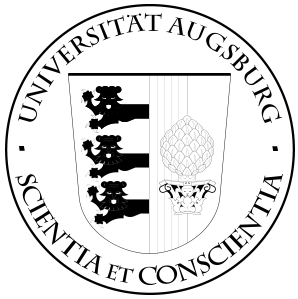
Universität Augsburg



Verifying Linearizability and Lock-Freedom with Temporal Logic

B. Tofan, S. Bäumler, G. Schellhorn, W. Reif

Report 2009-20 December 2009

Institut für Informatik
D-86135 Augsburg

Verifying Linearizability and Lock-Freedom with Temporal Logic

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

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

Abstract. The development and analysis of efficient concurrent algorithms is currently an active field of research. Lock-free implementations try to better utilize the capacity of modern multi-core computers, by increasing the potential to run in parallel. This leads to a high degree of possible interference which makes the verification of these algorithms challenging. Many techniques have been proposed to prove safety and liveness properties of these implementations. Our approach is fully mechanized and based upon rely-guarantee reasoning and 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 describe how the most complex parts of the proofs can be reduced to simple steps of symbolic execution.

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

1 Introduction

The classic approach to protect parts of a shared data structure from concurrent access are 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. Modification of the shared data structure is prepared, e.g. storage is allocated, local fields are initialized.
2. The part of the data structure to be modified is stored in a local variable (sometimes called "snapshot").
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 relevant parts of the shared state between execution of steps 2 and 3, [1] the current process must retry by executing step 2 and 3 again 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 [7–10] and liveness [11–13].

The main correctness criterion for lock-free algorithms is *linearizability*. It requires each operation to take effect instantaneously at some point (the linearization point) between its invocation and its response, behaving according to its sequential specification [14]. 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 [15]. 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.

Our verification approach is fully mechanized (using just one tool and logic to specify and verify both the derivation of proof obligations as well as their use in case studies) and provides two decomposition theorems based on rely-guarantee reasoning [16, 17] and interval temporal logic [18, 19]: a generic refinement theorem which can be instantiated to prove linearizability and a theorem for proving lock-freedom. Both theorems have been verified in the semi-automated prover KIV [20] and successfully applied on several lock-free algorithms. We demonstrate our technique using a practical lock-free queue algorithm published by Doherty et al. [7], based on the original implementation of Michael and Scott [3].

This report is an extension of [10] which mainly concentrates on the linearizability of a simple stack algorithm and briefly outlines the dequeue operation for the queue. Here we formalize the entire queue algorithm and give its linearizability proof. Additionally we describe a decomposition theorem for lock-freedom, illustrating its application on the queue. The main contributions are:

- A fully mechanized approach for the intuitive specification and verification of lock-free algorithms. Both theorems and the case study have been mechanically verified. We provide an easy to read specification language and require no program counter values for the reasoning.

[1] the number of code lines between phase 2 and 3 is typically as small as possible, in order to reduce the possibility of interference

- An expressive temporal logic framework which allows for the proof of safety and liveness properties. Moreover, the $\overset{+}{\rightarrow}$ operator from rely-guarantee reasoning can be easily defined.
- A refinement method which reduces complex parts of other proofs to simple steps of symbolic execution: no backward simulation or prophecy variable is necessary to prove refinement for the dequeue operation.
- A decomposition theorem to prove lock-freedom which does not rely on the explicit construction of well-founded orders.

We proceed as follows: in Section 2 we describe the queue algorithm and argue informally about its correctness and 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, both decomposition theorems are introduced. Section 5 shows how these theorems have been instantiated and applied to prove linearizability and lock-freedom for the queue. We conclude with a section about related work (Section 6) and a summary and outlook (Section 7).

2 The Queue Algorithm

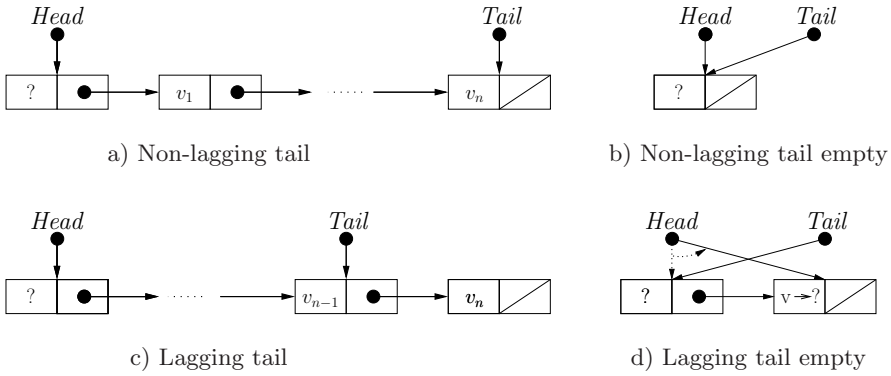
2.1 Michael and Scott's Lock-Free Queue

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

```
CAS(Old, New; G, Succ) {
  if* G = Old then {
    G := New,  Succ := true
  } else {
    Succ := false
  }
}
```

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 references are equal then G is updated to a new reference *New* and the success flag is set to true to indicate a successful CAS. Otherwise this flag is set to *false* indicating that no update has occurred. Since CAS executes atomically (a coma 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 [21] and do not explicitly model deallocation at this level of refinement, thus avoiding 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 helps to reduce the number of special cases in the implementation. There are



a) Non-lagging tail b) Non-lagging tail empty

c) Lagging tail d) Lagging tail empty

Fig. 1. Queue representation variants

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 utilized twice to cope with this problem. The first successful CAS-transition sets the tail's next field to the new node, leaving the tail pointer lagging behind the last node of the queue, as shown in Figure 1 c). The second employment of CAS shifts the lagging tail to its successive (new) node to reestablish a non-lagging tail representation. These two CAS-transitions do *not* both take effect atomically. Some other process j might encounter a lagging tail representation. In this case, j knows that another process i has successfully attached a new node but has not yet completed its operation. Process j helps i by applying CAS to shift the tail to the next node of the linked list which allows j to possibly add a new node in the next iteration. Hence, the tail pointer will never lag more than one node behind. The enqueue operation combines this helping technique with the typical retry scheme of lock-free algorithms. The formal specification of *CEnq* is shown in Figure 2.

