Aspect-Oriented Linearizability Proofs

Thomas A. Henzinger[1], Ali Sezgin[1], and Viktor Vafeiadis[2]

[1] IST Austria {tah,asezgin}@ist.ac.at
[2] MPI-SWS viktor@mpi-sws.org

Abstract. Linearizability of concurrent data structures is usually proved by monolithic simulation arguments relying on identifying the so-called linearization points. Regrettably, such proofs, whether manual or automatic, are often complicated and scale poorly to advanced non-blocking concurrency patterns, such as helping and optimistic updates.

In response, we propose a more modular way of checking linearizability of concurrent queue algorithms that does not involve identifying linearization points. We reduce the task of proving linearizability with respect to the queue specification to establishing four basic properties, each of which can be proved independently by much simpler arguments. As a demonstration of our approach, we verify the Herlihy and Wing queue, an algorithm that is challenging to verify by a simulation proof.

1 Introduction

Linearizability [8] is widely accepted as the standard correctness requirement for concurrent data structure implementations. It amounts to showing that all methods are atomic and obey the high-level sequential specification of the data structure. For example, an unbounded queue must support the following two methods: *enqueue*, which extends the queue by appending one element to its end, and *dequeue*, which removes and returns the first element of the queue.

The standard way to prove that a concurrent queue implementation is linearizable is to prove an invariant which relates the state of the implementation to the state of the specification. A well-established approach (e.g. [1–5, 11, 13–15]) is to identify the linearization points, which when performed by the implementation change the state of the specification, and to then construct a forward or backward simulation.

While for a number of concurrent algorithms, spotting the linearization points may be straightforward (and has even been automated to some extent [15]), in general specifying the linearization points can be very difficult. Due to helping, they can lie in code not syntactically belonging to the thread and operation in question, and can also depend on future behavior. There are numerous examples in the literature, where this is the case. To mention only a few concurrent queues: the Herlihy and Wing queue [8], the optimistic queue [10], the elimination queue [12], the baskets queue [9], the flat-combining queue [6].

```
1: var q.back : int ← 0
2: var q.items : array of val
       ← {NULL, NULL, ...}

3: procedure enq(x : val)
4:     ⟨i ← INC(q.back)⟩          ▷ E1
5:     ⟨q.items[i] ← x⟩           ▷ E2

6: procedure deq() : val
7:   while true do
8:     ⟨range ← q.back − 1⟩       ▷ D1
9:     for i = 0 to range do
10:      ⟨x ← SWAP(q.items[i], NULL)⟩   ▷ D2
11:      if x ≠ NULL then return x
```

Fig. 1. Herlihy and Wing queue [8].

The HW Queue. In this paper, we focus on the Herlihy and Wing queue [8] (henceforth, HW queue for short) that illustrates nicely the difficulties encountered when defining a simulation relation based on linearization points. The code is given in Fig. 1. The queue is represented as a pre-allocated unbounded array, $q.items$, initially filled with NULLs, and a marker, $q.back$, pointing to the end of the used part of the array. Enqueuing an element is done in two steps: the marker to the end of the array is incremented (E_1), thereby reserving a slot for storing the element, and then the element is stored at the reserved slot (E_2). Dequeue is more complex: it reads the marker (D_1), and then searches from the beginning of the array up to the marker to see if it contains a non-NULL element. It removes and returns the first such element it finds (D_2). If no element is found, dequeue starts again afresh. Each of the four statements surrounded by $\langle\rangle$ brackets and annotated by E_i or D_i for $i = 1, 2$ is assumed to execute in isolation.

Consider the following execution fragment, where $\cdot$ denotes context switches between concurrent threads,

$$(t : E_1) \cdot (u : E_1) \cdot (v : D_1, D_2) \cdot (u : E_2) \cdot (t : E_2) \cdot (w : D_1)$$

which have threads t and u executing enqueue instances, v and w executing dequeue instances. At the end of this fragment, v is ready to dequeue the element enqueued by u, and w is ready to dequeue the element enqueued by t. In order to define a simulation relation from this interleaving sequence to a valid sequential queue behavior, where operations happen in isolation, we have to pick the linearization points for the two completed enqueue instances. The difficulty lies in the fact that no matter which statements are chosen as the linearization points for the two enqueue instances, there is always an extension to the fragment inconsistent with the particular choice of linearization points. For instance, if we pick $(t : E_1)$ as the linearization point for t, then the extension

$$(v : D_2, \mathbf{return}) \cdot (z : D_1, D_2, \mathbf{return})$$

requiring u's element be enqueued before that of t's, will be inconsistent. If on the other hand, any statement which makes u linearize before t, then the extension

$$(w : D_2, \mathbf{return}) \cdot (z : D_1, D_2, D_2, \mathbf{return})$$

requiring the reverse order of enqueueing will be inconsistent. This shows not only that finding the correct linearization sequence can be challenging, but also

that the simulation proofs will require to reason about the entire state of the system, as the local state of one thread can affect the linearization of another.

Our Contribution. In our experience, this and similar tricks for reducing synchronization among threads so as to achieve better performance, make concurrent algorithms extremely difficult to reason about when one is constrained to establishing a simulation relation. However, if two methods overlap in time, then the only thing enforced by linearizability is that their effects are observed in *some* and same order by all threads. For instance, in the example given above, the simple answer for the particular ordering between the linearization points of the enqueue instances of t and u, is that it does not matter! As long as enqueue instances overlap, their values can be dequeued in any order.

