of actions in $actions(\mathcal{MSQ})$. The difference is that $(\prec^{\times}, PC(\mathcal{MSQ}))$ includes $enq5$ and $deq2$, the actions that falsify π_p, but does not include actions corresponding to "upward pointing" edges in Fig. 3. Conversely, because some process has made progress since π_p was falsified, relation $(\prec^{\checkmark}, PC(\mathcal{MSQ}))$ includes the actions corresponding to "upward pointing" edges in Fig. 3, but does not include the actions that falsify π_p. Let us define $\mathcal{MSQ}_{*}$ as the actions common to both $\mathcal{MSQ}_{\times}$ and $\mathcal{MSQ}_{\checkmark}$.

$$\begin{aligned} enq_{*} &\mathrel{\widehat{=}} \{enq1, enq2, enq3, enq6, enq7t, enq8t, enq8f, enq9t, enq10, enq17t, enq17f\} \\ deq_{*} &\mathrel{\widehat{=}} \{deq3, deq4, deq5t, deq6t, deq6f, deq7t, deq7f, deq8, deq12, deq13t, deq14, deq19, deq20\} \\ \mathcal{MSQ}_{*} &\mathrel{\widehat{=}} enq_{*} \cup deq_{*} \end{aligned}$$

We now define $\mathcal{MSQ}_{\times}$ and $\mathcal{MSQ}_{\checkmark}$.

$$\begin{aligned} \mathcal{MSQ}_{\times} &\mathrel{\widehat{=}} \mathcal{MSQ}_{*} \cup \{enq5\} \cup \{deq2\} \\ \mathcal{MSQ}_{\checkmark} &\mathrel{\widehat{=}} \mathcal{MSQ}_{*} \cup \{enq7f, enq9f, enq13f\} \cup \{deq5f, deq10f, deq13f\} \end{aligned}$$

Theorem 43. *The relations $(\prec^{\times}, PC(\mathcal{MSQ}))$ and $(\prec^{\checkmark}, PC(\mathcal{MSQ}))$ are well-founded.*

Proof. By inspection of the control-flow graph: the subset of edges forms directed, acyclic graphs. □

Theorem 44. *The relation $(\prec, states(\mathcal{MSQ}))$ is well-founded.*

Proof. Follows from general Theorems 38 and 33, and from Theorem 43. □

[1] Condition (37) is stronger than is required. That is, we could replace $\delta_q(s') = \delta_q(s)$ in (37) by $\delta_q(s') = \delta_q(s) \vee \delta_q(s') \lll \delta_q(s)$, which means execution of p can improve the state processes other than p, but (37) is more convenient for the proof.

3.3.3. Discussion

To better understand how the relation $(\prec, states(\mathcal{MSQ}))$ is useful in a proof of lock-freedom, let us consider the following scenario. Assume that the queue is empty (i.e., contains only the dummy node), $procs(MSQ) = \{p, q\}$, and p and q are idle. Hence, $Head = Tail$, and $\delta_p = \delta_q = (\mathsf{T}, \mathsf{idle})$. (We will leave the state parameter to δ implicit in this example, and represent *true* and *false* values for π_p by T and F, respectively. Recall that π_p is initialised to T for all p.) Consider the following execution sequence.

(i) p begins an enqueue operation and executes without interruption up to program counter value $E9$. This gives the following chain of δ_p values, in which pc_p advances first with respect to $\prec^{\checkmark}$, then $\prec^{\times}$, since π_p is set to false when p enters the loop.

$$(\mathsf{T}, \mathsf{idle}), (\mathsf{T}, E1), (\mathsf{T}, E2), (\mathsf{T}, E3), (\mathsf{T}, E5), (\mathsf{F}, E6), (\mathsf{F}, E7), (\mathsf{F}, E8), (\mathsf{F}, E9)$$

(ii) q executes the same sequence of actions as p. Hence $\delta_p = \delta_q = (\mathsf{F}, E9)$.

(iii) p executes $enq9t$, adding a new node to the queue, which means $\delta_p = (\mathsf{T}, E17)$, $\delta_q = (\mathsf{T}, E9)$ and *Tail* is lagging. Although the values of both δ_p and δ_q are worse with respect to $(\lll, ProcInfo))$, this transition is allowed because p has executed an action in $progress_{\mathcal{MSQ}}$.

(iv) q resumes execution, and now fails the `CAS` at `E9`, hence executes $enq9f$ and $\delta_q = (\mathsf{T}, E5)$. This is an improvement within $(\lll, ProcInfo)$, since $E5 \prec^{\checkmark} E9$.

(v) q, now on its second iteration of `enq`.*Loop*, executes $enq5$, which falsifies π_q, and thus $\delta_q = (\mathsf{F}, E6)$. This is an improvement according to $(\lll, ProcInfo)$, because $\mathsf{F} \prec^{\mathbb{B}} \mathsf{T}$.

(vi) q resumes execution, with the test at $E8$ failing because *Tail* is now lagging. This generates the following chain of δ_q values:

$$(\mathsf{F}, E6), (\mathsf{F}, E7), (\mathsf{F}, E8), (\mathsf{F}, E13), (\mathsf{T}, E5)$$

The final transition, which executes $enq13t$, increases the value of δ_q according to $(\lll, ProcInfo))$. However, the transition is allowed because $enq13t \in progress_{\mathcal{MSQ}}$.

(vii) q continues execution, enqueues a new node and catches up to process p.

$$(\mathsf{T}, E5), (\mathsf{F}, E6), (\mathsf{F}, E7), (\mathsf{F}, E8), (\mathsf{F}, E9), (\mathsf{T}, E17)$$

Thus, $\delta_q = \delta_p$. Notice that the value of δ_p has not changed throughout execution of q.

(viii) p executes $enq17f \in complete_{\mathcal{MSQ}}$ (because $Tail \neq tail_p$) and starts a `deq` operation, executing up to $D3$, thus generates the chain of δ_p values: $(\mathsf{T}, E17), (\mathsf{T}, \mathsf{idle}), (\mathsf{T}, D2), (\mathsf{F}, D3)$

(ix) q executes completing action $enq17t$ (because $Tail = tail_q$), thus, we have $\delta_q = (\mathsf{T}, \mathsf{idle})$ and $\delta_p = (\mathsf{T}, D3)$. Again, although the values of δ_q and δ_p have not been decreased, this transition is allowed because $enq17t \in progress_{\mathcal{MSQ}}$.