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

```
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         }
E14       }
E15     }
E16     CAS(Tle, Newe; Tail);
E17   }
E18 }
```

Fig. 2. Formal specification of the enqueue operation

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 E16) tries to shift the tail pointer after attaching a new node at the end of 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, E16). Moreover, since it will be 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 formal specification 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 check in line D7 fails, dequeue must retry as the snapshot has become obsolete. If the test in line D8 is true, the queue was empty when D6 was executed. Hence, *CDeq* completes returning *empty*. If the test in D8 is false process i locally stores *Nxtd*'s value and applies CAS to shift the global head pointer, making *Nxtd* the new dummy node, line D13. If this CAS-transition fails, the loop body is reiterated. Otherwise the global head is shifted and the remaining lines of code (D14-D17) then deal with a special

configuration which emerges from shifting the 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, dequeue can help the process which has enqueued v (line D17). [2] An additional assignment to transient parameter O returns a value in line D23.

```
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;
D10           SuccD := true;
D11         } else {
D12           Lo := Hp[Nxtd].val;
D13           CAS(Hdd, Nxtd; Head, SuccD);
D14           if SuccD then {
D15             Tld := Tail;
D16             if Tld = Hdd then {
D17               CAS(Tld, Nxtd; Tail);
D18             }
D19           }
D20         }
D21       }
D22     }
D23     O := Lo;
D24   }
D25 }
```

Fig. 3. Formal specification of the dequeue operation

2.2 Proving Linearizability with Refinement

The key idea of proving linearizability is to identify the linearization point for each concrete operation (*CEnq* and *CDeq*). This idea can be expressed in terms of refinement as shown in Figure 4. Each finite execution of concrete steps (CSTEP_m) of a process m is mapped (using a suitable abstraction function *Abs*) to an adequate abstract execution, given certain restrictions R_m on the behavior of other processes. Since our formalization requires corresponding concrete and

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

abstract executions to have the same number of steps, the externally invisible concrete steps are mapped to no operation steps (skip). In-between these skip steps the atomic step (ALINSTEP_m) corresponding to the linearization point must take place.

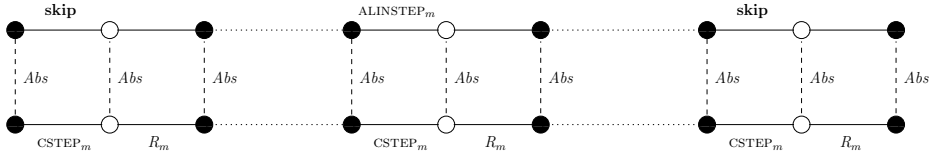


Fig. 4. Principle of proving linearizability via refinement

Abstractly, a queue is algebraically specified as type *queue* with atomic operations *enq* that adds an element (of type *elem*) at the end and *deq* that removes the first element. This specification is extended as shown in Figure 5 to meet the described refinement requirements. In *AEnq* parameter $v : elem$ is the value that

```
AE1 AEnq(v; Queue) {
AE2     skip*;
AE3     Queue := enq(Queue, v);
AE4     skip*
AE5 }
```

```
AD1  ADeq(; Queue, O) {
AD2     let Lo = empty in {
AD3         skip*;
AD4         if* Queue ≠ emptyq then {
AD5             Lo := head(Queue),
AD6             Queue := deq(Queue);
AD7         }
AD8         skip*;
AD9         O := Lo
AD10    }
AD11 }
```

Fig. 5. Abstract enqueue and dequeue operations

is attached to the queue and parameter $Queue : queue$ represents the queue. This extended abstract operation is linearizable because its finite executions consist of an arbitrary number of externally invisible skip steps (AE2) the atomic enqueue operation on the queue (AE3) and some further skip steps. The dequeue operation is similarly extended to a linearizable operation *ADeq*. Its linearization point is the if-statement (AD4) which discerns whether the queue is currently empty (*emptyq*). An additional assignment to transient parameter O is required to return a value.

2.3 Linearizability and Lock-Freedom of MS-Queue

The linearization point of the concrete operation *CEnq* is its successful CAS in line E10. This transition corresponds to abstractly adding an element at the end of the queue (AE3). All other transitions have no externally visible effect.

To find the possible linearization point for *CDeq*, two cases are discerned depending on what is read in line D6. If *Nxtd* is set to a non-null reference and the head pointer remains unchanged since the snapshot was taken, the linearization point is simply the successful CAS-transition in line D13. This transition removes the oldest value from the queue whereas all other transitions leave the queue untouched. The case of an empty queue is harder: only when setting *Nxtd* to null can *CDeq* assure that it was executed on an empty queue. However, this transition does not guarantee completion as shown in Figure 6. If the snapshot becomes obsolete between execution of lines D6 and D7, the current process must reiterate the loop and execution of line D6 has not been a linearization point, i.e. concrete trace (1) corresponds to abstract trace (4). If the snapshot is still accurate when line D7 is executed, D6 has been the linearization point corresponding to running *ADeq* on an empty queue, i.e. concrete trace (2) corresponds to abstract trace (3).

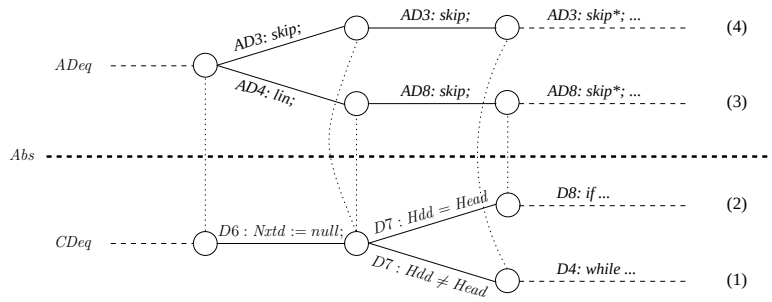


Fig. 6. Dequeue-empty refinement

Whether line D6 is a linearization point depends on future behavior which cannot be determined at the point of execution. It is this place, where verification approaches that refine single steps of a concrete algorithm individually, run into problems and require additional techniques since an abstract V-shaped diagram is refined to a Y-shaped diagram (moving nondeterminism backward). The solution of our approach is easier, since we directly verify trace inclusion.

The intuitive reason why the implementation is lock-free is that its loops are retried only if some other process changes the corresponding global part of the queue in the critical time slot between taking a snapshot and trying to update the data structure. This interference however implies that the interfering process

completes. In the formal proof we will reflect the simplicity of this intuitive argument by using an additional predicate U ("unchanged") which describes interference-freedom. Proving lock-freedom then requires to show that each data structure operation eventually terminates when it encounters no interference or when it changes the shared state itself. The dequeue operation exits its loop if it encounters no interference and if it changes the shared state, because the loop flag is set to true in line D13. If the enqueue operation is not interfered with after taking the snapshot in E6, its loop might be executed again before enqueue terminates, due to a lagging tail. 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 [22, 23].

