
ASPECT-ORIENTED LINEARIZABILITY PROOFS*

SOHAM CHAKRABORTY^a, THOMAS A. HENZINGER^b, ALI SEZGIN^c, AND VIKTOR VAFEIADIS^d

^{a,d} MPI-SWS, Kaiserslautern and Saarbrücken, Germany
e-mail address: {sohachak,viktor}@mpi-sws.org

^b IST Austria, Klosterneuburg, Austria
e-mail address: tah@ist.ac.at

^c University of Cambridge Computer Laboratory, Cambridge, U.K.
e-mail address: as2418@cam.ac.uk

ABSTRACT. Linearizability of concurrent data structures is usually proved by monolithic simulation arguments relying on the identification of 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 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 [10] is widely accepted as the standard correctness requirement for concurrent data structure implementations. It amounts to showing that each method provides the illusion that it *executes* atomically at some point after its call and before its return. Typically, what each method is expected to do (atomically) is given in terms of a sequential specification. For instance, 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 show that it is simulated by the idealised atomic queue implementation, which we take to be the specification of the queue. For example, using forward simulation [15], we have to define a relation S relating the state of the implementation to the state of the specification, and

2012 ACM CCS: [Theory of computation]: Semantics and reasoning—Program reasoning—Program verification; [Software and its engineering]: Software organization and properties—Software functional properties—Formal methods—Software verification.

Key words and phrases: Linearizability, Queue, Verification, Herlihy-Wing Queue.

* A preliminary version of this article appeared at CONCUR'13.

to show that (1) the initial states of the implementation and the specification are related by S , and (2) starting from S -related implementation and specification states $(\sigma_{\text{impl}}, \sigma_{\text{spec}})$, if the implementation takes a step and goes to state σ'_{impl} , the specification can also take a matching step (or stutter) and result in some state σ'_{spec} that is S -related to σ'_{impl} . The most important part of these proofs is to decide which of the implementation steps are matched by actual steps of the specification code and which by stuttering moves. For each method of the implementation, the step during its execution that in the simulation proof is matched by the atomic step of the corresponding method of the specification is known as the *linearization point*. A well-established approach (e.g. [1, 2, 3, 4, 5, 14, 17, 18, 19]) is therefore to identify these linearization points, which when performed by the implementation change the state of the specification, and to then construct a suitable 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 [19]), in general specifying the linearization points can be very difficult. For instance, in implementations using a helping mechanism, they can lie in code not syntactically belonging to the thread and operation in question, and can even 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 [10], the optimistic queue [13], the elimination queue [16], the baskets queue [11], the flat-combining queue [7].

The Herlihy and Wing Queue.

<pre> 1: var $q.back : int \leftarrow 0$ 2: var $q.items : \text{array of } val$ $\leftarrow \{\text{NULL}, \text{NULL}, \dots\}$ 3: procedure $\text{enq}(x : val)$ 4: $\langle i \leftarrow \text{INC}(q.back) \rangle \triangleright E_1$ 5: $\langle q.items[i] \leftarrow x \rangle \triangleright E_2$ </pre>		<pre> 6: procedure $\text{deq}() : val$ 7: while true do 8: $\langle range \leftarrow q.back - 1 \rangle \triangleright D_1$ 9: for $i = 0$ to $range$ do 10: $\langle x \leftarrow \text{SWAP}(q.items[i], \text{NULL}) \rangle \triangleright D_2$ 11: if $x \neq \text{NULL}$ then return x </pre>
---	--	--

Figure 1: Herlihy and Wing queue [10].

In this paper, we focus on the Herlihy and Wing queue [10] (henceforth, HW queue for short) that illustrates nicely the difficulties encountered when defining a simulation relation based on linearization points. We recall the code of the queue as given in [10] in Figure 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 atomically.

We now show that verifying this algorithm by finding its linearization points is difficult. 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 choose 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 choose $(t : E_1)$ 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, we choose 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 points 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 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) must not allow the same element to be dequeued twice;
- (3) must not allow elements to be dequeued out of order; and
- (4) must correctly report whether the queue is empty or not.¹

Although similar properties were already mentioned by Herlihy and Wing [10], 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

¹The HW queue trivially satisfies the fourth property as it never reports that the queue is empty.

requirement satisfied by all non-blocking implementations, as well as by possibly blocking implementations, such as HW `deq()` method, whose blocking executions do not modify the global state.

Paper Outline. The rest of the paper is structured as follows: Section 2 recalls the definition of linearizability in terms of execution histories. Section 3 develops an alternative characterization of legal queue behaviors, which is useful for our proofs. Section 4 formalizes the aforementioned four properties, and proves that they are necessary and sufficient conditions for proving linearizability of queues. Section 5 returns to the HW queue example and presents a detailed manual proof of its correctness by checking each of the properties separately. Section 6 shows how the checking of these four properties can be automated by reducing them to non-termination of certain parametric programs. Section 7 explains how we adapted CAVE [19] to prove these parametric programs non-terminating for the case of the HW queue. Finally, in Section 8 we discuss related work, and in Section 9 we conclude.

