Temporal logic verification of lock-freedom

Bogdan Tofan, Simon Bäumler, Gerhard Schellhorn, Wolfgang Reif

Angaben zur Veröffentlichung / Publication details:

Tofan, Bogdan, Simon Bäumler, Gerhard Schellhorn, and Wolfgang Reif. 2010. "Temporal logic verification of lock-freedom." In *Mathematics of Program Construction: 10th International Conference, MPC 2010, Québec City, Canada, June 21-23, 2010, proceedings*, edited by Claude Bolduc, Jules Desharnais, and Béchir Ktari, 377–96. Berlin: Springer. https://doi.org/10.1007/978-3-642-13321-3_21.

Temporal Logic Verification of Lock-Freedom

Bogdan Tofan, Simon Bäumler, Gerhard Schellhorn, and Wolfgang Reif

Institute for Software and Systems Engineering
University of Augsburg
D-86135 Augsburg, Germany
{tofan,baeumler,schellhorn,reif}@informatik.uni-augsburg.de

Abstract. Lock-free implementations of data structures try to better utilize the capacity of modern multi-core computers, by increasing the potential to run in parallel. The resulting high degree of possible interference makes verification of these algorithms challenging. In this paper we describe a technique to verify lock-freedom, their main liveness property. The result complements our earlier work on proving linearizability, the standard safety property of lock-free algorithms. Our approach mechanizes both, the derivation of proof obligations as well as their verification for individual algorithms. It is based on an encoding of rely-guarantee reasoning using the temporal logic framework of the interactive theorem prover KIV. By means of a slightly improved version of Michael and Scott's lock-free queue algorithm we demonstrate how the most complex parts of the proofs can be reduced to relatively simple steps of symbolic execution.

Keywords: Verification, Temporal Logic, Compositional Reasoning, Rely-Guarantee, Lock-Freedom, Linearizability.

1 Introduction

The classic approach for protecting parts of a shared data structure from concurrent access is mutual exclusion locks. One severe disadvantage of this method is that the crash or suspension of a single process can cause a deadlock or delay of the entire system. Lock-free algorithms were developed to overcome this shortcoming. One of their main features is that the crash or delay of a single process has no negative effect on the progress of other processes. This is usually achieved by applying atomic synchronization primitives such as CAS (compare and swap) or LL/SC (load linked/store conditional) and an optimistic try and retry scheme:

1. The relevant part of the shared data structure to be modified is stored in a local variable (sometimes called "snapshot").
2. Modification of the shared data structure is prepared, e.g. local fields are assigned.
3. The shared data structure is updated in one step if no interference has occurred since taking the local snapshot.

If another process has changed the snapshot during phase 2 (which is kept as small as possible), the current process must retry until no interference hinders its update.

This basic idea is extended in lots of different ways, such as by introducing reciprocal helping schemes or executing additional algorithms between the fail and the retry of an update. These techniques have resulted in lock-free implementations of various data structures, amongst others stacks [1,2], queues [3], deques [4] and hash tables [5]. Some of the proposed algorithms had subtle errors which were found when trying to formally prove their correctness [6]. The complexity of these implementations justifies the effort of formal verification and various approaches have been proposed to prove correctness [7,8,9,10,11] and liveness [12,13,14].

The main correctness criterion for lock-free algorithms is linearizability. It requires each operation to appear to take effect instantaneously at some point (the linearization point) between invocation and response, behaving according to its sequential specification [15]. This property rules out certain interleavings but does not guarantee any kind of progress. Lock-freedom is a global liveness condition which requires that at all times in a concurrent execution, one of the running operations eventually completes [16]. Consequently, as soon as no further operations are invoked, all currently active operations eventually complete. However, if the system repeatedly invokes new operations, single processes might never complete, i.e. lock-freedom does not prevent single processes from starvation.

In this paper we present a verification approach based on rely-guarantee reasoning [17,18] and interval temporal logic [19,20] and demonstrate it using a practical lock-free queue algorithm published by Doherty et al. [7], based on the original implementation of Michael and Scott [3]. The approach allows to prove two decomposition theorems: a generic refinement theorem which can be instantiated to prove linearizability and a theorem for proving lock-freedom. Both theorems have been mechanically verified using the semi-automated prover KIV [21]. The theorem for proving linearizability has been described in [11], where the resulting proof obligations have been shown to be provable for a simple stack algorithm as well as for the dequeue operation of the queue. In this work we therefore focus on describing the lock-freedom theorem and its application to the enqueue algorithm.

The main contributions are:

- A fully mechanized approach for the intuitive specification and verification of lock-free algorithms. We provide an easy to read specification language and require no program counter values for reasoning.
- An expressive temporal logic framework which allows for a simple definition of the $^+$ operator from rely-guarantee reasoning, and to prove compositionality results for parallel programs as well as refinement (= trace inclusion) theorems.
- A decomposition theorem to prove lock-freedom which does not rely on the explicit construction of well-founded orders, but on intuitive arguments of program progress.

The paper is subdivided as follows: in Section 2 we describe the queue algorithm and argue informally about its liveness. Section 3 gives a short introduction to the temporal logic framework implemented in KIV. Section 4 describes the concurrent system model and rely-guarantee reasoning. Moreover, the decomposition theorem for proving lock-freedom is introduced. Section 5 shows its application to the queue. We conclude with a section about related work (Section 6) and a summary (Section 7).

2 Michael and Scott's Lock-Free Queue

Lock-free algorithms typically use synchronization primitives such as CAS to atomically alter a shared data structure in the computer's memory. CAS can be formally specified in KIV as

$$\begin{array}{l} \mathsf{CAS}(\mathsf{Old}, \mathsf{New}; \mathsf{G}, \mathsf{Succ})\{ \\ \quad \textbf{if*}\ \mathsf{G} = \mathsf{Old}\ \textbf{then}\ \{\mathsf{G} := \mathsf{New},\ \mathsf{Succ} := \mathsf{true}\}\ \textbf{else}\ \{\mathsf{Succ} := \mathsf{false}\}\} \end{array}$$

where value-parameters Old and New are read only whereas G and Succ denote reference-parameters that can be read and modified. CAS compares a global pointer G with the (snapshot) reference stored in pointer Old. If these memory locations are equal then G is updated to a new reference New and boolean flag Succ is set to true to indicate a successful CAS. Otherwise the flag is set to false indicating that no update has occurred. Since CAS executes atomically (a comma separates parallel assignments), evaluating the if-condition should not require an extra step (denoted as if*). CAS does not guarantee that the value A of global pointer G has not been changed since it was read by a process. In the meantime, some other process might have changed G to B and then back to A. In a system that reuses freed references, these intermediate modifications can lead to subtle errors, since the content of a reallocated memory location might have been changed (ABA-problem). We assume (lock-free) garbage collection [22] and do not explicitly model memory reuse here. Any value assigned to G is going to be a newly allocated location, and this avoids an ABA-problem.