3.1 Interval Temporal Logic

The basis of interval temporal logic (ITL) [18, 19] 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 [24, 25]).

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. If the interval consists of one state only then $V'' := V' := V$ by convention.

The logic does not distinguish between formulas, terms, or even parallel programs; every operator defines an expression. Although we usually write α, β to indicate a program and φ, ψ to indicate a formula, programs and formulas can be mixed arbitrarily since they both evaluate to true or false over an interval I (hence system descriptions can be abstracted by temporal properties). In particular, a program evaluates to true ($I \models \alpha$) if I is a possible run of the program.

Expressions are evaluated over an algebra and an interval I. Predicate logic formulas are evaluated in the first state of an interval in the usual way. To

$\Box\ \varphi$	φ holds from now on in every state
$\Diamond\ \varphi$	there exists a state where φ holds
last	the current state is the last
φ **until** ψ	ψ holds eventually and φ holds in every state before
φ **unless** ψ	φ holds always or until a state with ψ is reached
$\alpha \parallel \beta$	weak fair interleaving
$X := t_1, Y := t_2$	parallel assignment (leaves other variables unchanged)
$\alpha \vee \beta$	nondeterministic choice
choose X **with** φ **in** α	runs α with local variable X satisfying φ
$\alpha; \beta$	sequential composition
if ψ **then** α_1 **else** α_2	case distinction
if* ψ **then** α_1 **else** α_2	case distinction (evaluation of ψ requires no step)
let $X = t$ **in** α	local variable declaration
while ψ **do** α	loop
α^*	iterate α any (finite or infinite) number of times
$p(\bar{x}; \bar{y})$	procedure call (only variables in $\bar{y}$ are modified)

Fig. 7. Temporal operators and programming constructs

describe properties of an interval the standard first order operators are extended with temporal operators and programming constructs as shown in Figure 7.

Assignments are placed within a program frame $[.]_{V1,\ldots,Vn}$ to avoid expressions with infinitely many free variables (similar to TLA [26], but without stuttering). As an example the semantics of the program

$$[X := 1]_{X,Y,Z}$$

consists of all intervals $[I(0), I'(0), I(1)]$ with one program transition from $I(0)$ to $I'(0)$ that changes X' to 1 and leaves Y and Z unchanged. All other variables may change arbitrarily (and are therefore not free in the formula). The environment transition from $I'(0)$ to $I(1)$ is not constrained by the program.

We exemplify the semantics of the **until** operator and the sequential composition operator in more detail ($|I|$ is the length of an interval):

$$\begin{aligned} I \models \varphi \textbf{ until } \psi \ :\Leftrightarrow\ & \exists\, n \in \mathbb{N}_0, n \le |I| \,.\quad (I(n), I'(n), \ldots) \models \psi \\ & \qquad \wedge\ \forall\, m,\ 0 \le m < n.\ (I(m), I'(m), \ldots) \models \varphi \\ I \models \alpha; \beta \ :\Leftrightarrow\quad & \exists\, n \in \mathbb{N}_0, n \le |I| \,.\quad (I(0), I'(0), \ldots, I(n)) \models \alpha \\ & \qquad \wedge\ (I(n), I'(n), \ldots) \models \beta \\ & \vee\ |I| = \infty \wedge I \models \alpha \end{aligned}$$

Other temporal logic operators, such as $\Box$, $\Diamond$ or **unless** can be derived, e.g.

$$\begin{aligned} \Diamond\ \varphi &:\equiv \mathit{true}\ \textbf{until}\ \varphi \\ \Box\ \varphi &:\equiv \neg \Diamond \neg\ \varphi \\ \varphi\ \textbf{unless}\ \psi &:\equiv \Box\ \varphi \vee \varphi\ \textbf{until}\ \psi \end{aligned}$$

3.2 Symbolic Execution and Induction

KIV is based on the sequent calculus. Sequents are assertions of the form

$$\Gamma \vdash \Delta$$

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

$$\alpha, E, I \vdash \varphi$$

where an interleaved program α 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 φ is the property which has to be shown. To verify that φ holds, symbolic execution is used. For example, a sequent of the form mentioned above might be:

$$[M := M + 1; \alpha]_M,\ \Box\ M'' = M', M = 1 \ \vdash\ \Box\ M > 0$$

The program executed is $M := M + 1; \alpha$ where α is an arbitrary program and the environment is assumed never to change counter M (formula $\Box\ M'' = M'$). The current state maps M to 1. The intuitive idea of a symbolic execution step is to execute the first program statement, i.e. applying 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

$$M = 1 \vdash M > 0$$

and a sequent that describes the remaining interval from the second state on

$$[\alpha]_M,\ \Box\ M'' = M', M = 2 \ \vdash\ \Box\ M > 0$$

M has value 2 in the new state. This 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. More complex formulas in the succedent might change too during a step (e.g. if the formula in the succedent is a program too, it has to be symbolically executed like the example program in the antecedent).

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 ordering can often be derived from liveness properties $\Diamond\ \varphi$ by inducing over the number of steps N until φ holds for the first time.

$$\Diamond\ \varphi\ \leftrightarrow\ \exists\ N.\ (N = N'' + 1)\ \textbf{until}\ \varphi$$

The equivalence states that φ is eventually true, if and only if N can be decremented (note that $N = N'' + 1$ is equivalent to $N > 0 \wedge N'' = N - 1$) until φ

becomes true. Proving a formula of the form $\Box\ \varphi$ is simply done by rewriting $\Box\ \varphi$ to $\neg \Diamond \neg\ \varphi$ and a proof by contradiction. Similarly, an **unless** formula (as needed later in rely-guarantee proofs) can be reduced to the case of an eventually formula using the equivalence

$$\varphi \textbf{ unless } \psi \leftrightarrow \forall\ B.\ (\Diamond\ B) \rightarrow (\varphi \textbf{ unless } (\psi \vee B))$$

φ **unless** ψ 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.

General safety formulas satisfy the principle, that they are true on an infinite interval I, already if every finite prefix $(I(0), I'(0), \ldots, I(n))$ of I can be extended with some interval J that starts with $J(0) = I(n)$ such that φ is true on this concatenation (see e.g. [27]). This class includes always, unless formulas as well as sequential programs without local variables and all predicate logic formulas (including those that mention primed and double primed variables). Safety formulas are closed against conjunction and disjunction (but not negation). The general principle of induction over the length of a prefix for arbitrary safety formulas is explained in [10].