(x) Now, although π_p holds, process p has not suffered from any real interference, thus, continued execution of p does not cause p to retry the loop. Instead, p will successfully complete the `deq` operation, as given by the following chain of δ_p values:

$$(\mathsf{T}, D3), (\mathsf{T}, D4), (\mathsf{T}, D5), (\mathsf{T}, D6), (\mathsf{T}, D12), (\mathsf{T}, D13), (\mathsf{T}, D19), (\mathsf{T}, D20), (\mathsf{T}, idle)$$

This scenario has shown the changing δ values for a process p executing an enqueue followed by a dequeue, concurrently with process q executing an enqueue. In general, all traces of a lock-free program follow this pattern, where a sequence of actions that improve the value of the state according to $(\prec, states_{\mathcal{MSQ}})$ are interspersed with actions from $progress_{\mathcal{P}}$.

3.4. Proof by case analysis on actions

In this section we prove that each transition obeys the well-founded relation. The following theorem allows us to prove the temporal definition (16) by case analysis on actions.

Theorem 45. *A program $\mathcal{P}$ that satisfies (18), (21) and (25) is lock-free if $(\prec, states(\mathcal{P}))$ is well-founded and given by Definition 36, and*

$$(\forall(\alpha, p, s, s'): transitions(\mathcal{P}) \bullet \alpha \notin progress_{\mathcal{P}} \implies \delta_p(s') \lll \delta_p(s)). \quad (46)$$

Proof. $\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(complete_{\mathcal{P}})$

$\Leftrightarrow$ {Theorem 26, using assumptions (18), (21), (25)}

$\mathcal{P} \models \Box\Diamond\, \mathsf{exec}(progress_{\mathcal{P}})$

$\Leftarrow$ {Theorem 11}

$(\forall t: transitions(\mathcal{P}) \bullet t.post \prec t.pre \vee t.\,\mathsf{exec}(progress_{\mathcal{P}}))$

$\Leftrightarrow$ {definition of exec, *transitions*}

$(\forall(\alpha, p, s, s'): transitions(\mathcal{P}) \bullet s' \prec s \vee \alpha \in progress_{\mathcal{P}})$

$\Leftrightarrow$ {logic}

$(\forall(\alpha, p, s, s') \colon transitions(\mathcal{P}) \bullet \alpha \notin progress_{\mathcal{P}} \Rightarrow s' \prec s)$
$\Leftrightarrow$ {Expand $\prec$ (Definition 36)}
$(\forall(\alpha, p, s, s') \colon transitions(\mathcal{P}) \bullet \alpha \notin progress_{\mathcal{P}} \Rightarrow$
$\quad (\exists r \colon procs(\mathcal{P}) \bullet \delta_r(s') \lll \delta_r(s) \wedge (\forall q \colon procs(\mathcal{P}) \bullet q \neq r \Rightarrow \delta_q(s') = \delta_q(s))))$
$\Leftarrow$ {Instantiate r with p}
$(\forall(\alpha, p, s, s') \colon transitions(\mathcal{P}) \bullet \alpha \notin progress_{\mathcal{P}} \Rightarrow$
$\quad \delta_p(s') \lll \delta_p(s) \wedge (\forall q \colon procs(\mathcal{P}) \bullet q \neq p \Rightarrow \delta_q(s') = \delta_q(s)))$
$\Leftarrow$ {from (35)}
$(\forall(\alpha, p, s, s') \colon transitions(\mathcal{P}) \bullet \alpha \notin progress_{\mathcal{P}} \Rightarrow \delta_p(s') \lll \delta_p(s))$ □

We have simplified the original definition of lock-freedom to a case analysis on transitions (in effect, actions), which requires the consideration of the local state of only a single process.

Theorem 47. *$\mathcal{MSQ}$ is lock-free.*

Proof. We define $progress_{\mathcal{MSQ}}$ as in (25), hence,

$$\begin{aligned} progress_{\mathcal{MSQ}} &= complete_{\mathcal{MSQ}} \cup exit_{\mathcal{MSQ}} \cup help_{\mathcal{MSQ}} \\ &= \{enq9t, enq13t, enq17t, enq17f, deq7t, deq10t, deq13t, deq20\} \end{aligned}$$

From Theorems 19 and 24, (18) and (21) hold for $\mathcal{MSQ}$, and from Theorem 44 $(\prec, states(\mathcal{MSQ}))$ is well-founded. We therefore prove lock-freedom by showing (46). This follows from case analysis on each action $\alpha \in action(\mathcal{P}) \setminus progress_{\mathcal{P}}$, by taking an arbitrary process p and arbitrary states s and s' to form transition $s \xrightarrow{\alpha_p} s'$.

The case analysis is split into three cases: $\alpha \in \mathcal{MSQ}_*$, $\alpha \in (\mathcal{MSQ}_\times \setminus \mathcal{MSQ}_*)$, and $\alpha \in (\mathcal{MSQ}_\checkmark \setminus \mathcal{MSQ}_*)$.

-**Case 1,** $\alpha \in \mathcal{MSQ}_*$. Note that because $\alpha \notin progress_{\mathcal{P}}$ and α does not modify π_p,

$$s'.\pi_p = s.\pi_p. \tag{48}$$

We prove the conclusion of (46).

$\delta_p(s') \lll \delta_p(s)$
$\Leftrightarrow$ {from (48) and (32)}
$(\neg s.\pi_p \Rightarrow s'.pc_p \prec^\times s.pc_p) \wedge (s.\pi_p \Rightarrow s'.pc_p \prec^\checkmark s.pc_p)$
$\Leftrightarrow$ {from (41), (42)}
$(\neg s.\pi_p \Rightarrow \alpha \in \mathcal{MSQ}_\times) \wedge (s.\pi_p \Rightarrow \alpha \in \mathcal{MSQ}_\checkmark)$
$\Leftarrow$ {definition of $\mathcal{MSQ}_\times$ and $\mathcal{MSQ}_\checkmark$}
$\alpha \in \mathcal{MSQ}_*$

-**Case 2,** $\alpha \in \{enq5, deq2\}$. These are the first actions in `enq`.*Loop* and `deq`.*Loop*, and hence they falsify π_p. We therefore assume

$$\neg s'.\pi_p \tag{49}$$

If $s.\pi_p$, then $\delta_p(s') \lll \delta_p(s)$ holds trivially because $s'.\pi_p \prec^{\mathbb{B}} s.\pi_p$. If $\neg s.\pi_p$, we prove the conclusion of (46) as follows.