The queue is represented in memory as a singly linked list of nodes (pairs of values and references along with .val and .nxt selector functions), a global pointer Head which marks the front of the queue and a global pointer Tail indicating the end of the queue as shown in Figure 1 (a) and (b). At all times Head points to a dummy node (its value is irrelevant and denoted by a question mark). This avoids special cases for the empty queue in the implementation. There are two queue operations: the enqueue operation (CEnq) adds a node at the end of the queue; the dequeue operation (CDeq) removes the first node from the queue and returns its value. If the queue is empty, i.e. the dummy node's next reference is null, a special value empty is returned.

Attaching a new node at the end of the queue requires two global updates: the last node's next field must be set to the new node and the global tail pointer must be shifted. Since CAS allows only one atomic write access, it must be called twice. When a process encounters a lagging tail in-between these two

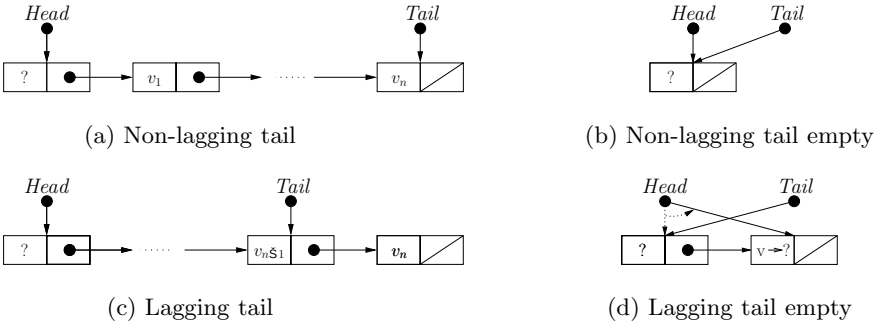

Fig. 1. Queue representation variants

CAS executions (see Figure 1 (c)), it helps by shifting the tail pointer before trying to add its new node in the next iteration.

```
E1  CEnq(v; Hp, Tail, Newe, Tle, Nxte, SuccE ) {
E2    choose Ref with Ref ≠ null ∧ ¬ Ref ∈ Hp in {
E3      Hp := Hp ∪ {Ref }, Newe := Ref , SuccE := false;
E4      Hp[Newe] := v × null;
E5      while ¬ SuccE do {
E6        Tle := Tail;
E7        Nxte := Hp[Tle].nxt;
E8        if Tle = Tail then {
E9          if Nxte = null then {
E10           CAS(Nxte, Newe; Hp[Tle].nxt, SuccE )
E11         } else {
E12           CAS(Tle, Nxte; Tail )}}}
E13     CAS(Tle, Newe; Tail )}}
```

Fig. 2. Enqueue operation

Figure 2 shows the KIV specification of the enqueue operation (line numbers are given for explanatory purposes; they are not used in KIV). In lines E2 - E4 a new node is allocated (a fresh reference is chosen and added to the global application heap Hp in one atomic step) and initialized with input value v and a null next reference (a semicolon denotes sequential composition which may be interleaved). In E6 a local snapshot is taken. Its next reference is stored locally in the following line. The test in line E8 checks whether the global tail has not been changed since the snapshot was taken. If this test fails CEnq must retry its update due to interference. The next test in E9 discerns the role of the current loop execution: if Nxte is null, line E7 was executed when the global queue was in a non-lagging tail state and the current run might successfully attach a new node at the end of the queue in line E10 and subsequently exit the loop, given that no interference has occurred in the meantime. If the test in E9 is false, the loop will be reiterated and the current process can only try to

help some other process by shifting the lagging tail pointer (line E12). The last instruction (line E13) tries to shift the tail pointer after attaching a new node to the queue. This "clean up" guarantees a non-lagging tail representation in quiescent states. CEnq uses a variant of CAS in which it is irrelevant to know whether it succeeds (lines E12 and E13). Since it is necessary to observe the values of local variables Newe, Tle, Nxte, SuccE in assertions, they have been lifted to transient parameters (see Section 5).

The formalization of the dequeue operation is shown in Figure 3. A process executing CDeq takes a snapshot of the global head pointer in line D5 and then locally stores its next reference. If the snapshot has not become obsolete and the local next reference is null, dequeue returns empty. If the queue is not empty CAS is applied in line D12 to shift the global head pointer, making Nxtd the new dummy node. The remaining lines of code (D13-D16) then deal with a special configuration which emerges from shifting Head when the queue contains exactly one value v and the tail pointer is lagging (see Figure 1 (d)). Since the head pointer gets shifted ahead of the tail pointer, dequeue can help the process which has enqueued v (line D16).[1]

```
D1 CDeq(; Hp, Head, Tail, Hdd, Nxtd, SuccD, O) {
D2   let Lo = empty, Tld = null in {
D3     SuccD := false;
D4     while ¬ SuccD do {
D5       Hdd := Head;
D6       Nxtd := Hp[Hdd].nxt;
D7       if Hdd = Head then {
D8         if Nxtd = null then {
D9           Lo := empty; SuccD := true
D10        } else {
D11          Lo := Hp[Nxtd].val;
D12          CAS(Hdd, Nxtd; Head, SuccD);
D13          if SuccD then {
D14            Tld := Tail;
D15            if Tld = Hdd then {
D16              CAS(Tld, Nxtd; Tail)}}}}}
D17    O := Lo}}
```

Fig. 3. Dequeue operation

The intuitive reason why the implementation is lock-free is that its loops are retried if some other process changes the queue in the critical time slot between taking a snapshot and trying to update the data structure. This change however implies that the interfering process eventually completes. In the formal proof we will reflect the simplicity of this intuitive argument by using an additional predicate U ("unchanged") which describes the absence of interference that can

[1] The original dequeue implementation of Michael and Scott reads the shared tail pointer whenever the loop-body is executed. The implementation given here reduces shared memory access if the loop has to be executed several times.

cause a process to retry its loop infinitely often. Proving lock-freedom then requires to show that each data structure operation eventually terminates when it either encounters no such interference or when it changes the shared state itself. For the dequeue operation this argument is simple. If it encounters no interference, then the CAS at D13 will be successful and dequeue terminates. A successful CAS at D13 is also the only place, where the process itself changes the shared data structure.

For the enqueue operation the argument is slightly more subtle. If the queue is not modified after taking the snapshot in E6, its loop might be executed once again before enqueue terminates, due to a lagging tail. But this does not hinder termination, so we do not count shifting a lagging tail as an interference (predicate U is true for that case). Finally, if enqueue changes the queue by adding a new node in E10 it terminates without further iterations.