4 Rely Guarantee Reasoning and Decomposition Theorems

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 present a refinement theorem that can be instantiated to prove linearizability, and a decomposition theorem for proving lock-freedom. The formal proof of both theorems can be found online at [28].

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. The following formal specification reflects this definition.

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

CSpawn is a concurrent system consisting of $n+1$ processes that execute CSeq in parallel. Operation CSeq finitely or infinitely often 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* or *CDeq*).

$$\begin{array}{l}\text{CSeq}(m; Act, In, CS, Out)\ \{\\ \quad (\,\textbf{skip}\ \vee\ \{Act(m) := true; \text{COP}(m, In; CS, Out); Act(m) := false;\})^*\\ \}\end{array}$$

Operation CSeq is called with a value parameter m of type *nat* which represents the identifier of the invoking process. Reference-parameter $Act : nat \to bool$ 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 $In : nat \to input$ 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 $Out : nat \to output$ for returned 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 [16, 17]. To this end each process (component) m is extended with two predicates: a two state (rely) predicate $R_m : cstate \times cstate$ 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 $G_m : cstate \times cstate$ 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.

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

The intuitive idea of the rely-guarantee approach is to claim that every process m fulfills G_m if every other process does not violate its rely condition R_m. This argument involves circular reasoning. To break circularity, a special implication operator $\overset{+}{\to}$ (as defined in [29]) is used to state that m fulfills its guarantee if its rely condition has not been violated in some preceding step (m satisfies $R_m \overset{+}{\to} G_m$). 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 $R_m(CS', CS'')$ can be specified using primed and double primed variables to restrict steps of m's environment. The formal definition of $\overset{+}{\to}$ is then simply based on the **unless** operator:

$$R_m \overset{+}{\to} G_m := G_m(CS, CS')\,\textbf{unless}\,(G_m(CS, CS') \wedge \neg\, R_m(CS', CS''))$$

In order to show that a process m satisfies $R_m \overset{+}{\rightarrow} G_m$, 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, $R_m \overset{+}{\rightarrow} G_m$ must be preserved by COP (by convention we omit writing frame variables whenever the list of frame variables is equal to the list of reference parameters).

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

To meet the need for invariant properties in proofs, we introduce an invariant predicate $Inv : cstate$. 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 the invariant is also stable over each local guarantee G_m, i.e. it is indeed an invariant property of CSeq.

$$\textsc{CSeq}(m; \dots), \Box\, R_m(CS', CS''), Inv(CS) \vdash \Box\, (Inv(CS) \wedge Inv(CS'))$$

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 claim R_m to be transitive.

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

Finally, since the generic setting assumes that even the interleaving of all processes still has a global environment which can do steps, we also define a global rely condition $R : cstate \times cstate$ and stipulate that this predicate preserves each local rely condition.

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

As several of the following proof obligations will assume R_m and Inv to always hold, we introduce the following abbreviation.

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

4.2 Decomposition Theorem for Linearizability

The rely-guarantee theory described in the previous subsection is applied to prove trace inclusion (refinement) between the previously defined concrete concurrent system CSpawn and an abstract interleaved system ASpawn. This abstract system is defined analogously to the concrete system; the only difference

is that it works on an abstract state $AS : astate$ and that its components execute an abstract operation ASeq which arbitrary often runs **skip** or abstract operation AOP (for the linearizability proof of the queue, it corresponds to the nondeterministic choice between abstract operations $AEnq$ and $ADeq$).

There is a straightforward way in our formalism to express trace inclusion between concrete and abstract operations by simply stating COP $\vdash$ AOP. Taking into account the behavior of the environment, the need for invariant properties and data refinement, the following local proof obligation evolves.

$$\begin{array}{l} \mathrm{COP}(m, In; CS, Out), \Box\ I(R_m) \\ \vdash \exists\ AS.\ \mathrm{AOP}(m, In; AS, Out) \wedge \Box\ (Abs(AS, CS) \wedge Abs(AS', CS')) \end{array} \quad (7)$$

This formula is the formal equivalent to Figure 4. As usual, an abstraction predicate $Abs : astate \times cstate$ is used to relate a concrete state with its abstract counterpart(s). This predicate must hold at all times during execution of COP. Moreover, abstract and concrete operations must have the same externally visible behavior, i.e. they work on the same input and yield the same output. According to the previous section, the invariant Inv and a suitable rely condition R_m can be assumed to hold in every state, respectively environment transition.

Based on (7) and the requirements from the previous section, the following composition theorem can be shown.

Theorem 1 (Decomposition Theorem for Refinement).
If formulas (1) to (7) hold, then:

$$\begin{array}{l} \mathrm{CSpawn}(n; In, CS, Out), \Box\ R(CS', CS''), Inv(CS) \\ \vdash \exists\ AS.\ \mathrm{ASpawn}(n; In, AS, Out)) \wedge \Box\ (Abs(AS, CS) \wedge Abs(AS', CS')) \end{array}$$

The theorem states that for every interleaved run of COP-operations, an equivalent abstract run of operations AOP exists such that concrete and abstract operations execute on the same input and yield the same output.

4.3 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 [15]. There are two further important liveness properties [30]: *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 $Act(m)$. In a concurrent system which consists of n processes, global progress P is defined in terms of the activity flags as

$$\begin{array}{l} \quad P(n, Act, Act') \\ \leftrightarrow (\exists\ m \leq n.\ Act(m)) \rightarrow \Diamond\ (\exists\ k \leq n.\ Act(k) \wedge \neg\ Act'(k)) \end{array}$$

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 $U : cstate \times cstate$ ("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

$$\begin{array}{l} U(CS, CS) \\ U(CS_0, CS_1) \wedge U(CS_1, CS_2) \rightarrow U(CS_0, CS_2) \end{array} \tag{8}$$

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:

$$\begin{array}{l} \quad R_{ext}(CS', Act', CS'', Act'') \\ \leftrightarrow R(CS', CS'') \wedge Act'' = Act' \wedge U(CS', CS'') \end{array}$$

Lock-freedom of system CSpawn then follows from the following intuitive local proof obligation

$$\begin{array}{l} \quad \mathrm{COP}(m, In; CS, Out), \Box\ I(R_m) \\ \vdash \Box\ (\neg\ U(CS, CS') \vee \Box\ U(CS', CS'') \rightarrow \Diamond\ \mathbf{last}) \end{array} \tag{9}$$

At any time (leading $\Box$), a lock-free operation that updates the shared state itself in a step ($\neg\ U(CS, CS')$) or encounters no interference ($\Box\ U(CS', CS'')$), eventually terminates ($\Diamond$ **last**).