Building on this main observation, our contribution is to simplify linearizability proofs by modularizing them. We reduce the task of proving linearizability to establishing four relatively simple properties, each of which may be reasoned about independently. In (loose) analogy to aspect-oriented programming, we are proposing "aspect-oriented" linearizability proofs for concurrent queues, where each of these four properties will be proved independently.

So what are these properties? A correct (i.e., linearizable) concurrent queue:
(1) must not allow dequeuing an element that was never enqueued;
(2) it must not allow the same element to be dequeued twice;
(3) it must never reorder enqueued elements; and
(4) it must correctly report whether the queue is empty or not.

Although similar properties were already mentioned by Herlihy and Wing [8], we for the first time prove that suitably formalized versions of these four properties are not only necessary, but also sufficient, conditions for linearizability with respect to the queue specification, at least for what we call *purely-blocking* implementations. This is a rather weak requirement satisfied by all non-blocking methods, as well as by possibly blocking methods, such as HW `deq`() method, whose blocking executions do not modify the global state.

The rest of the paper is structured as follows: §2 recalls the definition of linearizability in terms of execution histories; §3 formalizes the aforementioned four properties, and proves that they are necessary and sufficient conditions for proving linearizability of queues; §4 returns to the HW queue example and presents a detailed manual proof of its correctness; and §5 explains how the bulk of this proof was also performed automatically by an adaptation of Cave [15]. Finally, in §6 we discuss related work, and in §7 we conclude.

2 Technical Background

In this section, we introduce common notations that will be used throughout the paper and recall the definition of linearizability.

Histories, Linearizability. For any function f from A to B and $A' \subseteq A$, let $f(A') \stackrel{\text{def}}{=} \{f(a) \mid a \in A'\}$. Given two sequences x and y, let $x \cdot y$ denote their concatenation, and let $x \sim_{\text{perm}} y$ hold if one is a permutation of the other.

A *data structure* $\mathcal{D}$ is a pair $(D, \Sigma_{\mathcal{D}})$, where D is the *data domain* and $\Sigma_{\mathcal{D}}$ is the *method alphabet*. An *event* of $\mathcal{D}$ is a triple (m, d_i, d_o), for some $m \in \Sigma_{\mathcal{D}}$, $d_1, d_2 \in D$. Intuitively, (m, d_i, d_o) denotes the application of method m with input argument d_i returning the output value d_o. A sequence over events of $\mathcal{D}$ is called a *behavior*. The *semantics* of data structure $\mathcal{D}$ is a set of behaviors, called *legal* behaviors.

Each event $a = (m, d_i, d_o)$ generates two *actions*: the *invocation* of a, written as $inv(a)$, and the *response* of a, written as $res(a)$. We will also use $m_i(d_i)$ and $m_r(d_o)$ to denote the invocation and the response actions, respectively. When a particular method m does not have an input (resp., output) parameter, we will write $(m, \bot, x)$ (resp., $(m, x, \bot)$), and $m_i()$ (resp., $m_r()$) for the corresponding invocation (resp., response) action.

In this paper, a *history* of $\mathcal{D}$ is a sequence of invocation and response actions of $\mathcal{D}$. We will assume the existence of an implicit identifier in each history c that uniquely pairs each invocation with its corresponding response action, if the latter also occurs in c. A history c is *well-formed* if every response action occurs after its associated invocation action in c. We will consider only well-formed histories. An event is *completed* in c, if both of its invocation and response actions occur in c. An event is *pending* in c, if only its invocation occurs in c. We define $remPending(c)$ to be the sub-sequence of c where all pending events have been removed. An event e precedes another event e' in c, written $e \prec_c e'$, if the response of e occurs before the invocation of e' in c. For event e, $Before(e, c)$ denotes the set of all events that precede e in c. Similarly, $After(e, c)$ denotes the set of all events that are preceded by e in c. Formally,

$$Before(e, c) \stackrel{\text{def}}{=} \{e' \mid e' \prec_c e\} \qquad \text{and} \qquad After(e, c) \stackrel{\text{def}}{=} \{e' \mid e \prec_c e'\}.$$

History c is called *complete* if it does not have any pending events. For a possibly incomplete history c, a *completion* of c, written $\hat{c}$, is a (well-formed) complete history such that $\hat{c} = remPending(c \cdot c')$ where c' contains only response events. Let $Compl(c)$ denote the set of all completions of c.

A history is called *sequential* if all invocations in c are immediately followed by their matching responses, with the possible exception of the very last action which can only be the invocation of a pending event. We identify complete sequential histories with behaviors of $\mathcal{D}$ by mapping each consecutive pair of matching actions in the former to its event constructing the latter. A sequential history s is a *linearization* of a history c, if there exists $\hat{c} \in Compl(c)$ such that $\hat{c} \sim_{\text{perm}} s$ and whenever $e \prec_{\hat{c}} e'$ we have $e \prec_s e'$.