$\delta_p(s') \lll \delta_p(s)$
$\Leftrightarrow$ {By (49) and assumption, $\neg s'.\pi_p$ and $\neg s.\pi_p$}
$s'.pc_p \prec^\times s.pc_p$
$\Leftrightarrow$ {from (41)}
$\alpha \in \mathcal{MSQ}_\times$
$\Leftarrow$ {from definition of $\mathcal{MSQ}_\times$}
$\alpha \in \{enq5, deq2\}$

-**Case 3,** $\alpha \in \{enq7f, enq9f, enq13f, deq5f, deq10f, deq13f\}$. These are the retry actions that correspond to the "upward pointing" edges in Fig. 3. We split these into three further cases, based on the invariant their proof relies upon. Because these actions do not modify π, we assume (48).

. $\alpha \in \{enq7f, enq13f, deq10f\}$.

Each of these transitions has guard $tail_p \neq Tail$. If $\neg s.\pi_p$ holds, by invariant (28) we have $s.tail_p = s.Tail$, which contradicts the guard of each α. That is, these transitions can be taken only if progress has occurred, and so this case may be ignored. If $s.\pi_p$ holds, we prove the conclusion of (46) as follows.

$\delta_p(s') \lll \delta_p(s)$
$\Leftarrow$ {By assumption $s.\pi_p$}
$s'.pc_p \prec^\checkmark s.pc_p$
$\Leftrightarrow$ {from (41)}

$$\begin{aligned}
&\alpha \in \mathcal{MSQ}_{\checkmark} \\
\Leftarrow\quad &\{\text{from definition of } \mathcal{MSQ}_{\checkmark}\} \\
&\alpha \in \{enq7f, enq9f, enq13f\}
\end{aligned}$$

. $\alpha \in \{deq5f, deq13f\}$. Each of these actions has guard $head_p \neq Head_p$. By (29), this case may be discharged using similar reasoning to the case above.
. $\alpha \in \{en9f\}$. This action has guard $tail_p.ptr \to next \neq nxt_p$, which indicates that the *next* pointer of the last node has changed. By (30), this case may be discharged using similar reasoning to the case above.

The case analysis required only simple propositional reasoning to discharge. This completes the proof that $\mathcal{MSQ}$ is lock-free. □

4. A bounded array-based queue

Colvin and Groves [3] present a bounded array-based lock-free queue and a proof of linearisability (safety). The program, $\mathcal{BAQ}$, is presented in Fig. 4, and the control-flow diagram for the enqueue operation is given in Fig. 5. $\mathcal{BAQ}$ is implemented using a bounded array, and hence, unlike $\mathcal{MSQ}$, the queue may become full. In this section we use the technique developed in Section 3 to prove that $\mathcal{BAQ}$ is lock-free.

4.1. The algorithm

The queue is implemented as an array `Q` of size `L`, with indices `Head` and `Tail` which are forever increasing integers, initially 0. Entries are enqueued onto `Tail mod L` and dequeued from `Head mod L`. Each entry is of type `(Val, ℕ)` where `Val` is the queue element type and the natural number is the modification counter for avoiding the ABA problem. The value field of an entry may be null, which is different from the value of all real queue values. An enqueue operation must both enter the new value into the array and increment `Head`; similarly a dequeue must remove a value (replace it with null) and increment `Tail`. Using only a `CAS`, these two modifications cannot be performed atomically, hence, as with $\mathcal{MSQ}$, the `Head` and `Tail` indices may lag behind the true head and tail. As shown by Colvin and Groves [3], the indices `Head` and `Tail` can lag by at most one. Because two indices must be kept up-to-date, a single process may execute *op.Loop* up to three times, modifying a shared variable on each iteration, before finally completing. In comparison, this may happen in $\mathcal{MSQ}$ only twice, and in the lock-free stack of Michael and Scott [15,2] only once.

Below we describe the `enq` operation in detail; the `deq` operation is symmetric. Consider a process p performing an `enq(y)` operation without interference. The queue has a maximum size of 6 ($L = 6$), and there are currently 3 elements in the queue, with `Tail` $= 3$ and `Head` $= 0$, hence, neither is lagging. This is depicted below by Queue state A. We indicate the indices `Head mod L` and `Tail mod L` by `H` and `T`, respectively.

Queue state A

0	1	2	3	4	5
v_1	v_2	v_3	null	null	null
H			T		

Queue state B

0	1	2	3	4	5
v_1	v_2	v_3	y	null	null
H				T	

Process p takes a snapshot of `Tail`, the element at that index (null), and a snapshot of `Head` (lines `e2`–`e4`). Because there is no interference the condition at `e5` evaluates to *false*, and because the queue does not appear full the test at `e8` fails. Furthermore, the test at `e17` succeeds (since there is a space in the array), and the CAS at `e18` succeeds (since the array `Q` has not changed). Value `y` replaces the null at `Q[3]` and the modification counter at `Q[3]` is incremented. At `e19`, `Tail` is incremented to 4, giving Queue state B, and the operation terminates.

Consider an enqueue on a full queue, with `Head` $= 0$ and `Tail` $= 6$, as depicted in Queue state C below. The test `e8` succeeds; the test at `e9` succeeds (since there are no non-null elements in the queue); and `e10` succeeds (since there is no interference). The operation terminates, returning the special value full to indicate the queue is full.

Queue state C

0	1	2	3	4	5
v_1	v_2	v_3	v_4	v_5	v_6
H/T					

Queue state D

0	1	2	3	4	5
null	v_2	v_3	v_4	v_5	v_6
H					T

From the point of view of lock-freedom, an interesting case is the situation in which there are $L - 1$ elements in the queue, and both `Head` and `Tail` are lagging. This situation is depicted in Queue state D above. The non-lagging values of `Head` and `Tail` should be 1 and 6. Process p progresses as before, discovers that `Tail` is lagging and increments it to 6 at `e24`. The second time through the loop, the test at `e8` succeeds ($6 = 0 + 6$). The test at `e9` fails, because `Q[0]` is null. This implies that `Head` is lagging, so it is incremented at `e14` and the loop is started for a third time. This time, both `Head` and `Tail` are accurate, and the new value is placed into the queue at `e18`. This case shows that the behaviour of an individual process in the absence of interference may still be quite complex, requiring several iterations of the loop.