Properties (8) and (9) 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 2 (Decomposition Theorem for Lock-Freedom).
If formulas (1) to (6), (8), and (9) hold, then:
$\mathrm{CSpawn}(n; \ldots), \Box\ R_{ext}, Inv(CS), \forall\ m \leq n.\ \neg\ Act(m) \vdash \Box\ P(n, Act, Act')$

Given R_{ext} at all times, the presence of an active operation will always lead to the completion of some (active) operation. The theorem was proved in KIV by induction over the always formula in the succedent (cf. Section 3). The proof is available on the web [28].

5 Proving Linearizability and Lock-Freedom for the Queue

In this section we present the instantiation of the decomposition theorems for the queue algorithm and shortly outline the proofs. Full details are available online at [28].

5.1 Linearizability

The generic operation *COP* representing the possible operations on the shared data structure is instantiated with the nondeterministic choice between one of the two queue operations *CEnq* and *CDeq* (see Section 2). The generic state variable *CS* becomes a tuple consisting of a shared state *Hp*, *Head*, *Tail* and local states $Newef(m)$, $Tlef(m)$, $Nxtef(m)$, $Succef(m)$, $Hddf(m)$, $Nxtdf(m)$, $Succdf(m)$ for every process m. The abstract state variable *AS* simply corresponds to the dynamic variable *Queue* from Figure 5 and operation AOP is the nondeterministic choice between *AEnq* and *ADeq*.

Abstraction Predicate. In order to prove the refinement relation from AOP to COP, a suitable instantiation of the generic abstraction predicate *Abs* has to be defined. The shared variables *Hp*, *Head* and *Tail* of a concrete state are sufficient to uniquely define its abstract semantics. Using *v: elem* and *Q: queue*, the abstraction predicate is recursively defined in three steps (see Figure 1).

Step one defines the representation of the empty queue (*emptyq*).

$$
\begin{array}{ll}
 & Abs(emptyq, Hp, Head, Tail) \\
\leftrightarrow & \quad Head \neq null \wedge Head \in Hp \wedge Tail \neq null \wedge Tail \in Hp \\
 & \wedge\ Hp[Head].\mathrm{nxt} = null \\
 & \wedge\ (Tail = Head \vee Tail \neq Head \wedge Hp[Tail].\mathrm{nxt} = Head)
\end{array}
$$

Pointers *Head* and *Tail* are not null and both allocated in the global heap *Hp*; the head pointer has a null next pointer and the representation might have a non-lagging tail pointer ($Tail = Head$), or head is ahead of tail ($Hp[Tail].\mathrm{nxt} = Head$). Step two defines a queue with exactly one element v.

$$
\begin{array}{ll}
 & Abs(v, Head, Tail, Hp) \\
\leftrightarrow & \quad Head \neq null \wedge Head \in Hp \wedge Hp[Hp[Head].\mathrm{nxt}].\mathrm{val} = v \\
 & \wedge\ (Tail = Head \vee Tail \neq Head \wedge Tail = Hp[Head].\mathrm{nxt}) \\
 & \wedge\ Abs(emptyq, Hp[Head].\mathrm{nxt}, Tail, Hp)
\end{array}
$$

The dummy node is not null, allocated and its successor node contains value v. Pointer *Tail* might be lagging, i.e. $Tail = Head$, or the tail pointer is direct successor of *Head*; the triple $(Hp[Head].\mathrm{nxt}, Tail, Hp)$ represents the empty queue. Step three covers the representation of a queue with more than one element, i.e.

Q is assumed to not be empty in the following definition.

$$\begin{array}{ll} & Abs(v + Q, Head, Tail, Hp) \\ \leftrightarrow & \quad Head \neq null \wedge Head \in Hp \wedge Tail \neq Head \\ & \wedge\ Hp[Hp[Head].\text{nxt}].\text{val} = v \\ & \wedge\ (\quad Hp[Tail].\text{nxt} = null \\ & \qquad \vee \quad Hp[Tail].\text{nxt} \neq null \wedge Hp[Tail].\text{nxt} \in Hp \\ & \qquad\quad \wedge\ Hp[Hp[Tail].\text{nxt}].\text{nxt} = null) \\ & \wedge\ Abs(Q, Hp[Head].\text{nxt}, Tail, Hp)) \end{array}$$

The dummy node *Head* is not null and allocated; it is unequal *Tail* and its direct successor stores value v. When the tail pointer does not lag, it marks the end of the queue, i.e. $Hp[Tail].\text{nxt} = null$. Otherwise, it points to an allocated node with a null next pointer and finally (last conjunct), the rest of the queue must have an abstract semantics according to Abs.

A concrete state is called *valid* if there is a corresponding abstract queue such that Abs is satisfied. The above abstraction predicate defines a partial function $Absf : cstate \rightarrow astate$ on valid states. This allows for dropping the existential quantifier in proof obligation (7) by using $Absf(CS)$ as a witness for the existence of a unique sequence of abstract states AS, thus allowing for an inductive prove for the remaining safety formula (an instantiation for the local abstract output variable can be found too). As we will see, the invariant property ensures that all concrete states are valid throughout execution.

Rely-Guarantee Conditions and Invariant. Since all processes m execute the same set of operations, all processes will have the same rely condition R_m by symmetry. It consists of the invariant property Inv and predicates $Enqlocal_m$, $Deqlocal_m$.

$$\begin{array}{ll} & R_m(CS', CS'') \\ \leftrightarrow & \quad (Inv(CS') \rightarrow Inv(CS'')) \\ & \wedge\ Enqlocal_m(CS', CS'') \wedge Deqlocal_m(CS', CS'') \end{array}$$

Process m assumes that other processes will preserve the invariant and respect its locality assumptions (see below). The invariant is the conjunction of predicates *okrep*, *disj* and *valid*.

$$Inv(CS) \leftrightarrow okrep(CS) \wedge disj(CS) \wedge valid(CS)$$

Predicate *okrep* describes an admissible local pointer structure for each process m. It is the conjunction of several assertions that are described in the following. One rather standard assertion claims that pointers P are not dangling but pointing to the heap Hp. It has the form

$$P \neq null \rightarrow P \in Hp$$

where P is one of the following local pointers: $Newef(m)$, $Tlef(m)$, $Nxtef(m)$, $Hddf(m)$, $Nxtdf(m)$. This property must also hold for $Hp[Tlef(m)].\text{nxt}$ and

$Hp[Hddf(m)]$.nxt given that the local snapshot pointers $Tlef(m)$ respectively $Hddf(m)$ are not null.