Definition 1 (Linearizability [8]). *A set of histories C is linearizable with respect to a data structure $\mathcal{D}$, if for any $c \in C$, there exists a linearization of c which is a legal behavior of $\mathcal{D}$.*

Queues. The method alphabet Σ_Q of a queue is the set $\{\texttt{enq}, \texttt{deq}\}$. We will take the data domain to be the set of natural numbers, $\mathbb{N}$, and a distinguished symbol `NULL` not in $\mathbb{N}$. Events are written as $\texttt{enq}(x)$, short for $(\texttt{enq}, x, \bot)$, and $\texttt{deq}(x)$,

short for $(\texttt{deq}, \bot, x)$. Events with `enq` are called *enqueue* events, and those with `deq` are called *dequeue* events.

Let c be a history. $Enq(c)$ denotes the set of all enqueue events invoked (and not necessarily completed) in c. Similarly, $Deq(c)$ denotes the set of all dequeue events invoked in c. A set $A \subseteq Enq(c) \cup Deq(c)$ is *closed under* $\prec_c$ if $a \in A$ and $b \prec_c a$, then $b \in A$.

For an `enq` event e in c, $Val_c(e)$ denotes the value to be inserted by e in c. Formally, $Val_c(\texttt{enq}(x)) = x$. Similarly, for a completed `deq` event d in c, $Val_c(d)$ denotes the value removed by d in c. Formally, $Val_c(\texttt{deq}(x)) = x$. For a pending `deq` event, $Val_c(\texttt{deq}(x))$ is undefined.

We will use a labelled transition system, LTS_Q, to define the queue semantics. The states of LTS_Q are sequences over $\mathbb{N}$, the initial state is the empty sequence ε. There is a transition from q to q' with action a, written $q \xrightarrow{a} q'$, if (i) $a = \texttt{enq}(x)$ and $q' = q \cdot x$, or (ii) $a = \texttt{deq}(x)$ and $q = x' \cdot q'$, or (iii) $a = \texttt{deq}(\texttt{NULL})$ and $q = q' = \varepsilon$. A queue is *partial* if the last transition (`NULL` returning dequeue event) is not allowed.

A *run* of LTS_Q is an alternating sequence $q_0 l_1 q_1 \ldots l_n q_n$ of states and queue events such that for all $1 \leq i \leq n$, we have $q_{i-1} \xrightarrow{l_i} q_i$. The trace of a run is the sequence $l_1 \ldots l_n$ of the events occurring on the run. A queue behavior b is *legal* iff there is a run of LTS_Q with trace b.

We find it useful to express the queue semantics in an alternative formulation.

Definition 2. *A queue behavior b has a* sequential witness *if there is a total mapping μ_{seq} from $Deq(b)$ to $Enq(b) \cup \{\bot\}$ such that*

- $\mu_{\text{seq}}(d) = e$ *implies* $Val_b(d) = Val_b(e)$,
- $\mu_{\text{seq}}(d) = \bot$ *iff* $Val_b(d) = \texttt{NULL}$,
- $\mu_{\text{seq}}(d) = \mu_{\text{seq}}(d') \neq \bot$ *implies* $d = d'$,
- $e \prec_b e'$ *and there exists* d' *with* $\mu_{\text{seq}}(d') = e'$ *imply* $\mu^{-1}_{\text{seq}}(e) \prec_b d'$,
- $\mu_{\text{seq}}(d) = \bot$ *implies that*
  $|\{e \in Enq(b) \mid e \prec_b d\}| = |\{d' \in Deq(b) \mid d' \prec_b d \wedge \mu_{\text{seq}}(d') \neq \bot\}|$.

Proposition 1. *A queue behavior b is legal iff b has a sequential witness.*

Proof (Sketch). If b is legal, then, by definition, it has a run r in LTS_Q with trace b. Let d be a dequeue event occurring in b. Then there is a transition $q \xrightarrow{d} q'$ in r. If $d = \texttt{deq}(x)$ for some $x \in \mathbb{N}$, then set $\mu_{\text{seq}}(d) = e$ where e is the enqueue event $\texttt{enq}(x)$ which has inserted x into the state sequence. If $d = \texttt{deq}(\texttt{NULL})$, then set $\mu_{\text{seq}}(d) = \bot$. Then, it is easy to check that μ_{seq} satisfies all the conditions of being a sequential witness for b.

For the other direction, let μ_{seq} be a sequential witness for b. We observe that i) an element x is in state q iff an enqueue event $\texttt{enq}(x)$ has happened on the prefix of the run ending at q and the dequeue event with $\mu_{\text{seq}}(d) = e$ has not happened on the same prefix, ii) for any two enqueue events e, e' with $e \prec_b e'$, $Val_b(e)$ occurs in a state before $Val_b(e')$, iii) the relative ordering of inserted elements in a state does not change as long as both are in the state, iv) each

enqueue event inserts exactly one element to the state, v) each dequeue event $\texttt{deq}(x)$ with $x \neq \texttt{NULL}$ removes exactly one element from the state, and vi) the dequeue event $\texttt{deq}(\texttt{NULL})$ does not change the state. Then, by induction on the length of b, we show that b has a run in LTS_Q. □

3 Conditions for Queue Linearizability

3.1 Generic Necessary and Sufficient Conditions

We start by reducing the problem of checking linearizability of a given history, c, with respect to the queue specification to finding a mapping from its dequeue events to its enqueue events satisfying certain conditions. Intuitively, we map each dequeue event to the enqueue event whose value the dequeue removed, or to nothing if the dequeue event returns NULL. We say that the mapping is *safe* if it pairs each deq event with a proper enq event, implying that elements are inserted exactly once and removed at most once. A safe mapping is *ordered* if it additionally respects precedence induced by c. Finally, an ordered mapping is a *linearizability witness* if all NULL returning deq events see at least one state where the queue is logically empty. Below, we formalize these notions.