If another process interferes with p by enqueuing an element, the tests at `e5` or `e18` will fail, forcing p to retry `enq`*.Loop*. If the snapshot `x` is found to be non-null at `e17`, p checks at `e23` whether the location is still non-null, before attempting to help the lagging `Tail` at `e24`.

```
L: ℕ₁
entry ≙ {val: Val, mod: ℕ }
Q: 0..L − 1 → entry
Head, Tail: ℕ
```

```
enq(v: Val): {ok, full}≙

     enq_retry:
e2:  tail := Tail;
e3:  x := Q[tail mod L];
e4:  head := Head;
e5:  if tail != Tail then
       goto enq_retry;
     fi;
e8:  if tail = Head+L
e9:    if Q[head mod L].val !=  null
e10:     if head = Head
e11:       return full;
         fi;
       fi;
e14:   CAS(Head, head, head+1);
       goto enq_retry;
     fi;
e17: if x.val = null
e18:   if CAS(Q[tail mod L], x, <v, x.mod+1>)
e19:     CAS(Tail, tail, tail+1);
e20:     return ok;
       fi;
     else
e23:   if Q[tail mod L].val != null
e24:     CAS(Tail, tail, tail+1);
       fi;
     fi;
     goto enq_retry;
```

```
deq(): Val ∪ {empty}≙

     deq_retry:
d2:  head := Head;
d3:  x := Q[head mod L];
d4:  tail := Tail;
d5:  if head != Head then
       goto deq_retry;
     fi;
d8:  if head = Tail
d9:    if Q[tail mod L].val = null
d10:     if tail = Tail
d11:       return empty;
         fi
       fi;
d14:   CAS(Tail, tail, tail+1);
       goto deq_retry;
     fi;
d17: if x.val != null
d18:   if CAS(Q[head mod L], x, <null, x.mod+1>)
d19:     CAS(Head, head, head+1);
d20:     return x.val;
       fi
     else
d23:   if Q[head mod L].val = null
d24:     CAS(Head, head, head+1);
       fi
     fi
     goto deq_retry;
```

Fig. 4. The bounded array-based queue.

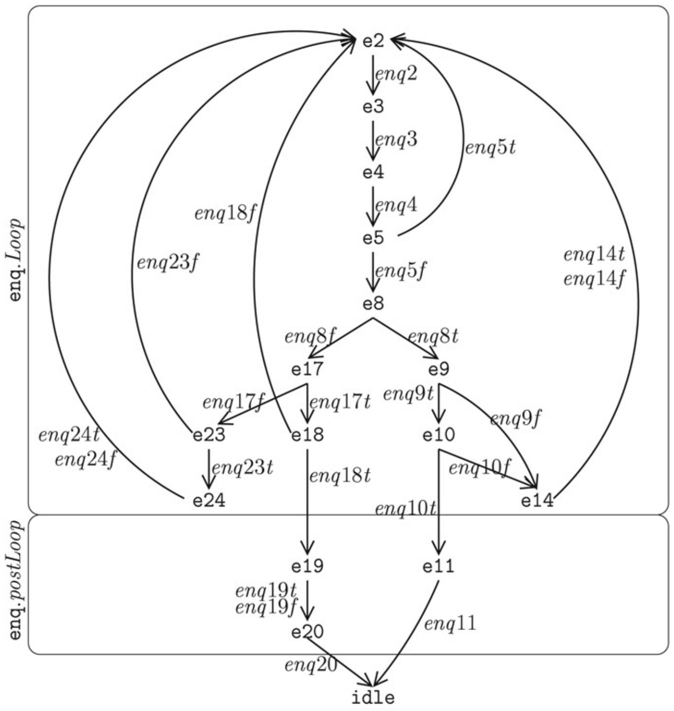


Fig. 5. Control-flow diagram for the bounded array enqueue operation.

4.2. *Proof that $\mathcal{BAQ}$ is lock-free*

We follow the four-step process outlined in Section 3.

Step 1: Define the progress actions. We must first identify the set of progress actions. Using definitions(14), (17) and (25) and identifying the helping actions as those that increment *Head* and *Tail*, we obtain the following sets:

$$\begin{aligned} complete_{\mathcal{BAQ}} &= \{enq11, enq20, deq11, deq20\} \\ exit_{\mathcal{BAQ}} &= \{enq10t, enq18t, deq10t, deq18t\} \\ help_{\mathcal{BAQ}} &= \{enq14t, enq19t, enq24t, deq14t, deq19t, deq24t\} \\ progress_{\mathcal{BAQ}} &= complete_{\mathcal{BAQ}} \cup exit_{\mathcal{BAQ}} \cup help_{\mathcal{BAQ}} \end{aligned}$$

Because $\mathcal{BAQ}$ is a program of the form in Fig. 1, Theorem 19 implies that property (18) holds. We prove that the sets $exit_{\mathcal{BAQ}}$ and $help_{\mathcal{BAQ}}$ satisfy property (21) as follows.

Theorem 50. *Property* (21) *holds for* $\mathcal{BAQ}$.

Proof. We take a similar approach to the proof of Theorem 24. We define predicates that indicate whether `Head` and `Tail` are lagging.

$$HL \mathrel{\widehat{=}} Q[Head \bmod L] = \mathsf{null} \wedge Head < Tail$$
$$TL \mathrel{\widehat{=}} Q[Tail \bmod L] \neq \mathsf{null} \wedge Tail < Head + L$$

We may then show the following equivalences, which identify the actions which cause and correct lagging indices.

$$\mathsf{exec}(\{enq14t, deq19t, deq24t\}) \Leftrightarrow HL \wedge \bigcirc\neg HL \qquad (51)$$
$$\mathsf{exec}(\{deq14t, enq19t, enq24t\}) \Leftrightarrow TL \wedge \bigcirc\neg TL \qquad (52)$$
$$\mathsf{exec}(\{enq18t\}) \Leftrightarrow \neg TL \wedge \bigcirc TL \qquad (53)$$
$$\mathsf{exec}(\{deq18t\}) \Leftrightarrow \neg HL \wedge \bigcirc HL \qquad (54)$$

The proof of (21) follows.

$$\begin{aligned} & \Box\Diamond(\mathsf{exec}(help_{\mathcal{BAQ}})) \\ \Leftrightarrow\ & \quad \{\text{definition of } help_{\mathcal{BAQ}}, (7)\} \\ & \Box\Diamond(\mathsf{exec}(\{enq14t, deq19t, deq24t\})) \vee \Box\Diamond(\mathsf{exec}(\{deq14t, enq19t, enq24t\})) \\ \Leftrightarrow\ & \quad \{(51) \text{ and } (52)\} \\ & \Box\Diamond(HL \wedge \bigcirc\neg HL) \vee \Box\Diamond(TL \wedge \bigcirc\neg TL) \\ \Leftrightarrow\ & \quad \{(10) \text{ twice}\} \\ & \Box\Diamond(\neg HL \wedge \bigcirc\neg HL) \vee \Box\Diamond(\neg TL \wedge \bigcirc TL) \\ \Leftrightarrow\ & \quad \{(53) \text{ and } (54), (7)\} \\ & \Box\Diamond\ \mathsf{exec}(\{enq18t, deq18t\}) \\ \Rightarrow\ & \quad \{\text{definition}\} \\ & \Box\Diamond\ \mathsf{exec}(exit_{\mathcal{BAQ}}) \quad \Box \end{aligned}$$

Step 2: Augment with progress detection. We introduce the progress-detection variable, π. Each process sets π_p to false at *enq*2 and *deq*2, the first action in *enq.Loop* and *deq.Loop*, and π_p is set to $(\lambda p{:}\ procs(\mathcal{BAQ}) \bullet true)$ by each action in $progress_{\mathcal{BAQ}}$.