Another rather standard invariant property requires disjointness of newly allocated nodes from the queue. Moreover, these nodes must have a null next reference.

$$\begin{array}{ll} & \neg\ Succef(m) \wedge Newef(m) \neq null \\ \rightarrow & \quad \neg\ reachable(Head, Newef(m), Hp) \\ & \wedge\ Hp[Newef(m)].\mathrm{nxt} = null \end{array}$$

Predicate $reachable(R_0, R_1, H)$ defines reachability of a non-null reference R_1 in heap H from R_0 via next pointers in the standard way.

Finally, the following interconnection between local snapshot pointers and their shared counterparts is required.

$$\begin{array}{l} Tlef(m) \neq null \wedge Hp[Tlef(m)].\mathrm{nxt} = null \rightarrow Tlef(m) = Tail \\ Hddf(m) \neq null \wedge Hp[Hddf(m)].\mathrm{nxt} = null \rightarrow Hddf(m) = Head \end{array} \tag{10}$$

If a snapshot's next reference is null, the local snapshot still coincides with its global counterpart.

With respect to interference between two processes m and n, predicate $disj$ claims disjointness of newly allocated nodes.

$$\begin{array}{ll} & disj(CS) \\ \leftrightarrow & Newef(m) \neq null \rightarrow Newef(m) \neq Newef(n) \end{array}$$

As already explained, a process assumes a valid representation in every concrete state which is formally reflected by predicate *valid*. It describes the set of concrete states that have an abstract counterpart according to Abs.

$$valid(CS) \leftrightarrow \exists\ AS.\ Abs(AS, CS)$$

The remaining properties ($Enqlocal_m$, $Deqlocal_m$) of the local rely condition R_m reflect locality of specific pointer variables or heap nodes. Several of these statements result from lifting local variables to global functions in order to be able to refer to them in assertions. Their typical form is

$$V'' = V'$$

where V'' (V') denotes variable V after (before) an environment transition. Predicate $Enqlocal_m$ asserts this for local variables $Succef(m)$, $Newef(m)$, $Tlef(m)$ and $Nxtef(m)$. Furthermore, it contains the following properties.

$$\begin{array}{ll} & (Tlef'(m) \neq null \wedge Hp'[Tlef'(m)].\mathrm{nxt} \neq null \\ & \quad \rightarrow Hp''[Tlef''(m)].\mathrm{nxt} = Hp'[Tlef'(m)].\mathrm{nxt}) \\ \wedge & (\neg\ Succef'(m) \wedge Newef'(m) \neq null \\ & \quad \rightarrow Hp''[Newef''(m)] = Hp'[Newef'(m)]) \end{array}$$

Whenever the snapshot's next pointer is not null, this reference remains untouched by m's environment. A newly allocated node will not be modified by

other processes as long as m has not attached it to the queue (indicated by $Succef'(m)$ being false).

Similar assumptions are defined in predicate $Deqlocal_m$.

$$
\begin{array}{ll}
 & Deqlocal_m(CS', CS'') \\
\leftrightarrow & \quad Succdf''(m) = Succdf'(m) \\
 & \wedge\ Hddf''(m) = Hddf'(m) \wedge Nxtdf''(m) = Nxtdf'(m) \\
 & \wedge\ (\quad Hdd'(m) \neq null \wedge Hp'[Hddf'(m)].\mathrm{nxt} \neq null \\
 & \qquad \rightarrow \quad Hp''[Hddf''(m)].\mathrm{nxt} = Hp'[Hddf'(m)].\mathrm{nxt} \\
 & \qquad\quad \wedge\ Hp''[Hp''[Hddf''(m)].\mathrm{nxt}].\mathrm{val} = \\
 & \qquad\quad\ \ Hp'[Hp'[Hddf'(m)].\mathrm{nxt}].\mathrm{val})
\end{array}
$$

In particular, if the snapshot's next reference is not null, other processes leave this reference unchanged as well as the value of its direct successor node.

The remaining predicates (the local guarantee condition and the global rely condition) are weakly defined according to constraints (1) and (6) from Section 4.

Proof Outline. It is straightforward to show that the described instantiation fulfills the predicate logic requirements from Section 4. The proof for the temporal logic proof obligation (3) also requires only few interactions. More work has to be done to prove the refinement relation (7) from *AEnq* to *CEnq*, resp. from *ADeq* to *CDeq*. These proofs mainly consist of symbolically stepping through the concrete code and choosing the correct abstract counterpart for each step.

The correctness of an enqueue operation strongly relies on properties from $Enqlocal_m$. It is for instance vital to know that once m has successfully attached its new node at the end of the queue, the environment leaves the snapshot's next reference unchanged. Otherwise the abstraction predicate would be violated by shifting the tail pointer in line E16 of *CEnq*. Similarly, changes to the newly allocated node, as long as m has not attached it to the queue, could violate the representation, e.g. by enqueuing a wrong value.

While the choice of an abstract step is unique for the enqueue refinement, there is more than one possible abstract step to choose, when the queue is empty (see Figure 6) and m executes the instruction at line D6 of *CDeq* (setting $Nxtdf(m)$ to $null$). We outline the proof for this transition in Figure 8. There **Dk** denotes the remaining program of *CDeq* starting from line Dk, e.g. **D6** denotes the sequence of instructions D6 to D10, followed by the while loop. Analogously **ADk** stands for the remaining code of *ADeq* from line ADk. Further abbreviations are introduced in the figure. The conclusion of the proof tree shows the proof goal after symbolically executing lines D1 - D5 of *CDeq*, the program to execute is **D6**. Process i has reached a state which satisfies predicate logic formula S_0. Assuming the rely guarantee and invariance conditions, it has to be shown that the sequence of values $Absf(Head, Tail, Hp)$ is an execution of **AD3** which preserves the abstraction (A).[3]