Definition 3 (Safe Mapping). *A mapping Match from $Deq(c)$ to $Enq(c) \cup \{\bot\}$ is* safe *for c if*
(1) for all $d \in Deq(c)$, if $Match(d) \neq \bot$, then $Val_c(d) = Val_c(Match(d))$;
(2) for all $d \in Deq(c)$, $Match(d) = \bot$ iff $Val_c(d) = \texttt{NULL}$; and
(3) for all $d, d' \in Deq(c)$, if $Match(d) = Match(d') \neq \bot$, then $d = d'$.

Definition 4 (Ordered Mapping). *A safe mapping Match for c is* ordered *if*
(1) for all $d \in Deq(c)$, we have $d \not\prec_c Match(d)$; and
(2) for all $d, d' \in Deq(c)$, if $Match(d) \prec_c Match(d')$, then $d' \not\prec_c d$.

Definition 5 (Linearization Witness). *An ordered mapping Match for c is a* linearization witness *if for any $d \in Deq(c)$ with $Val_c(d) = \texttt{NULL}$, there exists a subset $D' \subseteq Deq(c)$ such that $Match(D')$ is closed under $\prec_c$ and $D' \cap After(d, c) = \emptyset$ and $Before(d, c) \cap Enq(c) \subseteq Match(D')$.*

The main result of this section is stated below.

Theorem 1. *A set of histories C is linearizable with respect to queue iff every $c \in C$ has a completion $\hat{c} \in Compl(c)$ that has a linearization witness.*

Proof. (⇒) If $c \in C$ is linearizable with respect to queue, then there is a linearization s of c which is a legal queue behavior. By Prop. 1, s has a sequential witness μ_{seq}. The mapping μ_{seq} satisfies the conditions of a linearization witness since all $\prec_c$ orderings are preserved in s.

(⇐) Pick a $c \in C$ and let $\hat{c} \in Compl(c)$ be its completion that has a linearization witness *Match*. Let $<$ be some arbitrary total order on the events of $\hat{c}$. We construct the linearization of $\hat{c}$ inductively as follows:

Let c' be the prefix of $\hat{c}$ that has been processed, and let s' be the resulting sequential history. All events in s' are *placed*. Events that are not placed but are pending after c' are called *candidate*. We extend c' until the first response action that happens after c' in $\hat{c}$. Formally, let $c' \cdot c_e \cdot a_r$ be a prefix of $\hat{c}$ such that c_e contains only invocation actions and a_r is a response action. Let A denote the set of all candidate events after $c' \cdot c_e \cdot a_r$. The new s' is obtained by appending some $a \in A$ as the next event if
(1) a is an enqueue event, and there does not exist another enqueue event e such that $Match^{-1}(e) \prec_{\hat{c}} Match^{-1}(a)$ and e is not placed in s'; or
(2) a is a dequeue event with $Val_{\hat{c}}(a) \neq \texttt{NULL}$, $Match(a)$ is placed in s', and there does not exist another dequeue event d such that $Match(d) \prec_{\hat{c}} Match(a)$ and d is not placed in s'; or
(3) a is a dequeue event with $Val_{\hat{c}}(a) = \texttt{NULL}$ and the number of enqueue events in s' is equal to the number of dequeue events d with $Val_{\hat{c}}(d) \neq \texttt{NULL}$ in s'.
In case, where both first and second conditions are satisfied, the candidate element minimal with respect to $<$ is appended to s'. This iteration is repeated until there are no candidate events that satisfy any of the conditions, at which point the inductive step ends with setting c' to $c' \cdot c_e \cdot a_r$. The existence of *Match* guarantees that such a sequence can be constructed. The constructed sequence s has *Match* also as a sequential witness, completing the proof. □

3.2 Necessary and Sufficient Conditions for Complete Histories

We now focus on complete histories, namely ones with no pending events. We observe that their linearizability violations can always be manifested in terms of the dequeued values. Intuitively, the possible violations are:

(VFresh) A dequeue event returning a value not inserted by any enqueue event.
(VRepet) Two dequeue events returning the value inserted by the same enqueue event.
(VOrd) Two ordered dequeue events returning values inserted by enqueue events in the inverse order.
(VWit) A dequeue event returning NULL even though the queue is never logically empty during the execution of the dequeue event.

We have the following result which ties the above violation types to linearizable queues.

Proposition 2. *A complete history c has a linearization which is a legal queue behavior iff it has none of the* VFresh, VRepet, VOrd, VWit *violations.*

Proof (Sketch). First, note that as c has no pending events, $Compl(c) = \{c\}$. If c has a linearization which is a legal queue behavior, then by Theorem 1, c has a linearization witness *Match*, and so none of the violations can happen. As *Match* is safe, (VFresh) and (VRepet) cannot happen; as it is ordered, (VOrd) cannot occur; and as it is a linearization witness, likewise (VWit) cannot happen. Similarly, in the other direction, the absence of all the violations ensures the existence of a linearizability witness. □

We remark that none of the violations mentions the possibility of an element inserted by an enqueue being lost forever. This is intentional, as such histories are ruled out by the following proposition.