As we will see, this intuitive reasoning is formally performed in KIV, by applying symbolic execution to step forward through each line of code of an operation.

3 Temporal Logic in KIV

This section briefly describes the temporal logic calculus integrated into the interactive theorem prover KIV. A more detailed description can be found in [23,24].

3.1 Interval Temporal Logic

The basis of interval temporal logic (ITL) [19,20] are algebras (to interpret the signature) and intervals, i.e. finite or infinite sequences of states (each mapping variable symbols to values in the algebra). Intervals typically evolve from program execution. In contrast to standard ITL, the formalism used here explicitly includes the behavior of the program's environment into each step: in an interval $I = [I(0), I'(0), I(1), I'(1), \ldots]$ the first program transition leads from the initial state $I(0)$ to the primed state $I'(0)$ whereas the next transition (from state $I'(0)$ to $I(1)$) is a transition of the program's environment. In this manner program and environment transitions alternate (similar to [25,26]).

Variables are partitioned into static variables v (written lower case), which never change their value ($I(0)(v) = I'(0)(v) = I(1)(v) = \ldots$) and flexible variables V (starting with an uppercase letter) which can have different values in different states of an interval. We write V, V', V'' to denote variable V in states $I(0)$, $I'(0)$ and $I(1)$ respectively. In the last state (characterized by the atomic formula last) of an interval, the value of a primed or double primed variable is equal to the value of the unprimed variable, i.e. after a program has terminated its variables do not change by convention.

The logic uses standard temporal operators (, , €, until, unless, ...) as well as sequential programming constructs (:=, ;, if, ...). We usually write , to indicate a program and , to indicate a formula. is weak fair interleaving, await blocks execution until the test is satisfied (not used in

the algorithms here). Programs and formulas can be mixed arbitrarily since they both evaluate to true or false over an algebra A and an interval I (hence system descriptions can be abstracted by temporal properties). A program evaluates to true (A, I ⊨) if I is a possible run of the program. Note that a run of a program is always interleaved by arbitrary transitions of its environment. More details on the syntax and semantics of these operators can be found in [23].

3.2 Symbolic Execution and Induction

KIV is based on the sequent calculus. Sequents are assertions of the form where and are sets of formulas. A sequent states that the conjunction of all formulas in antecedent implies the disjunction of all formulas in succedent . Sequents are implicitly universally closed. A typical sequent (proof obligation) about interleaved programs has the form

, E, I ⊢

where an interleaved program executes the system steps; the system's environment behavior is constrained by temporal formula E; I is a predicate logic formula that describes the current state and is the property which has to be shown. To verify that holds, symbolic execution is used. For example, a sequent of the form mentioned above might be

(M := M + 1;), M = M , M = 1 ⊢ M > 0

The program executed is M := M +1; where is an arbitrary program and the environment is assumed never to change counter M (formula M = M). The current state maps M to 1. The intuitive idea of a symbolic execution step is to execute the first program statement, i.e. to apply the changes on the current state and to discard the first statement. In the example above, a symbolic execution step leads to a trivial predicate logic goal for the initial state (M = 1 M > 0) and a sequent that describes the remaining interval from the second state on:

, M = M , M = 2 ⊢ M > 0

M has value 2 in the new state which follows from the fact that after M has been set to two by the program transition, the environment leaves M unchanged. Otherwise M would have an arbitrary value in the new state. Symbolic execution concerns both programs and formulas and has two phases. In the first phase information about the first transition (both system and environment) is separated from information about the rest of the run. We get M = M + 1 M = M from the assignment and the environment assumption for the first step and € for the rest of the run. For M > 0 we get M > 0 and € M > 0 using the unwinding rule € . In the second phase of a symbolic execution step, the unprimed and primed variables M and M are substituted with fresh static variables that describe the former state, whereas the double primed variable M is replaced with the unprimed variable M in the new state, and leading next operators (€) are dropped.

In addition to symbolic execution, well-founded induction is used to deal with loops. For finite intervals it is possible to induce over the length of an interval. For infinite traces a well-founded order can often be derived from liveness properties by inducing over the number of steps N until holds:

$$\leftrightarrow\ \exists\, \mathsf{N}.\ (\mathsf{N} = \mathsf{N}\ \ + 1)\ \textbf{until}$$

The equivalence states that is eventually true, if and only if N can be decremented (note that N = N + 1 is equivalent to N > 0 N = N Š 1) until becomes true. Proving a formula of the form on infinite traces is then simply done by rewriting to ¬ ¬ and a proof by contradiction. Similarly, an unless formula (as needed later in rely-guarantee proofs, cf. Section 5.1) can be reduced to the case of an eventually formula using the equivalence

$$\textbf{unless}\quad \leftrightarrow \forall\, \mathsf{B}.\ (\quad \mathsf{B}) \rightarrow (\quad \textbf{unless}\ (\quad \wedge\, \mathsf{B}\ \vee\quad))$$

unless is true if it is true on every prefix of the trace that is terminated by the first time when boolean variable B becomes true. This rewriting allows for extracting the liveness property B to prove that the initial unless formula holds, by applying well-founded induction over the number of steps until B is true. The (semantic) proofs of both equivalences above are straightforward.

4 Rely-Guarantee Reasoning and the Decomposition Theorem for Lock-Freedom

This section gives a short introduction to the concurrent system model in our approach and to the well-known decomposition technique of rely-guarantee reasoning. Furthermore, we describe a decomposition theorem for proving lock-freedom. Its formal proof is available online [27].

4.1 System Model and Rely-Guarantee Reasoning

A concurrent system is a program which spawns an arbitrary positive number of processes to execute in parallel

```
CSpawn(n; Act, In, CS, Out) {
  if* n = 0 then
    CSeq(n; Act, In, CS, Out)
  else
      CSeq(n; Act, In, CS, Out)
      CSpawn(n − 1; Act, In, CS, Out)}
```

```
CSeq(m; Act, In, CS, Out) {
  {   skip
    ∨ {Act(m) := true;
       COP(m, In; CS, Out);
       Act(m) := false}
  }*}
```

CSpawn consists of n + 1 processes that execute CSeq in parallel. Operation CSeq finitely or infinitely often (denoted by *) does some computations that have no direct influence on the underlying data structure (modeled as no operation skip) or it executes an arbitrary data structure operation COP (in the

queue example, COP is simply the nondeterministic choice ( ) between one of the two operations CEnq and CDeq).

Operation CSeq is called with a value parameter m of type nat which represents the identifier of the invoking process. Reference-parameter Act : nat bool is a boolean function which is used to distinguish whether a process is currently active in the sense of currently executing COP (this activity flag is only relevant for proving lock-freedom). Function In : nat input is used to pass an arbitrary input value In(m) to COP. In is a reference parameter in CSeq whereas it is a value parameter in COP, i.e. whenever COP is invoked, its input value can differ from previous invocations due to changes on In by CSeq's environment (this ensures that different values can be enqueued). The remaining parameters include a generic state variable CS : cstate for the (shared and local) state on which COP works and an output function Out : nat output to return values.