[3] In the following we abstract from the substitution of the abstract state and output variables to improve readability.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dots}{\mathbf{D8}, RI, S_2 \vdash \dots}}{\dots \vdash \mathbf{AD8} \wedge A}\,(7)}{\dots, S_1, Hddf(m) = Head \vdash \dots}\,(5) \quad \dfrac{\dfrac{\dfrac{}{\mathbf{D4}, RI, S_3 \vdash \dots}\,(9)}{\dots \vdash \mathbf{AD3} \wedge A}\,(8)}{\dots, S_1, Hddf(m) \neq Head \vdash \dots}\,(6)}{\mathbf{D7},\ RI,\ S_1 \vdash \mathbf{AD3} \wedge A,\ \mathbf{AD8} \wedge A}\,(4)}{\mathbf{D6},\ RI,\ S_0, Hp[Hddf(m)].\mathrm{nxt} = null \vdash \mathbf{skip}\,;\mathbf{AD3} \wedge A,\ \mathbf{AD4} \wedge A}\,(3)}{\mathbf{D6},\ RI,\ S_0, Hp[Hddf(m)].\mathrm{nxt} = null \vdash \mathbf{AD3} \wedge A}\,(2) \qquad \cdots \neq null \dots}{\mathbf{D6},\ RI,\ S_0 \vdash \mathbf{AD3} \wedge A}\,(1)$$

$$\begin{aligned}
RI &:\equiv \Box\ I(R_m)\\
A &:\equiv \Box\ Abs(Queue, CS) \wedge Abs(Queue', CS')\\
S_0 &:\equiv \neg\ Succdf(m) \wedge Hddf(m) \neq null \wedge Abs(Queue, CS)\\
S_1 &:\equiv \neg\ Succdf(m) \wedge Hddf(m) \neq null \wedge Nxtdf(m) = null \wedge Abs(Queue, CS)\\
S_2 &:\equiv \neg\ Succdf(m) \wedge Hddf(m) \neq null \wedge Nxtdf(m) = null \wedge Abs(Queue, CS)\\
S_3 &:\equiv \neg\ Succdf(m) \wedge Hddf(m) \neq null \wedge Nxtdf(m) = null \wedge Abs(Queue, CS)
\end{aligned}$$

Fig. 8. Proof Outline Dequeue Empty

The first step (1) of the proof does a case split on whether $Hddf(m)$'s next pointer is *null*, to get the critical case in the first premise (left hand side). The current queue is then indeed empty, since according to invariant property (10), the snapshot $Hddf(m)$ is still the head of the queue.

The next step (2) expands the star operator using the equivalence

$$[skip^*; \gamma] \equiv [skip;\ skip^*;\ \gamma] \vee [\gamma]$$

for an arbitrary program γ (either another skip is executed or γ starts immediately). This gives the two succedent formulas of the premise.

The next step (3) symbolically executes the instruction at D6 which sets $Nxtdf(m)$ to *null* and the skip transition of the first formula in the succedent is executed. The second formula in the succedent also executes a skip step, as the test of the if-statement at AD4 is false. The remaining program in this case is just $skip^*; O := Lo$ (written **AD8** in the goal). This gives a proof obligation according to the premise of (3) and a side goal which demands to prove that the last step has indeed preserved A. The proof of this side goal is trivial, as S_0 fulfills Abs and none of the three formulas has changed the concrete resp. abstract state. Therefore, both abstract transitions can be pursued in constructing an abstract trace. This is not the case with linearization points that change the data representation, where only one choice of executing an abstract skip or executing the abstract operation will give a suitable abstract state. For these, one of the two resulting formulas will simplify to false and disappear. Delaying the decision whether the linearization point has been executed, seems to be possible only when the abstract data structure is not modified. The case is common for

operations that merely observe the data structure (e.g. the test for an element being in a list in the algorithm studied in [9]).

The proof continues with the new state S_1. This is the state after execution of D6 and after the environment has executed its rely step. In this state the case split whether the last step has been a linearization point can be made (step (4) of the proof). If the global head pointer has not changed since the snapshot was taken, executing the transition in line D6 has been a linearization point. Formula **AD3** $\wedge$ A can be weakened from the succedent in step (5) by applying the weakening rule of the sequent calculus which permits to strengthen a sequent by dropping formulas from its succedent (or antecedent). Since in the current state formulas $Hddf(m) = Head$ and $Nxtdf(m) = null$ hold, the following symbolic execution steps (7), ... lead to eventually exiting the loop-body and returning output value *empty*. If however the snapshot is deprecated, the executed concrete transition has not been a linearization point and formula **AD8** $\wedge$ A can be weakened in step (6). In this case of non-linearization the next symbolic execution step (8) jumps back to the start of the loop body **D4** and the resulting sequent can be closed with an inductive argument in step (9).

5.2 Lock-Freedom

According to proof obligation (9) 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

$$Id_H :\equiv Head'' = Head'$$

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

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

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)

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

Predicate U is the conjunction of these identities:

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

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 (9) from Section 4.3 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. For enqueue we get two proof obligations

$$\begin{array}{l} \mathbf{E1}, \Box\ I(R_m) \vdash \Box\ (\neg\ U(CS, CS') \rightarrow \Diamond\ \mathbf{last}) \\ \mathbf{E1}, \Box\ I(R_m) \vdash \Box\ (\Box\ U(CS', CS'') \rightarrow \Diamond\ \mathbf{last}) \end{array} \tag{11}$$

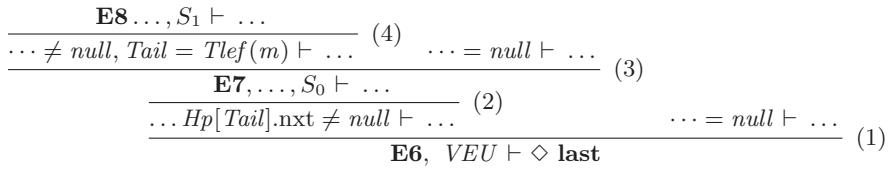
where **Ek** denotes the remaining program starting from line Ek, e.g. $\mathbf{E1} \equiv CEnq$. The first is rather simple, since the only step with $\neg\ U(CS, CS')$ is a succeeding CAS at line E10 which sets the loop-flag to true, so the algorithm terminates after the final step E13.

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 $\Box\ U(CS', CS'')$ must lead to termination. This can be proved by stepping to the start of the loop (instruction E5) and applying the following lemma

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

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

Its proof requires no induction, but 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 9. 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 **E6** (instruction E6 takes the local snapshot $Tlef(m)$). In a valid state, the required rely conditions and the unchanged predicate are assumed to hold at all times (VEU); 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 ($Hp[Tail].\text{nxt} \neq null$). If the tail pointer is not lagging (second premise, right hand side) no further interference will hinder m to complete according to VEU, i.e. the proof consists of executing **E6** until completion. If the tail pointer is lagging behind (first



$$VEU :\equiv \Box\ (valid(Head, Tail, Hp) \wedge Enqlocal_m(CS', CS'') \wedge U(CS', CS''))$$
$$S_0 :\equiv Tlef(m) \neq null$$
$$S_1 :\equiv Tlef(m) \neq null \wedge Nxtef(m) \neq null \wedge Hp[Tlef(m)].\mathrm{nxt} = Nxtef(m)$$

Fig. 9. Proof outline enqueue lock-free

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 **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 **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. $Tail = Tlef(m)$, and the proof continues with symbolic execution of E7 (proof step (4)). In the new state S_1, the snapshot's next reference is $Nxtef(m)$ 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 (11) 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 [10] for details).