Proposition 3. *Given an infinite sequence of complete histories $c_1, c_2, \ldots$ not containing any of the violations above, where for every i, c_i is a prefix of c_{i+1}, and the number of dequeue events in c_i is less than that of c_{i+1}, if c_1 contains an enqueue event* $\texttt{enq}(x)$*, then exists some c_j containing* $\texttt{deq}(x)$*.*

Proof. We prove this by contradiction. If there is no $\texttt{deq}(x)$ event, then $\texttt{enq}(x)$ is always in the queue, and so, from the absence of VWit violations, none of the dequeue events following $\texttt{enq}(x)$ can return `NULL`. Also, since dequeue events cannot return values that were not previously enqueued (VFresh) and cannot return the same value multiple times (VRepet), and since the number of dequeue events is increasing, then there must also be new enqueue events. However, only finitely many of those are not preceded by $\texttt{enq}(x)$ which completes in c_1. This means that eventually one dequeue event has to return an element inserted by $\texttt{enq}(y)$ such that $\texttt{enq}(x) \prec_{c_j} \texttt{enq}(y)$, which is VOrd. □

For checking purposes, we find it useful to re-state the third violation as the following equivalent proof obligation.

(POrd) For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $Val_c(e_1) \neq Val_c(e_2)$, a dequeue event cannot return $Val_c(e_2)$ if $Val_c(e_1)$ is never removed in c.

Thus, we need an invariant which specifies all those executions satisfying the premise of POrd, and prove that such an execution cannot end with a dequeue event (in the sense that no other method is preceded by that dequeue event) returning the value of e_2.

3.3 Necessary and Sufficient Conditions for Purely-Blocking Queues

There is a subtle complication in the statement of Theorem 1. The witness mapping is chosen relative to some completion of the concurrent history under consideration. However, because implementations may become blocked, such completions may actually never be reached. This means that one cannot reason about the correctness of a queue implementation by considering only reachable states. What we would ideally like to do is to claim that if the implementation violates linearizability, then there is a finite complete history of the implementation which has no witness. In other words, if the implementation contains an incomplete history with no witness, then that execution is the prefix of a complete history of the implementation.

Let C be the set of all possible execution histories of a library implementation. We call a library implementation *completable* iff for every history $c \in C$, we have $Compl(c) \cap C \neq \emptyset$. For completable implementations, it suffices to consider only complete executions.

Theorem 2. *A completable queue implementation is linearizable iff all its complete histories have none of the* VFresh, VRepet, VOrd *and* VWit *violations.*

Proof. ($\Rightarrow$) If some complete history has a violation, by Prop. 2, it has no linearization, contradicting the assumption that the implementation is linearizable.

($\Leftarrow$) Consider an arbitrary history c of the implementation. As the implementation is completable, there exists a completion $\hat{c} \in \mathit{Compl}(c)$ that is a valid history of the implementation. From our assumptions, $\hat{c}$ cannot have a violation, and so by Prop. 2, $\hat{c}$ has a linearization, and therefore so does c. □

Since it may not be obvious how to easily prove that an implementation is completable, we introduce the stronger notion of purely-blocking implementations, that is straightforward to check. We say that an implementation is *purely-blocking* when at any reachable state, any pending method, if run in isolation will terminate or its entire execution does not modify the global state.

Proposition 4. *Every purely-blocking implementation is completable.*

Proof. Given a history $c \in C$, we will construct $\hat{c} \in \mathit{Compl}(c) \cap C$. We fix a total order of pending events, and consider them in that order. For a pending method e, if running it in isolation terminates, then extend c with the corresponding response for e. Otherwise, the execution of e does not modify any global state and so can be removed from the history without affecting its realizability. □

We remark that our new notion of purely-blocking is a strictly weaker requirement than the standard non-blocking notions: *obstruction-freedom*, which requires all pending methods to terminate when run in isolation, as well as the stronger notions of lock-freedom and wait-freedom. (See [7] for an in depth exposition of these three notions.)

4 Manually Verifying the Herlihy-Wing Queue

Let us return to the HW queue presented in §1 and prove its correctness manually following our aspect-oriented approach.

First, observe that HW queue is purely-blocking: `enq()` always terminates, and `deq()` can update the global state only by reading $x \neq$ `NULL` at E_2, in which case it immediately terminates. So from Prop. 4 and Theorem 2, it suffices to show that it does not have any of the four violations. The last one, VWit, is trivial as the HW `deq()` never returns `NULL`. So, we are left with three violations whose absence we have to verify: VFresh, VRepet, and VOrd.

Intuitively, there are no VFresh violations because `deq()` can return only a value that has been stored inside the $q.items$ array. The only assignments to $q.items$ are E_1 and D_2: the former can only happen by an `enq`(x), which puts x into the array; the latter assigns `NULL`.

Likewise, there are no VRepet violations because whenever in an arbitrary history two calls to `deq()` return the same x, then at least twice there was an element of the $q.items$ array holding the value x and was updated to `NULL`

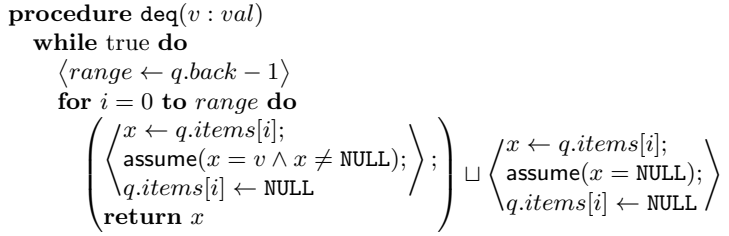

Fig. 2. The HW dequeue method instrumented with the prophecy variable v guessing its return value, where $\sqcup$ stands for non-deterministic choice.

by the SWAP instruction at D_2. Therefore, at least two assignments of the form $q.items[\_] \leftarrow x$ happened; i.e. there were at least two $\texttt{enq}(x)$ events in the history.