Rely-guarantee reasoning is a widely used decomposition technique to prove properties of an overall concurrent system by looking at the system's components only [17,18]. To this end each process (component) m is extended with two predicates: a two-state rely predicate R_m : cstate× cstate describing the behavior of m's environment (including other processes within the system plus the environment of the entire system) and a binary guarantee predicate G_m : cstate× cstate which describes the impact of m on its environment (the first parameter of a guarantee/rely condition denotes the state before the system/environment step and the second argument denotes the next state). To ensure correctness each guarantee condition must preserve the rely conditions of all other processes

$$m \neq n \wedge G_m(CS_0, CS_1) \rightarrow R_n(CS_0, CS_1) \tag{1}$$

The intuitive idea of the rely-guarantee approach is to claim that every process m fulfills its guarantee G_m if every other process does not violate its rely condition R_m. To break circularity of this argument, a special implication operator $^+$ (as defined in [28]) is used which states that m fulfills its guarantee if its rely condition has not been violated in some preceding step ($R_m \; ^+ \; G_m$). The explicit separation between program and environment transitions in our logic enables us to specify guarantees as predicates $G_m(CS, CS')$ with unprimed and primed variables describing steps of process m. Rely conditions $R_m(CS', CS'')$ instead use primed and double primed variables to restrict steps of m's environment. The formal definition of $^+$ is then simply based on the temporal operator unless

$$R_m \; ^+ \; G_m :\equiv G_m(CS, CS') \; \mathbf{unless} \; (G_m(CS, CS') \wedge \neg R_m(CS', CS''))$$

Since unless ( ) (until), either the guarantee G_m always holds or it holds until a system step occurs in which the guarantee still holds, but where the subsequent environment transition violates m's rely condition.

In order to show that a process m which executes CSeq satisfies $R_m \; ^+ \; G_m$, two properties must be fulfilled. First, each guarantee must be reflexive (in case of skip or a step that sets the activity flag, the current state stays the same)

$$G_m(CS, CS) \tag{2}$$

Second, $R_m^+ \; G_m$ must be preserved by the data structure operation

$$COP(m, In; CS, Out), Inv(CS) \vdash R_m^+ \; G_m \tag{3}$$

where predicate $Inv : cstate$ introduces an invariant. Properties (2) and (3) also imply that every process m preserves its guarantee condition at all times, in an environment that always respects m's rely condition. To show that in this case m always preserves the invariant too, we stipulate stability of the invariant over rely steps:

$$Inv(CS') \wedge R_m(CS', CS'') \rightarrow Inv(CS'') \tag{4}$$

With (1) it follows that Inv is also stable over each local guarantee (note that (1) holds for arbitrary distinct natural numbers) and specifies indeed an invariant property

$$CSeq(m; \ldots), \quad R_m(CS', CS''), Inv(CS) \vdash \quad (Inv(CS) \wedge Inv(CS''))$$

To lift this property (resp. (3)) to the level of an interleaved execution of the overall system $CSpawn$, it is necessary to be able to summarize several consecutive local rely steps in one rely step, i.e. we require R_m to be transitive

$$R_m(CS_0, CS_1) \wedge R_m(CS_1, CS_2) \rightarrow R_m(CS_0, CS_2) \tag{5}$$

Since the generic setting also takes into account the environment of the overall system, a global rely condition $R : cstate \times cstate$ is required too. It preserves each local rely condition

$$R(CS', CS'') \rightarrow R_m(CS', CS'') \tag{6}$$

Conditions (1) to (6) are the same as described in [11] for linearizability. The few extensions required to prove lock-freedom are introduced in the next section. As several of the following proof obligations will assume an invariant and a rely condition to always hold, we define the following abbreviation:

$$I(R) :\equiv Inv(CS) \wedge Inv(CS') \wedge R(CS', CS'')$$

4.2 Decomposition Theorem for Lock-Freedom

Lock-freedom is a global progress property of a concurrent system which states that at all times throughout an (infinite) execution of the system, eventually one process completes its currently running operation [16]. There are two further important liveness properties [29]: wait-freedom requires each invoked operation to eventually complete (thus it is stronger than lock-freedom); obstruction-freedom requires completion of every operation that eventually executes in isolation (hence it is a weaker property than lock-freedom). In contrast to lock-freedom, proofs of these properties require no decomposition technique, since they are already process-local. All three properties preclude the standstill (deadlock) of the

system but in a lock-free implementation, repeated change of the data structure can force a single process to retry again and again.

In our formal setting (see Section 4.1) - apart from executing infinitely often COP - processes may also execute skip or terminate. Therefore an additional activity flag is required to detect termination of the data structure operation. A process m finishes its current execution of an operation when it resets its activity flag $\mathsf{Act}(\mathsf{m})$. In a concurrent system which consists of n processes, global progress P is defined in terms of the activity flags as

$$\begin{aligned} &\mathsf{P}(\mathsf{n}, \mathsf{Act}, \mathsf{Act}') \\ \leftrightarrow\ &((\exists\, \mathsf{m} \leq \mathsf{n}.\ \mathsf{Act}(\mathsf{m})) \rightarrow \quad (\exists\, \mathsf{k} \leq \mathsf{n}.\ \mathsf{Act}(\mathsf{k}) \wedge \neg\, \mathsf{Act}'(\mathsf{k}))) \end{aligned}$$

That is, if there is at least one active process (m), one of them (k) will eventually reset its activity flag, i.e. complete its operation on the data structure.

To model the absence of interference that forces a process to reiterate, an additional predicate $\mathsf{U} : \mathsf{cstate} \times \mathsf{cstate}$ ("unchanged") is added to the rely-guarantee theory. This predicate must be reflexive, because steps that leave the state unchanged do not interfere with other processes. It is also necessary (for the lifting) to be able to summarize several consecutive steps which satisfy U into one step by transitivity

$$\begin{aligned} &\mathsf{U}(\mathsf{CS}, \mathsf{CS}) \\ &\mathsf{U}(\mathsf{CS}_0, \mathsf{CS}_1) \wedge \mathsf{U}(\mathsf{CS}_1, \mathsf{CS}_2) \rightarrow \mathsf{U}(\mathsf{CS}_0, \mathsf{CS}_2) \end{aligned} \tag{7}$$

Furthermore, we exclude steps from the system's environment which unpredictably change the activity flags or the critical parts of the data structure by extending the global rely condition:

$$\begin{aligned} &\mathsf{R}_{\mathsf{ext}}(\mathsf{CS}, \mathsf{Act}, \mathsf{CS}', \mathsf{Act}') \\ \leftrightarrow\ &\mathsf{R}(\mathsf{CS}, \mathsf{CS}') \wedge \mathsf{Act}' = \mathsf{Act} \wedge \mathsf{U}(\mathsf{CS}, \mathsf{CS}') \end{aligned}$$