We may prove the following invariants.

$$(\forall p{:}\ procs(\mathcal{BAQ}) \bullet \neg\pi_p \Rightarrow$$
$$pc_p \notin \{d3, d5\} \Rightarrow tail_p = Tail \wedge \qquad (55)$$
$$pc_p \notin \{e3, e5\} \Rightarrow head_p = Head \wedge \qquad (56)$$
$$pc_p \in PCEnq \setminus \{e3\} \Rightarrow x_p = Q[tail_p \bmod L] \wedge \qquad (57)$$
$$pc_p \in PCDeq \setminus \{d3\} \Rightarrow x_p = Q[head_p \bmod L]) \wedge \qquad (58)$$
$$(\forall p{:}\ procs(\mathcal{BAQ}) \bullet pc_p = e23 \Rightarrow x_p.val \neq \mathsf{null}) \wedge \qquad (59)$$
$$(\forall p{:}\ procs(\mathcal{BAQ}) \bullet pc_p = d23 \Rightarrow x_p.val = \mathsf{null}) \qquad (60)$$

Invariants (55)–(58) state that, if no progress has been made, the local copies are accurate as long as $\neg\pi_p$ holds. Invariants (59) and (60) do not rely on π, and follow from the failed tests at $e17$ and $d17$, respectively. Invariant (59) is required to prove that a retry via a failed test $e23$ implies progress has been made, i.e., that $Q[tail \bmod L]$ has been modified (in combination with (57)). The symmetric invariant (60) is required for operation `deq`.

Step 3: The well-founded relation. We define the sets enq_*, $enq_\times$ and $enq_\checkmark$.

$$\begin{aligned} enq_* &\mathrel{\widehat=} \{enq3, enq4, enq5f, enq8t, enq8f, enq9t, enq9f, enq10t,\\ &\quad enq10f, enq17t, enq17f, enq19t, enq19f, enq23t\}\\ enq_\times &\mathrel{\widehat=} enq_* \cup \{enq2\}\\ enq_\checkmark &\mathrel{\widehat=} enq_* \cup \{enq5t, enq14f, enq18f, enq23f, enq24f\} \end{aligned}$$

Hence $enq_\times$ for $\mathcal{BAQ}$ does not contain the upward pointing actions, and $enq_\checkmark$ does not contain $enq2$. Sets deq_*, $deq_\times$, and $deq_\checkmark$ are symmetrically defined.

We now define

$$\begin{aligned} \mathcal{BAQ}_* &\mathrel{\widehat=} enq_* \cup deq_*\\ \mathcal{BAQ}_\times &\mathrel{\widehat=} enq_\times \cup deq_\times\\ \mathcal{BAQ}_\checkmark &\mathrel{\widehat=} enq_\checkmark \cup deq_\checkmark \end{aligned}$$

Theorem 61. *The relation* $(\prec, states(\mathcal{BAQ}))$ *is well-founded*

Proof. The relations $(\prec^{\mathbb{B}}, \mathbb{B})$, $(\prec^{\times}, PC(\mathcal{BAQ}))$ and $(\prec^{\checkmark}, PC(\mathcal{BAQ}))$ are defined as in (2), (39) and (40), respectively. Relations $(\prec^{\times}, PC(\mathcal{BAQ}))$ and $(\prec^{\checkmark}, PC(\mathcal{BAQ}))$ are well-founded because the corresponding control-flow graphs are acyclic. We define relation $(\lll, ProcInfo(\mathcal{BAQ}))$ using (32) and relation $(\prec, states(\mathcal{BAQ}))$ using (37). By Theorem 33, $(\lll, ProcInfo)$ is well-founded, and hence, by Theorem 38, $(\prec, states(\mathcal{BAQ}))$ is well-founded. □

Step 4: Case analysis. The proof follows the same pattern as that for $\mathcal{MSQ}$: we show (46) using a three-way case analysis on actions and using invariants (55)–(60) to discharge the obligations on the third case. We omit the details here, since they involve only propositional reasoning and have been verified using PVS [20].

Theorem 62. *Program* $\mathcal{BAQ}$ *is lock-free.*

Proof. From Theorems 19 and 50, (18) and (21) hold for $\mathcal{BAQ}$, and from Theorem 61, $(\prec, states(\mathcal{BAQ}))$ is well-founded. By propositional reasoning verified in PVS (see [20]), (46) holds for $\mathcal{BAQ}$. Therefore by Theorem 45, $\mathcal{BAQ}$ is lock-free. □

5. Tool support

We have encoded the programs and relevant theorems in the PVS theorem prover [18] (available online [20]), and have devised proof strategies to automate as much of the work as possible. We have also written a generic PVS theory, `well_foundedness`, which contains the generic relation on type *ProcInfo* and a proof that it is well-founded (Theorem 33). The PVS code for `well_foundedness` is listed in Appendix A.1.

Proving that a program is lock-free using PVS consists of the following steps from the process outlined in Section 3:

(i) *Encode the supporting definitions and program as a transition system.*

This includes defining types *PC*, *action*, *state* and any other algorithm-specific types (pointers, nodes, bounded arrays, etc.). The transitions, which include the updates to the auxiliary variable π, are defined via three functions: `pcpair`, which returns the *pcpre*/*pcpost* pair for an action; `guards`, which returns a predicate on *state* representing the guard of the transitions (4); and `updates`, a function from *state* to *state*, representing the updates in (4). These are combined to define `trans`, representing the transition function.