We move on to the more challenging third condition, VOrd. We actually consider its equivalent reformulation, POrd. Fix a value v_2 and consider a history c where every method call enqueuing v_2 is preceded by some method call enqueuing some different value v_1 and there are no deq() calls returning v_1 (there may be arbitrarily many concurrent enq() and deq() calls enqueuing or dequeuing other values). The goal is to show that in this history, no deq() return v_2.

Let us suppose there is a dequeue d returning v_2, and try to derive a contradiction. For d to return v_2, it must have read $range \geq i_2$ such that $q.items[i_2] = v_2$. So, d must have read $q.back$ at D_1 after $\texttt{enq}(v_2)$ incremented it at E_1.

Since, $\texttt{enq}(v_1) \prec_c \texttt{enq}(v_2)$, it follows that $\texttt{enq}(v_2)$ will have read a larger value of $q.back$ at E_1 than $\texttt{enq}(v_1)$. So, in particular, once $\texttt{enq}(v_1)$ finishes, the following assertion will hold:

$$\exists i_1 < q.back.\ q.items[i_1] = v_1 \wedge (\forall j < i_1.\ q.items[j] \neq v_2) \qquad (*)$$

Note that since, by assumption, v_1 can never be dequeued, and any later $\texttt{enq}(v_2)$ can only affect the $q.items$ array at indexes larger than i_1, $(*)$ is an invariant.

Given this invariant, however, it is impossible for d to return v_2, as in its loop it will necessarily first have encountered v_1.

5 Automation

As can be seen from our previous informal argument, establishing absence of VFresh and VRepet violations was relatively straightforward, whereas proving POrd was somewhat more involved. Therefore, in this section, we will focus on automating the proof of the third property, POrd. Towards the end of the section, we will discuss the automatic verification of the absence of VWit violations for queue implementations, where deq may return NULL.

Prophetic Instrumentation of Dequeues. Our proof technique relies heavily on instrumenting the deq() function with a prophecy variable 'guessing' the value that will be returned when calling it. Essentially, we construct a method, $\texttt{deq}(v)$,

such that the set of traces of $\bigsqcup_{x \in \mathbb{N} \cup \{\texttt{NULL}\}} \texttt{deq}(x)$ is equal to the set of traces of $\texttt{deq}()$, where $\sqcup$ stands for non-deterministic choice. Figure 2 shows the resulting automatically-generated instrumented definition of $\texttt{deq}(v)$ for the HW queue.

Our implementation of the instrumentation performs a sequence of simple rewrites, each of which does not affect the set of traces produced:

$$\begin{aligned}
\textbf{return } E &\rightsquigarrow \mathsf{assume}(v = E); \textbf{return } E \\
\textbf{if } B \textbf{ then } C \textbf{ else } C' &\rightsquigarrow (\mathsf{assume}(B); C) \sqcup (\mathsf{assume}(\neg B); C') \\
C; \mathsf{assume}(B) &\rightsquigarrow \mathsf{assume}(B); C \qquad \text{provided } \mathit{fv}(B) \subseteq \mathit{Locals} \setminus \mathit{writes}(C) \\
C; (C_1 \sqcup C_2) &\leftrightsquigarrow (C; C_1) \sqcup (C; C_2) \\
(C_1 \sqcup C_2); C &\leftrightsquigarrow (C_1; C) \sqcup (C_2; C)
\end{aligned}$$

In general, the goal of applying these rewrite rules is to bring the introduced $\mathsf{assume}(v = E)$ statements as early as possible without unduly duplicating code.

Proving Absence of VOrd *Violations.* It turns out that our automated technique for proving POrd also establishes absence of VFresh violations as a side-effect. We reduce the problem of proving absence of VFresh and VOrd violations to the problem of checking non-termination of non-deterministic programs with an unbounded number of threads. The reduction exploits the instrumented $\texttt{deq}(v)$ definition: $\texttt{deq}()$ cannot return a result x in an execution precisely if $\texttt{deq}(x)$ cannot terminate in that same execution.

Theorem 3. *A completable queue implementation has no* VFresh *and* VOrd *violations iff for all $n \in \mathbb{N}$ and forall v_1 and v_2 such that $v_1 \neq v_2$, the program*[3]

$$\mathit{Prg} \stackrel{\text{def}}{=} b \leftarrow \mathsf{false};\ (\texttt{deq}(v_2) \parallel \overbrace{C \parallel \ldots \parallel C}^{n \text{ times}})$$

does not terminate, where

$$C \stackrel{\text{def}}{=} (\texttt{enq}(v_1); b \leftarrow \mathsf{true}) \sqcup (\mathsf{assume}(b); \texttt{enq}(v_2)) \sqcup \bigsqcup_{x \neq v_2} \texttt{enq}(x) \sqcup \bigsqcup_{x \neq v_1} \texttt{deq}(x)\,.$$