This extension is acceptable, since we assume that only processes within the overall interleaved system are allowed to manipulate these specific resources. Lock-freedom of CSpawn then follows from the following intuitive local proof obligation

$$\begin{aligned} &\mathsf{COP}(\mathsf{m}, \mathsf{In}; \mathsf{CS}, \mathsf{Out}), \quad \mathsf{I}(\mathsf{R}_\mathsf{m}) \\ \vdash\ &(\neg\, \mathsf{U}(\mathsf{CS}, \mathsf{CS}') \vee (\quad \mathsf{U}(\mathsf{CS}, \mathsf{CS}')) \rightarrow \quad \mathbf{last}) \end{aligned} \tag{8}$$

At any time (leading $\quad$), a lock-free operation that updates the relevant part of the shared state itself in a step ($\neg\, \mathsf{U}(\mathsf{CS}, \mathsf{CS}')$) or encounters no interference ($\quad \mathsf{U}(\mathsf{CS}, \mathsf{CS}')$), eventually terminates ($\quad \mathsf{last}$).

Properties (7) and (8) together with the rely-guarantee conditions of the previous subsection are sufficient to prove lock-freedom of the overall system, when initially the invariant holds and all activity flags are false.

Theorem 1 (Decomposition Theorem for Lock-Freedom)
If formulas (1) to (8) can be proved (for some $\mathsf{Inv}, \mathsf{U}, \mathsf{R}, \mathsf{R}_\mathsf{m}, \mathsf{G}_\mathsf{m}$), then:

$$\mathsf{CSpawn}(\mathsf{n}; \ldots), \quad \mathsf{R}_{\mathsf{ext}}, \mathsf{Inv}(\mathsf{CS}), \forall\, \mathsf{m} \leq \mathsf{n}.\ \neg\, \mathsf{Act}(\mathsf{m}) \vdash \quad \mathsf{P}(\mathsf{n}, \mathsf{Act}, \mathsf{Act}') \tag{9}$$

Given that the global environment satisfies R_{ext} at all times, the presence of an active operation will always lead to the completion of some (active) operation. Although there are no blocking steps in the queue example, the theorem holds for algorithms COP which include such steps too.

The theorem is proved in two stages. The first stage proves

$$\begin{array}{l} \mathsf{CSeq}(m;\ldots),\quad \mathsf{I}(R_m), \neg\,\mathsf{Act}(m) \\ \vdash\quad (\quad \mathsf{Act}(m) \wedge (\neg\,\mathsf{U}(CS, CS\,) \vee (\quad \mathsf{U}(CS\,, CS\,))) \\ \qquad \rightarrow\quad (\mathsf{Act}(m) \wedge \neg\,\mathsf{Act}\,(m))) \end{array} \tag{10}$$

while the second proves the main theorem. Both proofs rely on the fact, that our logic allows to reduce a goal (resp. ;) to (resp. ;), when a lemma is available (see [23] for more details). Note that for interleaving this fact crucially depends on our semantics with alternating system and environment steps. It does not hold in standard temporal logic.

A detailed description of the proofs is beyond the scope of this paper, we just give the main idea of the second proof. The proof of (9), which can be written in the form $\mathsf{CSpawn}(n;\ldots)$ (n), starts by induction over the number of processes. Lemma (10), which can be written as CSeq , directly closes the base case. In the induction step, unfolding of $\mathsf{CSpawn}(n+1;\ldots)$ gives an interleaving of CSeq and $\mathsf{CSpawn}(n;\ldots)$. The first formula in the interleaving can be replaced with , while the second can be replaced with (n) by the induction hypothesis. Therefore it remains to prove (n) $(n+1)$. The main part of the proof is now by induction over $\mathsf{P}(n+1, \mathsf{Act}, \mathsf{Act}\,)$ in $(n+1)$ and symbolic execution. The proof has a large number of cases, since a symbolic execution step of each of the two formulas and (n) can terminate (causing the other formula to remain), or do an unblocked or blocked step (the latter forcing a step of the other formula, or a blocked step if both block). Also in each symbolic execution step we have to prove the implication of the progress property for the current state. The proof is more complex than all the proofs of the case study. Since it has to be done once only, it moves much of the complexity of analyzing the lock-freedom property into the generic theory. The proof has been mechanized using KIV and is online [27].

5 Proving Lock-Freedom for the Queue

In this section we present the instantiation of the decomposition theorem for the queue. The presentation is in two parts: first we give the necessary rely and invariant conditions which are a subset of those used for proving linearizability in [11]; second we describe the instantiation of the unchanged predicate and outline the proof of termination for the enqueue operation. Full details are available online [27].

5.1 Rely-Guarantee Conditions and Invariant

The generic operation COP is instantiated with the nondeterministic choice between the two queue operations. The generic state variable CS becomes a tuple

consisting of a shared state Hp, Head, Tail and local states $\mathsf{Newef}(\mathsf{m})$, $\mathsf{Tlef}(\mathsf{m})$, $\mathsf{Nxtef}(\mathsf{m})$, $\mathsf{Succef}(\mathsf{m})$, $\mathsf{Hddf}(\mathsf{m})$, $\mathsf{Nxtdf}(\mathsf{m})$, $\mathsf{Succdf}(\mathsf{m})$ for every process m.

Since all processes execute the same set of operations, all processes will have the same rely condition R_m by symmetry. It claims that the environment step preserves the invariant Inv and that predicates $\mathsf{Enqlocal}_m$ and $\mathsf{Deqlocal}_m$ hold.

$$\begin{aligned} & R_m(CS, CS') \\ \leftrightarrow\ & (\mathsf{Inv}(CS) \rightarrow \mathsf{Inv}(CS')) \wedge \mathsf{Enqlocal}_m(CS, CS') \wedge \mathsf{Deqlocal}_m(CS, CS') \end{aligned}$$

The invariant ensures that there are no dangling pointers and that newly allocated nodes are disjoint from one another and from the queue. It also guarantees that the current state is a valid queue representation, i.e. it conforms to one of the variants shown in Figure 1.