Other definitions required include the set of progress transitions (`progress_transitions`) and the invariants (`inv`). The encoding of (46) is given below, with the addition of the invariant in the antecedent.

```
ind: THEOREM
    FORALL alpha, p, s0, s:
        trans(alpha, p, s0, s) and
        not progress_transitions(alpha) and
        inv(s0, p) IMPLIES
            lt_msq_proc(delta(s)(p), delta(s0)(p))
```

The definition of the relation `lt_msq_proc` is discussed below.

An abbreviated listing of the PVS code for $\mathcal{MSQ}$ is given in Appendix A.2 and the full listing is available online [20].

(ii) *Encode the relations* $(\prec^{\times}, PC(\mathcal{P}))$ *and* $(\prec^{\checkmark}, PC(\mathcal{P}))$.

The relations $(\prec^{\times}, PC(\mathcal{P}))$ and $(\prec^{\checkmark}, PC(\mathcal{P}))$ are defined as `lt_pc_cross` and `lt_pc_tick`, respectively. For the $\mathcal{MSQ}$ proof we defined `lt_pc_common`, representing $\mathcal{MSQ}_*$. For example, given that function `pcrevpair` returns the reverse relation to `pcpair`, `lt_pc_cross` is defined as follows.

```
lt_pc_cross(pc, pc0: PC): boolean =
    (pc, pc0) = pcrevpair(enq5) OR
    (pc, pc0) = pcrevpair(deq2) OR
    lt_pc_common(pc, pc0)
```

The two relations on *PC* are used to instantiate the generic definition of `lt_proc` in theory `well_foundedness` in Appendix A.1. This relation is called `lt_msq_proc`. The full definitions may be found in Appendix A.2.

(iii) *Prove that* `lt_pc_cross` *and* `lt_pc_tick` *are well-founded.*

We prove this with respect to the PVS definition `well_founded?`, which encodes Definition 1. The proof is conducted by case analysis on whether or not each $pc \in PC$ is in some arbitrary set, S. The proof starts with a minimal element in S, then considers non-decreasing values of *PC* until all values in *PC* have been considered. To automate this task we defined a strategy `prove-wf`, which takes an ordered list of *PC* values, where x appears before y if x is less than y. For example, the relation $(\prec^{\times}, PC(\mathcal{MSQ}))$ may be proved to be well-founded via the following PVS command.

```
(prove-wf (
    E17 E10 E9 E13 E8 E7 E6 E5 E3 E2 E1
    D20 D19 D14 D13 D12 D8 D10 D7 D6 D5 D4 D3 D2
    idle
))
```

By inspection of the control-flow graph (Fig. 3) and ignoring idle for the moment, the ordering of the list for `enq` and `deq` *pc* values is simply the topological ordering corresponding to $\mathcal{MSQ}_{\times}$ for the `enq` and `deq` operations, respectively. Value idle is the greatest element for both `enq` and `deq` program counter values by definition. Intuitively, it is the greatest element because it is the furthest away from completing an operation.

Given the proofs of well-foundedness for the two program counter relations, it is trivial to use the generic proof in theory `well_foundedness` to prove that Theorem 33 holds.

(iv) Prove the case analysis.

As indicated earlier, the bulk of the proof is simple propositional reasoning. The PVS command `grind` is capable of automatically discharging such obligations for each action `alpha`. However, the proof must be initially set up by doing the case analysis (structural induction) on actions. We developed a simple proof strategy `prove-case-analysis` to bundle these tasks together.

```
(defstep prove-case-analysis()
    (branch (induct alpha) ( (grind) )))
```

The strategy first applies the `(induct alpha)` command, which splits the proof obligation into a subproof for each action in the program, and then applies `grind` to each subproof. This behaviour is achieved using the `branch` strategy. Proving `ind` using `prove-case-analysis` for $\mathcal{MSQ}$, PVS takes approximately 38 s on a standard laptop (36 actions), and for $\mathcal{BAQ}$ approximately 80 s (52 actions). This time could potentially be shortened by applying a custom-built strategy instead of `grind`.

6. Conclusions

In this paper we have presented a general technique for proving lock-freedom. The technique is based on the observation that at each step of the program as a whole, one process is closer to making substantive progress, and at any point there are only a finite number of interference-free steps before this happens. The act of making progress may interfere with other processes and set them back in achieving their goal, but this is allowed because the program as a whole is making progress.

As the final step in the proof, a case analysis on each statement in the program is performed. This case analysis will be trivial if the program is lock-free and the technique has been followed correctly. Indeed, the case analysis can be proved using the theorem prover PVS with minimal user interaction. We have also provided reusable PVS proof strategies for showing well-foundedness of the *pc* relations for particular programs, and given a general result that the generic relation on program states is well-founded. As shown by the proof of lock-freedom for the bounded array queue [3] in Section 4, the technique is straightforward to apply and the resulting proof obligations can be discharged in PVS.

The theory for the proofs is based on transition systems, in which traces explicitly include the program statement that is executed at each step. This has allowed us to define lock-freedom in a way which is more direct and amenable to proof than in earlier work [8,2]. Because the case analysis proofs are based on augmenting the program with a simple Boolean array variable, the technique lends itself to model checking. As future work, we will investigate how a program may be annotated for this purpose.

6.1. Discussion of the proof approach

The mechanics of the proof requires identifying a set of progress statements in the operations, augmenting the program with auxiliary progress-detection variables, and devising two well-founded relations on *pc* values. These steps are straightforward and guided by heuristics. Invariants that give the relationship between progress and snapshots are required, and typically will also be straightforward to prove. It must also be shown that if progress occurs infinitely often, then it must be the case that some process completes infinitely often. The proof of this property may not be straightforward, but, as is indicated by the examples in this paper, such properties will be required in a safety proof of the program, and hence not a particular overhead on the progress proof.

The technique we have described is based around the principle that if "progress" has been made by some other process, then it is allowable to retry the loop. This is regardless of whether or not the "progress" has actually caused interference or not, and it may seem that a more direct proof approach, which would not need the progress-detection variables nor the use of the set $help_{\mathcal{P}}$, is possible. This is the approach the authors originally took, and was applied successfully to a lock-free stack algorithm [2]. However, when applied to the queue algorithm in Fig. 2, this approach resulted in the creation of complex tuples to describe what it meant for the state of the queue to be improved, taking into account the lagging `Tail` and also that `enq` and `deq` operations have different views of what constitutes interference: an enqueue is not interfered with by dequeues, and dequeues are not interfered with by enqueues, except when there are zero or one elements in the queue. The book-keeping overhead became too large, and the proof was not generalisable.