Vafeiadis [31] 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. [32] 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. [33] using shape analysis [34]. 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. [35]. 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 [36] like proof obligations are generated. The approach is the only one we are aware of, that proves linearizability formally using the original definition of [14]. All other approaches (including ours in [10]) argue informally that linearizability holds.

Related to lock-freedom, we are aware only of two approaches: Colvin and Dongol [11, 12] describe the verification of several lock-free implementations (including Treiber's stack and Michael and Scott's queue) 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. [13] 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 a concurrent system model with potentially unfair scheduling, while we assume 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 could be replaced with an unfair one. Yet 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.

7 Summary

We have described two decomposition theorems that reduce the proof of global properties to process-local proof obligations and we have shown how these theorems can be applied to prove linearizability and lock-freedom of a non-trivial lock-free queue implementation. All specifications and proofs are fully mechanized in the interactive theorem prover KIV and many proofs in the queue case study have been highly automated. As explained in Section 5 we believe that our technique slightly reduces the proof effort of other approaches both regarding linearizability and lock-freedom.

In current and future work we consider the ABA-problem in an additional refinement step, by extending the current implementation with explicit deallocation and version numbers. Moreover, we try to improve our method by better exploiting the symmetry of typical lock-free implementations in the rely-guarantee theory. Another aspect is the inclusion of the formal definition of linearizability within the reduction approach. We plan to test these improved techniques on further and more complex non-blocking implementations.

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, New York, NY, USA, ACM Press (2004) 206–215

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. (1996) 267–275
4. Michael, M.M.: Cas-based lock-free algorithm for shared deques. In: In the 9th Euro-Par Conference on Parallel Processing, Springer Verlag (2003) 651–660
5. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA 2002, ACM (2002) 73–82
6. Doherty, S., Detlefs, D.L., Groves, L., Flood, C.H., Luchangco, V., Martin, P.A., Moir, M., Shavit, N., Jr., G.L.S.: 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, New York, NY, USA, ACM (2004) 216–224
7. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE 2004. Volume 3235 of LNCS. (2004) 97–114
8. Groves, L., Colvin, R.: Trace-based derivation of a scalable lock-free stack algorithm. Formal Aspects of Computing (FAC) **21**(1–2) (2009) 187–223
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, New York, NY, USA, ACM (2006) 129–136
10. Bäumler, S., Schellhorn, G., Tofan, B., Reif, W.: Proving linearizability with temporal logic. Formal Aspects of Computing (FAC) (2009) appeared online first, http://www.springerlink.com/content/7507m59834066h04/.
11. Colvin, R., Dongol, B.: Verifying lock-freedom using well-founded orders. In: Theoretical Aspects of Computing - ICTAC 2007. Volume 4711 of LNCS., Springer-Verlag (2007) 124–138
12. Colvin, R., Dongol, B.: A general technique for proving lock-freedom. Sci. Comput. Program. **74**(3) (2009) 143–165
13. Gotsman, A., Cook, B., Parkinson, M., Vafeiadis, V.: Proving that nonblocking algorithms don't block. In: Principles of Programming Languages, ACM (2009) 16–28
14. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems **12**(3) (1990) 463–492
15. Massalin, H., Pu, C.: A lock-free multiprocessor os kernel. SIGOPS Oper. Syst. Rev. **26**(2) (1992) 108
16. Jones, C.B.: Specification and design of (parallel) programs. In: Proceedings of IFIP'83, North-Holland (1983) 321–332
17. Misra, J.: A reduction theorem for concurrent object-oriented programs. In McIver, A., Morgan, C., eds.: Programming methodology. Springer-Verlag New York, Inc., New York, NY, USA (2003) 69–92
18. Moszkowski, B.: Executing Temporal Logic Programs. Cambridge University Press (1986)
19. 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) `www.cms.dmu.ac.uk/~cau/itlhomepage`.
20. 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. Volume II: Systems and Implementation Techniques. Kluwer Academic Publishers, Dordrecht (1998) 13 – 39

21. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free parallel and concurrent garbage collection by mark&sweep. Sci. Comput. Program. **64**(3) (2007) 341–374
22. Balser, M., Bäumler, S., Reif, W., Schellhorn, G.: Interactive Verification of Concurrent Systems using Symbolic Execution. In: Proceedings of 7th International Workshop of Implementation of Logics (IWIL 08). (2008)
23. Balser, M.: Verifying Concurrent System with Symbolic Execution. Shaker Verlag, Germany (2006)
24. Collette, P., Knapp, E.: Logical foundations for compositional verification and development of concurrent programs in unity. In: Lecture Notes of Computer Science. Number 936 (1995) 353 – 367
25. 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. Number 54 in Cambridge Tracts in Theoretical Computer Science. Cambridge University Press (2001)
26. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3) (1994) 872–923
27. Alpern, B., Schneider, F.B.: Recognizing safety and liveness. Distributed Computing **2**(3) (1987)
28. 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`.
29. Abadi, M., Lamport, L.: Conjoining specifications. ACM Transactions on Programming Languages and Systems (1995)
30. Dongol, B.: Formalising progress properties of non-blocking programs. In: Formal Methods and Software Engineering. Volume 4260 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2006) 284–303
31. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis, University of Cambridge (2007)
32. Gao, H., Groote, J.F., Hesselink, W.H.: Lock-free dynamic hash tables with open addressing. Distrib. Comput. **18**(1) (2005) 21–42
33. Amit, D., Rinetzky, N., Reps, T.W., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV. (2007) 477–490
34. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. **24**(3) (2002) 217–298
35. Derrick, J., Schellhorn, G., Wehrheim, H.: Proving linearizabilty via non-atomic refinement. In J. Davies, J.G., ed.: Proceedings of the International Conference on integrated formal methods (iFM) 2007. Volume 4591 of LNCS., Springer (2007) 195–214
36. Owicki, S.S., Gries, D.: An Axiomatic Proof Technique for Parallel Programs I. Acta Inf. **6** (1976) 319–340
37. O'Hearn, P., Reynolds, J., Yang, H.: Local reasoning about programs that alter data structures. Lecture Notes in Computer Science **2142** (2001)