Proof. ($\Rightarrow$) We argue by contradiction. Consider a terminating history c of *Prg*. If $\texttt{enq}(v_2)$ is not invoked in c, then as there are no VFresh violations, we know that no $\texttt{deq}()$ in c can return v_2, contradicting our assumption that c is a terminating history of *Prg*. Otherwise, if $\texttt{enq}(v_2)$ is invoked in c, then at some earlier point $\mathsf{assume}(b)$ was executed, and since initially b was set to false, this means that $b \leftarrow \mathsf{true}$ was executed and therefore $\texttt{enq}(v_1) \prec_c \texttt{enq}(v_2)$. Consequently, from

[3] For simplicity, we assume that the methods cannot distinguish the thread in which they are running (i.e., they do not use thread-local storage or thread identifiers). Handling thread identifiers properly is not difficult: we have to record a set of thread identifiers that are not currently in use. Before each method invocation, we have to atomically pick and remove an identifier from that set, and on returning from the method, we have to add the current identifier back the set of unused identifiers.

POrd, if there is deq() in c returns v_2, there must be a deq() in c that can be completed to return v_1, contradicting our assumption that c is a terminating history of *Prg*.

($\Leftarrow$) We have two properties to prove. For VFresh, it suffices to consider the restricted parallel context that never chooses to execute the first two of the non-deterministic choices. In this restricted context, namely one that never enqueues v_2, deq(v_2) does not terminate, and so deq() cannot return v_2. For VOrd, consider a history in which every enq(v_2) happens after some enqueue of a different value, say enq(v_1), and in which there is no deq(v_1). Such a history can easily be produced by the unbounded parallel composition of C, and so deq(v_2) also does not terminate, as required. □

To prove non-termination, we essentially prove the partial-correctness Hoare triple, {true} *Prg* {false}. Given a sound program logic, the only way for such a triple to hold is for the program to always diverge.

Implementation within Cave. To prove such triples, we have midly adapted the implementation of Cave [15], a sound but incomplete thread-modular concurrent program verifier that can handle dynamically allocated linked list data structures, fine-grained concurrency. The tool takes as its input a program consisting of some initialization code and a number of concurrent methods, which are all executed in parallel an unbounded number of times each. When successful, it produces a proof in RGSep that the program has no memory errors and none of its assertions are violated at runtime. Internally, it performs RGSep action inference [16] with a rich shape-value abstract domain [14] that can remember invariants of the form that value v_1 is inside a linked list. Cave also has a way of proving linearizability by a brute-force search for linearization points (see [15] for details), but this is not applicable to the HW queue and therefore irrelevant for our purposes.

The main modifications we had to perform to the tool were: (1) to add code that instruments deq() methods with a prophecy argument guessing its return value, thereby generating deq(v); (2) to improve the abstraction function so that it can remember properties of the form $v_2 \notin X$, which are needed to express the ($*$) invariant of the proof in §4; and (3) to add some glue code that constructs the *Prg* verification condition and runs the underlying prover to verify it.

As Cave does not support arrays (it only supports linked lists), we gave the tool a linked-list version of the HW queue, for which it successfully verified that there are no VFresh and VOrd violations.

Showing Absence of VWit *Violations.* Here, we have to show that any dequeue event cannot return empty if it never goes through a state where the queue is logically empty. This in turn means that we have to express non-emptiness using only the actions of the history (and not referring to the linearization point or the gluing invariant which relates the concrete states of the implementation to the abstract states of the queue). For the following let us fix a (complete) concurrent history c and a dequeue of interest d which returns NULL and does not precede any other event in c.

Let c' be some prefix of c and let $e \in Enq(c')$ be a complete enqueue event in c'. We will call e *alive* after c' if there is no completion of c' in which the dequeue event $\texttt{deq}(Val_{c'}(e))$ occurs. Let d_i denote the dequeue event which removes the element inserted by the enqueue event e_i; that is, $d_i = \texttt{deq}(Val_c(e_i))$. A sequence $e_0 e_1 \dots e_n$ of enqueue events in $Enq(c)$ is *covering* for d in c if the following holds:

- e_0 is alive at c' where c' is the maximal prefix of c such that $d \notin Deq(c')$.
- For all $i \in [1, n]$, e_i starts before d completes.
- For all $i \in [1, n]$, we have $e_i \prec_c d_{i-1}$.
- e_n is alive at c.

Note that all d_i must exist by the third condition and that d_n does not exist by the last condition. Then, the sequence is covering for d if d_0 does not start before d starts, and every enqueue event e_i completes before the dequeue event d_{i-1} starts. Intuitively, this means that at every state visited during the execution of d, the queue contains at least one element. The property corresponding to the last violation (VWit) then becomes the following:

(PWit) A dequeue event d cannot return `NULL` if there is a covering for d.

We will actually re-state the same property in a simpler way by making the following observation.

Proposition 5. *There is a covering for d in c iff at every prefix c' of c such that d is running, there is at least one alive enqueue event.*

Then, we can alternatively state PWit as follows:

(PWit′) A dequeue event d cannot return `NULL` if for every prefix c' at which d is pending there exists an alive enqueue event.

Note that, POrd can also be stated in terms of alive enqueue events.

(POrd′) For any enqueue events e_1 and e_2 with $e_1 \prec_c e_2$ and $Val_c(e_1) \neq Val_c(e_2)$, a dequeue event cannot return $Val_c(e_2)$ if e_1 is alive at c.