Predicate $\mathsf{Enqlocal}_m$ specifies that pointer variables $\mathsf{Succef}(\mathsf{m})$, $\mathsf{Tlef}(\mathsf{m})$ and $\mathsf{Nxtef}(\mathsf{m})$ (which were lifted from originally local variables to global ones) are unchanged by other processes

$$\begin{aligned} & \mathsf{Succef}'(\mathsf{m}) = \mathsf{Succef}(\mathsf{m}) \wedge \mathsf{Tlef}'(\mathsf{m}) = \mathsf{Tlef}(\mathsf{m}) \\ \wedge\ & \mathsf{Nxtef}'(\mathsf{m}) = \mathsf{Nxtef}(\mathsf{m}) \end{aligned} \tag{11}$$

The main interesting information necessary to prove lock-freedom is that whenever the snapshot's next pointer is not null, this reference remains untouched by m's environment:

$$\begin{aligned} & \mathsf{Tlef}(\mathsf{m}) \neq \mathsf{null} \wedge \mathsf{Hp}[\mathsf{Tlef}(\mathsf{m})].\mathrm{nxt} \neq \mathsf{null} \\ \rightarrow\ & \mathsf{Hp}'[\mathsf{Tlef}(\mathsf{m})].\mathrm{nxt} = \mathsf{Hp}[\mathsf{Tlef}(\mathsf{m})].\mathrm{nxt} \end{aligned} \tag{12}$$

This rely condition is interesting when a process shifts a lagging tail pointer for two reasons: first, to argue that this step maintains a valid queue representation and second, to ensure that it does not violate the unchanged predicate (cf. predicate Id_S in the next subsection). Proof obligation (3) from the rely-guarantee theory implies that this assumption is acceptable. Its proof rewrites the unless formula in the succedent as described in Section 3.2 to extract an inductive argument in case that a loop is reiterated. When symbolically executing the code of a queue operation of an arbitrary process m, it has to be shown that each step preserves m's guarantee condition G_m, given that m's local rely condition was true for the last environment transition. The local guarantee condition G_m (and the global rely condition R) is defined as weak as possible by constraint (1) (resp. (6)) of the rely-guarantee theory. Since proving (3) has also been necessary in our previous work to show that the queue algorithm is linearizable and since the required rely-guarantee conditions from the linearizability-proof were sufficient to prove lock-freedom too, the former proof of (3) has been reused.

Altogether the required rely conditions for lock-freedom of enqueue are a valid queue representation, (11), and (12). Similar assumptions as defined in $\mathsf{Enqlocal}_m$ are defined in in $\mathsf{Deqlocal}_m$ for the linearizability poof of the dequeuing process. However, for proving lock-freedom of dequeue, only the locality of $\mathsf{Succdf}(\mathsf{m})$ and $\mathsf{Hddf}(\mathsf{m})$ are required.

5.2 Unchanged Predicate

According to proof obligation (8) a suitable instantiation of predicate U must ensure termination of a process in an environment that respects U at all times and it must be preserved by each program transition, unless a transition eventually leads to completion (e.g. a successful CAS).

That is, when a process dequeues it is sufficient for its termination to assume that the global head pointer remains unchanged by the environment

$$\mathsf{Id_H} :\equiv \mathsf{Head} \quad = \mathsf{Head}$$

When m enqueues, assuming that other processes n will not change the global tail pointer is not sufficient to ensure termination. Suppose a system execution in which m repeatedly shifts the lagging tail for every n which attaches a new node to the queue. In this situation, no other process ever changes the tail pointer, as this is done by m who never completes. Instead, U must ensure that m finally can attach its newly allocated node to the queue, i.e. no other process may add a new node. Two cases are discerned regarding the current representation. If the tail pointer does not lag (its next reference is null) neither the global tail pointer nor its next reference may be changed

$$\mathsf{Id_T} :\equiv \mathsf{Tail} \quad = \mathsf{Tail} \quad \wedge \mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt} = \mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt}$$

When the tail pointer is lagging, m assumes the following environment behavior: other processes leave the tail pointer and its next reference unchanged or they shift the tail to its direct successor node (which has a null next reference)

$$\mathsf{Id_S} :\equiv \mathsf{Id_T} \vee \mathsf{Tail} \quad = \mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt} \wedge \mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt} = \mathsf{null}$$

Predicate U is the conjunction of these identities:

$$\mathsf{Id_H} \ \wedge (\mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt} = \mathsf{null} \rightarrow \mathsf{Id_T}) \wedge (\mathsf{Hp} \ [\mathsf{Tail} \ ].\mathrm{nxt} \neq \mathsf{null} \rightarrow \mathsf{Id_S})$$

It specifies that changes relevant for progress are enqueuing or removing an element, while moving a lagging tail does not guarantee progress and can only be done according to Figure 1.

5.3 Proof Outline

The unchanged predicate is reflexive and transitive. The temporal logic proof obligation (8) from Section 4.2 is divided into four subgoals by discerning which operation is currently executed (enqueue or dequeue) and splitting the disjunction in the succedent to distinguish whether a local transition of the current process changes the data structure or the environment satisfies the unchanged property at all times.[2] For enqueue we get two proof obligations

$$\begin{array}{ll} \mathbf{E1}, & \mathsf{I}(\mathsf{R_m}) \vdash \quad (\neg \, \mathsf{U}(\mathsf{CS}, \mathsf{CS}\,) \rightarrow \quad \mathbf{last}) \\ \mathbf{E1}, & \mathsf{I}(\mathsf{R_m}) \vdash \quad (\quad \mathsf{U}(\mathsf{CS}\,, \mathsf{CS}\,) \rightarrow \quad \mathbf{last}) \end{array} \tag{13}$$

[2] As the interleaving operator is not used in proof obligation (8), its proof is independent from the underlying scheduler. Scheduling issues are covered in the lifting proof of the decomposition theorem only.

$$\dfrac{\dfrac{\dfrac{\mathbf{E8}\ldots, S_1 \vdash \ldots}{\cdots \neq \mathsf{null}, \mathsf{Tail} = \mathsf{Tlef}(m) \vdash \ldots}\,(4) \qquad \cdots = \mathsf{null} \vdash \ldots}{\dfrac{\mathbf{E7}, \ldots, S_0 \vdash \ldots}{\ldots \mathsf{Hp}[\mathsf{Tail}].\mathrm{nxt} \neq \mathsf{null} \vdash \ldots}\,(2)}\,(3) \qquad \cdots = \mathsf{null} \vdash \ldots}{\mathbf{E6},\ \mathsf{VRU} \vdash \quad \mathbf{last}}\,(1)$$

$$\begin{aligned} \mathsf{VRU} &:\equiv \quad (\mathsf{valid}(\mathsf{Head}, \mathsf{Tail}, \mathsf{Hp}) \wedge (11) \wedge (12) \wedge \mathsf{U}(\mathsf{CS}, \mathsf{CS}')) \\ S_0 &:\equiv \mathsf{Tlef}(m) \neq \mathsf{null} \\ S_1 &:\equiv \mathsf{Tlef}(m) \neq \mathsf{null} \wedge \mathsf{Nxtef}(m) \neq \mathsf{null} \wedge \mathsf{Hp}[\mathsf{Tlef}(m)].\mathrm{nxt} = \mathsf{Nxtef}(m) \end{aligned}$$

Fig. 4. Proof outline enqueue lock-free

where Ek denotes the remaining program starting from line Ek , e.g. and $\mathsf{E1}$ CEnq and $\mathsf{E12}$ $\mathsf{CAS}(\mathsf{Tle}, \mathsf{Nxte}; \mathsf{Tail}); \mathsf{while}\ \neg\ \mathsf{SuccE}\ \mathsf{do}\ \ldots$ (these abbreviations are not used in KIV). The first is rather simple, since the only step with $\neg\ \mathsf{U}(\mathsf{CS}, \mathsf{CS}')$ is a succeeding CAS at line E10 which sets the loop-flag to true, so the algorithm terminates after the final step E13.