We also tried to avoid treating the $help_{\mathcal{P}}$ transitions as members of $progress_{\mathcal{P}}$. This approach did not straightforwardly work for $\mathcal{MSQ}$, because a dequeuer could retry the loop after executing $deq10t$, even though the queue is non-empty and *Head* is accurate. Interestingly, this behaviour is closely related to the behaviour of a dequeue operation that required the safety proof of the algorithm in [7] to use backward simulation. Backward simulation, although an accepted technique, is a more complex verification technique to apply than forward simulation, and appears more rarely than forward simulation in proofs of real algorithms. That the same part of the code caused complications for both progress and safety proofs is notable, if unsurprising.

The main proof, (46), is fully automated, and the proof of well-foundedness is automatic if the relations have been defined correctly and the *pc* values given as a list. A proof of lock-freedom also relies on safety invariants. We consider proofs of such invariants to lie outside the scope of our topic. In a full correctness proof of the algorithm, experience has shown (Section 3 and [7], and Section 4 and [3]) that the safety properties required to prove lock-freedom are also required in the safety proof.

6.2. Related work

In a previous work [2] we presented a proof of lock-freedom for a stack algorithm. The proof did not use progress-detection variables and was not based on generic principles. When applying the program-specific technique to the queue algorithm given in this paper, the approach failed due to problems with interference.

Gao and Hesselink [10] present a general lock-free algorithm using Compare-and-Swap, and for which they prove lock-freedom using a counter recording the number of completed processes. The counter serves a similar purpose to our progress-detection variables. However, the lock-freedom proof is argued informally, and cannot be applied generically. It also assumes weak-fairness, which is not implemented on most systems. In contrast our technique is fully formal, generic, and does not make any fairness assumptions about the system. Derrick et al. [5] claim to prove lock-freedom for the same stack algorithm as in [2], by counting the number of steps left to completion, similarly to our well-founded relation. However, they consider only two processes operating in parallel, greatly reducing the potential for interference, and hence do not give a general result for an arbitrary number of processes. Furthermore, they do not give a formal definition of lock-freedom.

Our approach of treating programs as transition systems and conducting proofs in PVS is the approach taken in related work of proving safety properties of lock-free programs [3,7,4]. This work proves that the programs satisfy linearisability [12], but are not concerned with progress. Our intention is that the lock-freeness proofs can reuse and augment the PVS definitions and, more importantly, the invariants. Fully formal and machine-checked proofs of safety and progress can then sit together, with relatively little extra effort required for the latter.

Acknowledgements

The authors thank Ian Hayes for helpful discussions on the paper, and three anonymous reviewers for their constructive comments. The second author was supported by Australian Research Council Discovery Grant DP0558408, *Analysing and generating fault-tolerant real-time systems*.

Appendix. PVS code

A.1. Theory `well_foundedness`

```
well_foundedness[PC: TYPE]: THEORY
BEGIN
  b,b0: VAR boolean
  i,i0: VAR [boolean, PC]
  lt_bool(b, b0):boolean = not b and b0
  lt_pc_cross, lt_pc_tick: VAR pred[[PC, PC]]

  lt_proc
    (lt_pc_cross: pred[[PC, PC]])
    (lt_pc_tick: pred[[PC, PC]])
    (i, i0): boolean =
      lt_bool(i`1, i0`1)  or
      (i`1 = i0`1 and
        ((not i`1 and lt_pc_cross(i`2, i0`2)) or
        (i`1 and lt_pc_tick(i`2, i0`2)))
      )

  wflt_bool: THEOREM well_founded?(lt_bool)
  wflt_proc: THEOREM
    well_founded?(lt_pc_cross) and
    well_founded?(lt_pc_tick) implies
      well_founded?(lt_proc(lt_pc_cross)(lt_pc_tick))
end well_foundedness
```

A.2. *Theory for $\mathcal{MSQ}$*

```
msqProgd: THEORY BEGIN

  PROC: TYPE
  PC: TYPE = {
    idle,
    E1, E2, E3, E5, E6, E7, E8, E9, E10, E13, E17,
    D2, D3, D4, D5, D6, D7, D8, D10, D12, D13, D14, D19, D20
  }

  action: TYPE = {
    startEnq, enq1, enq2, enq3, enq5, enq6, enq7t, enq7f, enq8t, enq8f,
    enq9t, enq9f, enq10, enq13t, enq13f, enq17t, enq17f,
    startDeq, deq2, deq3, deq4, deq5t, deq5f, deq6t, deq6f, deq7t, deq7f,
    deq8, deq10t, deq10f, deq12, deq13t, deq13f, deq14, deq19, deq20
  }
  [ type definitions ]

  state: TYPE =
  [#
     progd: [PROC->boolean],

     Tail:  pointer_t,
     Head:  pointer_t,
     next:  [node_t_ptr->pointer_t],
     val:   [node_t_ptr -> Val],
     tail:  [PROC -> pointer_t],
     head:  [PROC -> pointer_t],
     nxt:   [PROC -> pointer_t],
     pc:    [PROC -> PC],

     [ other declarations ]

  #]


  pcpair(alpha): [PC,PC] = CASES (alpha) OF
    startEnq: (idle, E1),
    enq1: (E1, E2),
    enq2: (E2, E3),
    ....
    deq14: (D14, D19),
    deq19: (D19, D20),
    deq20: (D20, idle)
  ENDCASES

  pcpre(alpha): PC = pcpair(alpha)`1
```

```
pcpost(alpha): PC = pcpair(alpha)`2
pcrevpair(alpha): [PC,PC] = (pcpost(alpha), pcpre(alpha))


guards(alpha)(p)(s): boolean = CASES (alpha) OF
  enq7t: s`tail(p) = s`Tail,
  enq7f: NOT (s`tail(p) = s`Tail),
  ....
  deq13t: s`Head = s`head(p),
  deq13f: NOT (s`Head = s`head(p))
  else true
 ENDCASES


updates(alpha)(p)(s): state = CASES (alpha) OF
  ....
  enq5: s WITH [`tail(p) := s`Tail, `progd(p) := false],
  ....
  enq13t: s WITH [
  `Tail := (# ptr:= s`nxt(p)`ptr, count := s`tail(p)`count + 1 #),
     `progd := (lambda p: true)],
  enq13f: s,
  ....
  deq20: s
 ENDCASES


grd(alpha)(p)(s): boolean =
  guards(alpha)(p)(s) and s`pc(p) = pcpre(alpha)
eff(alpha)(p)(s): state =
  updates(alpha)(p)(s) with [`pc(p) := pcpost(alpha)]
trans(alpha, p, s0, s): boolean =
  grd(alpha)(p)(s0) and s = eff(alpha)(p)(s0)


lt_pc_common(pc, pc0: PC): boolean =
  (pc, pc0) = pcrevpair(startEnq) OR
  (pc, pc0) = pcrevpair(enq1) OR
  (pc, pc0) = pcrevpair(enq2) OR
  ....
  (pc, pc0) = pcrevpair(deq19)

lt_pc_cross(pc, pc0: PC): boolean =
  (pc, pc0) = pcrevpair(enq5) OR
  (pc, pc0) = pcrevpair(deq2) OR
  lt_pc_common(pc, pc0)

lt_pc_tick(pc, pc0: PC): boolean =
  (pc, pc0) = pcrevpair(enq7f) OR
  (pc, pc0) = pcrevpair(enq9f) OR
  (pc, pc0) = pcrevpair(enq13f) OR
  (pc, pc0) = pcrevpair(deq5f) OR
  (pc, pc0) = pcrevpair(deq10f) OR
  (pc, pc0) = pcrevpair(deq13f) OR
  lt_pc_common(pc, pc0)


ProcInfo: TYPE = [boolean, PC]
delta(s)(p): ProcInfo = (s`progd(p), s`pc(p))


importing orders
importing well_foundedness[PC]
```

```
  wflt_cross: LEMMA well_founded?(lt_pc_cross)
  wflt_tick: LEMMA well_founded?(lt_pc_tick)

  lt_msq_proc: pred[[ProcInfo, ProcInfo]] = lt_proc(lt_pc_cross)(lt_pc_tick)

  wflt_msq_proc: LEMMA well_founded?(lt_msq_proc)


  progress_transitions(alpha): boolean =
    alpha = enq9t or alpha = enq13t or alpha = enq17t or alpha = enq17f or
    alpha = deq8 or alpha = deq10t or alpha = deq13t or alpha = deq20


  inv(s, p): boolean =
    Let pc = s`pc(p) in
    not s`progd(p) implies
      (pc /= d3 implies s`tail(p) = s`Tail) and
      (not PCEnq(pc) implies s`head(p) = s`Head) and
      (PCEnq(pc) and pc /= e6 implies s`nxt(p) = s`next(s`tail(p)`ptr))

  ind: THEOREM
    FORALL alpha, p, s0, s:
      trans(alpha, p, s0, s) and
      not progress_transitions(alpha) and
      inv(s0, p) IMPLIES
        lt_msq_proc(delta(s)(p), delta(s0)(p))

END msqProgd
```