6 Related Work

Linearizability was first introduced by Herlihy and Wing [8], who also presented the HW queue as an example whose linearizability cannot be proved by a simple forward simulation where each method performs its effects instantaneously at some point during its execution. The problem is, as we have seen, that neither of E_1 or E_2 can be given as the (unique) linearization point of `enq` events, because the way in which two concurrent enqueues are ordered may depend on not-yet-completed concurrent `deq` events. In other words, one cannot simply define a mapping from the concrete HW queue states to the queue specification states. Nevertheless, Herlihy and Wing do not dismiss the linearization point technique completely, as we do, but instead construct a proof where they map concrete states to non-empty sets of specification states.

This mapping of concrete states to non-empty sets of abstract states is closely related to the method of *backward simulations*, employed by a number of manual proof efforts [3, 5, 13], and which Schellhorn et al. [13] recently showed to be a complete proof method for verifying linearizability. Similar to forward simulation proofs, backward simulation proofs, are monolithic in the sense that they prove linearizability directly by one big proof. Sadly, they are also not very intuitive and as a result often difficult to come up with. For instance, although the definition of their backward simulation relation for the HW queue is four lines long, Schellhorn et al. [13] devote two full pages to explain it.

As a result, most work on automatically verifying linearizability (e.g. [2, 14, 15, 1]) has relied on the simpler technique of forward simulations, even though it is known to be incomplete. The programmer is typically required to annotate each method with its linearization points and then the verifier uses some kind of shape analysis that automatically constructs the simulation relation. This approach seems to work well for simple concurrent algorithms such as the Treiber stack and the Michael and Scott queues, where finding the linearization points may be automated by brute-force search [15], but cannot handle more challenging examples such as the ones mentioned in the introduction.

Among this line of work, the most closely related one to this paper is the recent work by Abdulla et al. [1], who verify linearizability of stack and queue algorithms using observer automata that report specification violations such as our VOrd. Their approach, however, still requires users to annotate methods with linearization points, because checker automata are synchronized with the linearization points of the implementation.

We would also like to point out that the use of forward simulations is not limited to automated verifications of linearizability. Several manual verification also used forward simulations (e.g. [4, 3]).

To the best of our knowledge, there exist only two earlier published proofs of the HW queue: (1) the original pencil-and-paper proof by Herlihy and Wing [8], and (2) a mechanized backward simulation proof by Schellhorn et al. [13].

Both proofs are manually constructed. In comparison, our new proof is simpler, more modular, and largely automatically generated.[4] This is largely due to the fact that we have decomposed the goal of proving linearizability into proving four simpler properties, which can be proved independently. This may allow one to adapt the HW queue algorithm, e.g. by checking emptiness of the queue and allowing `deq` to return `NULL`, and affecting only the proof of absence of VWit violations without affecting the correctness arguments of the other properties.

Our violation conditions are arguably closer to what programmers have in mind when discussing concurrent data structures. Informal specifications written by programmers and bug reports do not mention that some method is not linearizable, but rather things like that values were dequeued in the wrong order.

[4] We say 'largely' because we have not yet automated the verification of the absence of VRepet violations, which requires a simple counting argument, nor the (admittedly trivial) proof that the HW queue is purely-blocking. We intend to implement these in the near future.

7 Conclusion

We have presented a new method for checking linearizability of concurrent queues. Instead of searching for the linearization points and doing a monolithic simulation proof, we verify four simple properties whose conjunction is equivalent to linearizability with respect to the atomic queue specification. By decomposing linearizability proofs in this way, we obtained a much simpler correctness proof of the Herlihy and Wing queue [8], and one which can be produced automatically.

We conjecture that our new property-oriented approach to linearizability proofs will be equally applicable to other kinds of concurrent shared data structures, such as stacks, sets, and maps. In the future, we would like to build tools that will automate this kind of reasoning for such data structures.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS'13. pp. 324–338. Springer (2013)
2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: CAV (2007)
3. Colvin, R., Doherty, S., Groves, L.: Verifying concurrent data structures by simulation. ENTCS 137(2), 93–110 (2005)
4. Derrick, J., Schellhorn, G., Wehrheim, H.: Verifying linearisability with potential linearisation points. In: FM'11. pp. 323–337. Springer (2011)
5. Doherty, S., Moir, M.: Nonblocking algorithms and backward simulation. In: Keidar, I. (ed.) DISC. LNCS, vol. 5805, pp. 274–288. Springer (2009)
6. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: SPAA '10. pp. 355–364. ACM (2010)
7. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc. (2008)
8. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. pp. 463–492 (1990)
9. Hoffman, M., Shalev, O., Shavit, N.: The baskets queue. In: OPODIS'07. pp. 401–414. Springer (2007)
10. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. In: DISC '04. pp. 117–131. Springer-Berlin (2004)
11. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: FM '09. pp. 321–337. Springer (2009)
12. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: SPAA '05. pp. 253–262. ACM (2005)
13. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: CAV'12. pp. 243–259. Springer (2012)
14. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI. LNCS, vol. 5403, pp. 335–348. Springer (2009)
15. Vafeiadis, V.: Automatically proving linearizability. In: CAV'10. pp. 450–464. Springer (2010)
16. Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M.V. (eds.) VMCAI. LNCS, vol. 5944, pp. 345–361. Springer (2010)