The second proof is more challenging. It consists of an induction for the leading always operator and symbolically executing the enqueue operation until it either terminates or the induction hypothesis can be applied. During execution we get a side goal for every step: starting from the considered step, formula $\mathsf{U}(\mathsf{CS}, \mathsf{CS}')$ must lead to termination. This can be proved by stepping to the start of the loop (instruction E5) and applying the following lemma

$$\mathbf{E5}, \quad \mathsf{I}(R_m) \vdash \quad \mathsf{U}(\mathsf{CS}, \mathsf{CS}') \rightarrow \quad \mathbf{last}$$

which states that the additional environment assumption $\mathsf{U}(\mathsf{CS}, \mathsf{CS}')$ is sufficient to guarantee termination of the loop of the enqueue operation.

Its proof needs no induction, but requires stepping through the loop once or twice, depending on whether the tail is lagging when the snapshot is taken; the basic idea is illustrated in Figure 4. In the conclusion of the proof tree, the first symbolic execution step to enter the while loop has already been executed. The remaining program is $\mathsf{E6}$ (instruction E6 takes the local snapshot $\mathsf{Tlef}(m)$). In a valid state, the required rely conditions and the unchanged predicate are assumed to hold at all times (VRU); no further restrictions on the current state are necessary to prove termination of the loop. Proof step (1) is a case distinction on whether the current queue has a lagging tail pointer ($\mathsf{Hp}[\mathsf{Tail}].\mathrm{nxt} = \mathsf{null}$). If the tail pointer is not lagging (second premise, right hand side) no further interference will hinder m to complete according to VRU, i.e. the proof consists of executing $\mathsf{E6}$ until completion. If the tail pointer is lagging behind (first premise, left hand side), proof step (2) symbolically executes the instruction at E6 (followed by an environment transition) which yields the new state S_0 and the remaining program is $\mathsf{E7}$. Case distinction (3) tests whether the environment has helped m according to predicate U by shifting the lagging tail pointer (second premise). If this is true, the current proof obligation can be discarded by symbolic execution until the remaining program is again $\mathsf{E6}$ and using the second premise

of proof step (1) as a lemma (during these symbolic execution steps - the test at E8 is false - the tail pointer and its next reference null remain unchanged). If however the tail is still lagging (first premise of proof step (3)) the snapshot is accurate, i.e. $\mathsf{Tail} = \mathsf{Tlef}(\mathsf{m})$, and the proof continues with symbolic execution of E7 (proof step (4)). In the new state S_1, the snapshot's next reference is $\mathsf{Nxtef}(\mathsf{m})$ which is not null. We proceed analogously discerning whether the tail pointer is lagging and symbolic execution: at the latest when the CAS transition at E12 is (successfully) executed, a non-lagging tail representation is established and the second premise of step (1) can eventually be used again as a lemma to finish the proof.

Proving the analog properties to (13) for dequeue is straightforward. The locality assumptions (for the loop-flag and the snapshot) from the rely condition and knowing that the head pointer always remains unchanged according to U, imply termination. This is because after the snapshot is taken, the CAS at D12 will be successfully executed: it is the only dequeue step that does not satisfy the unchanged predicate, but it guarantees progress.

6 Related Work

The analysis of non-blocking algorithms is a current and highly active field of research. Several techniques have been proposed to prove correctness and liveness of these algorithms.

With respect to linearizability, Doherty et al. [7] were the first to publish a formal verification of the queue algorithm (including memory reuse and version numbers to avoid an ABA-problem) based on refinement of IO automata. In contrast to our approach, program counters and a global simulation relation are used to mechanize the proofs using PVS. Since single steps of a concrete algorithm are refined individually, an intermediate automaton and backward simulation had to be used to complete the formal proof for the dequeue operation, while our approach verifies trace inclusion directly avoiding backward simulation (see [11] for details).

Vafeiadis [30] also proves linearizability of the queue. His proof technique is closer to ours in also using rely-guarantee reasoning. A major difference is that his approach is based on adding abstract ghost code to the implementation, and not on refinement. To solve the problem of the dequeue operation, the use of a prophecy variable is suggested (which is basically equivalent to the use of backward simulation).

Many other groups have contributed to the verification of non-blocking algorithms. Groves et al. [8] for instance present the verification of linearizability of a more complex lock-free implementation based on trace reduction. Our approach is currently not able to formally handle these kind of (elimination) algorithms, where the linearization of an operation can be part of the execution of another process. Gao et al. [31] have described the verification of a lock-free hash table which took more than two man years of work.

A rather different approach is taken by Yahav et al. [32] using shape analysis [33]. The approach assumes that the abstract operations - although atomic - already work on the low level heap and that only their interleaving has to be shown correct. Therefore it compares the intermediate heaps that occur during interleaved execution of the algorithms to the structures at the beginning and the end and keeps track of the differences by a finite abstraction ("delta heap abstraction") to verify linearizability.

The third author has also contributed to Derrick et al. [34]. The approach given there is rather different: it is based on the Z specification language and requires program counters to encode steps of the algorithm as Z operations. Instead of rely-guarantee reasoning, Owicki-Gries [35] like proof obligations are generated. The approach is the only one we are aware of, that proves linearizability formally using the original definition of [15]. All other approaches (including ours in [11]) argue informally that linearizability holds.

Related to lock-freedom, we are aware only of two approaches: Colvin and Dongol [12,13] describe the verification of several lock-free implementations (including an array-based nonblocking queue [36]) by explicitly constructing a well-founded order on program counters and proving that each action either guarantees progress or reduces the value of the state according to the well-founded order. They identify progress actions, which correspond to those steps where our predicate U is false. Constructing a well-founded order is unnecessary in our approach, since it is implicit in stepping through the program.

A higher degree of automation is achieved by Gotsman et al. [14] based on rely-guarantee reasoning and techniques like shape analysis and separation logic [37]. Their approach can verify proof obligations that imply lock-freedom for several non-trivial algorithms automatically, using a combination of several tools. Derivation of these proof obligations however is done on paper. There are several differences in the proof obligations too: our approach does not use a reduction of CSpawn to a spawning procedure where the call to CSeq is replaced by COP (which needs some assumptions about symmetry to be correct). Our proof obligation ensures that the algorithm terminates after a step which falsifies U, while their proof obligation requires that no process can execute steps which change the data structure infinitely often. A close comparison for the queue example is hard, since the queue is only mentioned as one of the examples automatically provable.