References

[1] K.M. Chandy, J. Misra, Parallel Program Design: A Foundation, Addison-Wesley Longman Publishing Co., Inc., 1988.
[2] R. Colvin, B. Dongol, Verifying lock-freedom using well-founded orders, in: C.B. Jones, Z. Liu, J. Woodcock (Eds.), Proceedings of 4th International Colloquium on Theoretical Aspects of Computing, ICTAC 2007, in: Lecture Notes in Computer Science, vol. 4711, Springer, 2007.
[3] R. Colvin, L. Groves, Formal verification of an array-based nonblocking queue, in: 10th International Conference on Engineering of Complex Computer Systems, ICECCS 2005, IEEE Computer Society, 2005.
[4] R. Colvin, L. Groves, A scalable lock-free stack algorithm and its verification, in: Fifth IEEE International Conference on Software Engineering and Formal Methods, SEFM 2007, IEEE Computer Society, 2007.
[5] J. Derrick, G. Schellhorn, H. Wehrheim, Proving linearizability via non-atomic refinement, in: J. Davies, J. Gibbons (Eds.), Integrated Formal Methods, 6th International Conference, IFM 2007, Oxford, UK, July 2–5, 2007, Proceedings, in: Lecture Notes in Computer Science, vol. 4591, Springer, 2007.
[6] E.W. Dijkstra, C.S. Scholten, Predicate calculus and program semantics, Springer-Verlag New York, Inc., New York, NY, USA, 1990.
[7] S. Doherty, L. Groves, V. Luchangco, M. Moir, Formal verification of a practical lock-free queue algorithm, in: Proceedings of Formal Techniques for Networked and Distributed Systems, FORTE 2004, 24th IFIP WG 6.1 International Conference, in: Lecture Notes in Computer Science, vol. 3235, Springer, 2004.
[8] B. Dongol, Formalising progress properties of non-blocking programs, in: Z. Liu, J. He (Eds.), Formal Methods and Software Engineering (8th International Conference on Formal Engineering Methods ICFEM 2006), in: Lecture Notes in Computer Science, vol. 4260, Springer, 2006.
[9] L. Fix, O. Grumberg, Verification of temporal properties, J. Logic Comput. 6 (3) (1996) 343–361.
[10] H. Gao, W.H. Hesselink, A general lock-free algorithm using compare-and-swap, Inform. and Comput. 205 (2) (2007) 225–241.
[11] D. Gries, F.B. Schneider, A Logical Approach to Discrete Math, Springer-Verlag New York, Inc., New York, NY, USA, 1993.
[12] M.P. Herlihy, J.M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Trans. Program. Lang. Syst. 12 (3) (1990) 463–492.
[13] Z. Manna, P. Pnueli, Temporal Verification of Reactive and Concurrent Systems: Specification, Springer-Verlag New York, Inc., 1992.
[14] H. Massalin, C. Pu, A lock-free multiprocessor OS kernel, Tech. Rep. CUCS-005-91, Columbia University, New York, 1991.
[15] M.M. Michael, M.L. Scott, Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors, J. Parallel Distrib. Comput. 51 (1) (1998) 1–26.
[16] J. Misra, A Discipline of Multiprogramming, Springer-Verlag, 2001.
[17] M. Moir, Practical implementations of non-blocking synchronization primitives, in: Proceedings of the Twentieth Annual ACM Symposium on Principles of Distributed Computing, PODC 2001, 1997.
[18] S. Owre, S. Rajan, J.M. Rushby, N. Shankar, M.K. Srivas, PVS: Combining specification, proof checking, and model checking, in: R. Alur, T. Henzinger (Eds.), 8th International Conference on Computer Aided Verification, CAV 1996, in: Lecture Notes in Computer Science, vol. 1102, Springer-Verlag, 1996.
[19] Java API: Class ConcurrentLinkedQueue, 2008. http://java.sun.com/javase/6/docs/api/java/util/concurrent/ConcurrentLinkedQueue.html.
[20] Encoding of programs and proof technique in PVS theorem prover, 2008. http://www.itee.uq.edu.au/~nbverif/.