Both related approaches assume potentially unfair scheduling, which is more adequate than our assumption of weak fairness. A closer analysis shows that we need fairness only to prove that a process is not suspended in favor of another process which executes skip steps only. Both related approaches consider processes which execute an infinite loop of calls to COP and no other instructions. If we replace the implementation of CSeq with such a loop, the fair interleaving operator can be replaced with an unfair one. We prefer the more general formalization of CSeq, since it is realistic that a process executes other statements or terminates rather than just calling COP repeatedly. Nevertheless we have mechanized a version for this loop with unfair interleaving too. For simplicity, the current proof is limited to algorithms without blocking steps.

The proof proceeds much like the original one, since the symbolic execution rules for non-fair interleaving are the same as for fair interleaving. The main difference is that without weak fairness, it can no longer be guaranteed that the first of two interleaved processes will do a step eventually. Instead, an additional case split is necessary which gives the same goal as for weak fairness, plus an extra goal for the case where the first process is never scheduled again, so only the second remains. This proof is available online [27] too.

7 Summary

We have described a decomposition theorem that reduces the proof of the global property lock-freedom to process-local proof obligations and we have shown how this theorem can be applied to prove lock-freedom of a non-trivial lock-free queue implementation. All specifications and proofs are fully mechanized in the interactive theorem prover KIV and the main proofs of lock-freedom in the queue case study are highly automated. The theory shares rely-guarantee conditions with those necessary to prove linearizability. We believe that our technique closely follows the intuitive arguments necessary to prove lock-freedom.

In future work we will consider the ABA-problem in an additional refinement step (similar to [8]), by extending the current implementation with reference-recycling and version numbers. Moreover, we will try to improve our method by better exploiting the symmetry of typical lock-free implementations in the rely-guarantee theory and by including a formal definition of linearizability within the reduction approach (similar to [34]).

Acknowledgements

We want to thank the reviewers for their constructive remarks, in particular on the weak fairness assumption, which have helped to improve this paper.

References

1. Treiber, R.K.: System programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center (1986)
2. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA '04: ACM symposium on Parallelism in algorithms and architectures, pp. 206–215. ACM Press, New York (2004)
3. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proc. 15th ACM Symp. on Principles of Distributed Computing, pp. 267–275 (1996)
4. Michael, M.M.: Cas-based lock-free algorithm for shared deques. In: Kosch, H., Böszörményi, L., Hellwagner, H. (eds.) Euro-Par 2003. LNCS, vol. 2790, pp. 651–660. Springer, Heidelberg (2003)
5. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA 2002, pp. 73–82. ACM, New York (2002)

6. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Steele Jr., G.L.: Dcas is not a silver bullet for nonblocking algorithm design. In: SPAA '04: Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures, pp. 216–224. ACM, New York (2004)
7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
8. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) 21(1-2), 187–223 (2009)
9. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, pp. 129–136. ACM, New York (2006)
10. Gao, H., Hesselink, W.H.: A formal reduction for lock-free parallel algorithms. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 305–309. Springer, Heidelberg (2004)
11. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. In: Formal Aspects of Computing (FAC), (2009), `http://www.springerlink.com/content/7507m59834066h04/`
12. Colvin, R., Dongol, B.: Verifying lock-freedom using well-founded orders. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 124–138. Springer, Heidelberg (2007)
13. Colvin, R., Dongol, B.: A general technique for proving lock-freedom. Sci. Comput. Program. 74(3), 143–165 (2009)
14. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: Principles of Programming Languages, pp. 16–28. ACM, New York (2009)
15. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
16. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. SIGOPS Oper. Syst. Rev. 26(2), 108 (1992)
17. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, pp. 321–332. North-Holland, Amsterdam (1983)
18. Misra, J.: A reduction theorem for concurrent object-oriented programs. In: McIver, A., Morgan, C. (eds.) Programming methodology, pp. 69–92. Springer, New York (2003)
19. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press, Cambridge (1986)
20. Cau, A., Moszkowski, B., Zedan, H.: ITL – Interval Temporal Logic. Software Technology Research Laboratory, SERCentre, De Montfort University, The Gateway, Leicester LE1 9BH, UK (2002), `http://www.cms.dmu.ac.uk/~cau/itlhomepage`
21. Reif, W., Schellhorn, G., Stenzel, K., Balser, M.: Structured specifications and interactive proofs with KIV. In: Bibel, W., Schmitt, P. (eds.) Automated Deduction—A Basis for Applications. Systems and Implementation Techniques, vol. II, pp. 13–39. Kluwer Academic Publishers, Dordrecht (1998)
22. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free parallel and concurrent garbage collection by mark&sweep. Sci. Comput. Program. 64(3), 341–374 (2007)

23. Bäumler, S., Balser, M., Nafz, F., Reif, W., Schellhorn, G.: Interactive verification of concurrent systems using symbolic execution. AI Communications 23(2-3), 285–307 (2010)
24. Balser, M.: Verifying Concurrent System with Symbolic Execution. Shaker Verlag, Germany (2006)
25. Collette, P., Knapp, E.: Logical foundations for compositional verification and development of concurrent programs in unity. In: Alagar, V.S., Nivat, M. (eds.) AMAST 1995. LNCS, vol. 936, pp. 353–367. Springer, Heidelberg (1995)
26. Roever, W.P.D., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Methods. Cambridge Tracts in Theoretical Computer Science, vol. 54. Cambridge University Press, Cambridge (2001)
27. Online Presentation of the KIV-specifications and the Verification of the Queue (and Stack), `http://www.informatik.uni-augsburg.de/swt/projects/lock-free.html`
28. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
29. Dongol, B.: Formalising progress properties of non-blocking programs. In: Liu, Z., He, J. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 284–303. Springer, Heidelberg (2006)
30. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
31. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distrib. Comput. 18(1), 21–42 (2005)
32. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
33. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
34. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizability via non-atomic refinement. In: Davies, J., Gibbons, J. (eds.) IFM 2007. LNCS, vol. 4591, pp. 195–214. Springer, Heidelberg (2007)
35. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. 6, 319–340 (1976)
36. Colvin, R., Groves, L.: Formal verification of an array-based nonblocking queue. In: ICECCS '05: Proceedings of the 10th IEEE International Conference on Engineering of Complex Computer Systems, Washington, DC, USA, pp. 507–516. IEEE Computer Society Press, Los Alamitos (2005)
37. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, p. 1. Springer, Heidelberg (2001)