Differences from the Conference Paper. This article is an extended version of our CONCUR’13 conference paper [8], containing all the proofs of the lemmas and theorems mentioned in the paper. Since the conference, we have also implemented a checker for the VRepet property, and have expanded the discussion of automation in Sections 6 and 7 to cover the verification of VRepet.

2. TECHNICAL BACKGROUND

In this section, we introduce common notations that will be used throughout the paper and recall the definition of 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. We use $x\langle i \rangle$ to refer to the i^{th} element in sequence x , and $x\langle i : j \rangle$ to refer to the subsequence of x containing all elements from position i to j inclusive. We write $x|A$ for the subsequence of x containing only elements in the set A .

Behaviors. 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 quadruple (uid, m, d_i, d_o) , for a unique event identifier, $uid \in \mathbb{N}$, a method $m \in \Sigma_{\mathcal{D}}$, and data elements $d_i, d_o \in D$. Intuitively, (uid, m, d_i, d_o) denotes the application of method m with input argument d_i returning the output value d_o . Throughout the paper, we will assume that the *uid* components of events are globally unique. A duplicate-free 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*.

The method alphabet Σ_Q of a queue is the set $\{\text{enq}, \text{deq}\}$. 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 $\text{enq}^{uid}(x)$, short for $(uid, \text{enq}, x, \perp)$, and $\text{deq}^{uid}(x)$, short for $(uid, \text{deq}, \perp, x)$. For conciseness, we will often also omit the *uid* superscripts. Events with `enq` are called *enqueue* events, and those with `deq` are called *dequeue* events. We use `Enq` and `Deq` to denote all enqueue and dequeue events, respectively.

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) $a = \text{enq}(x)$ and $q' = q \cdot x$, or (ii) $a = \text{deq}(x)$ and $q = x \cdot q'$, or (iii) $a = \text{deq}(\text{NULL})$ and $q = q' = \varepsilon$.

A run of LTS_Q is an alternating sequence $q_0 l_1 q_1 \dots l_n q_n$ 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 $l_1 \dots l_n$ of the events occurring on the run. A queue behavior b is *legal* iff there is a run of LTS_Q with trace b . In what follows, we will consider only legal queue behaviors, and hence usually omit *legal*, unless explicitly stated otherwise. Let \mathcal{Q} denote the set of all (legal) queue behaviors.

Histories and Linearizability. Each event $a = (\text{uid}, m, d_i, d_o)$ generates two *actions*: the *invocation* of a , written as $\text{inv}(a)$, and the *response* of a , written as $\text{res}(a)$. We will also use $m_i^{\text{uid}}(d_i)$ and $m_r^{\text{uid}}(d_o)$ 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_i^{uid} (resp., m_r^{uid}) for the corresponding invocation (resp., response) action. We will also often omit the superscripts, when they are not important.

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 $\text{remPending}(c)$ 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 , $\text{Before}(e, c)$ denotes the set of all events that precede e in c . Similarly, $\text{After}(e, c)$ denotes the set of all events that are preceded by e in c . Formally,

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

A set of events A is *closed under* \prec_c iff whenever $a \in A$ and $b \prec_c a$, then $b \in A$.

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 $\hat{c} = \text{remPending}(c \cdot c')$ where c' contains only response actions. Let $\text{Compl}(c)$ denote the set of all completions of c .

A history is called *sequential* if all invocations in it 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 $\hat{c} \in \text{Compl}(c)$ such that $\hat{c} \sim_{\text{perm}} s$ and whenever $e \prec_{\hat{c}} e'$ we have $e \prec_s e'$.

Definition 2.1 (Linearizability [10]). A history c is linearizable with respect to a data structure \mathcal{D} if there exists a linearization of c that is a legal behavior of \mathcal{D} . A set of histories C is linearizable with respect to \mathcal{D} if every $c \in C$ is linearizable with respect to \mathcal{D} .

An *execution trace* is a sequence of instruction labels coupled with thread identifiers executing the instruction. For instance, $(t : i)$ denotes the execution of instruction with the unique label i by thread t . An instruction label is the *entry* point of method m , written $\text{enter}(m)$, if it is the label of the first instruction of m . Similarly, an instruction label is

an *exit* point of m , written $exit(m)$, if it is the label of an instruction that completes the execution of m . Each execution trace τ induces a history $h(\tau)$ which is obtained by replacing each $(t : enter(m))$ with $m_i^{t_{uid}}(d_i)$, each $(t : exit(m))$ with $m_r^{t_{uid}}(d_o)$, and removing the remaining symbols. We assume that states of an execution trace contain enough information to deduce the values of d_i and d_o associated with each entry and exit point. To illustrate this definition, consider the following execution trace from the introduction:

$(t : E_1) \cdot (u : E_1) \cdot (v : D_1, D_2) \cdot (u : E_2) \cdot (t : E_2) \cdot (w : D_1) \cdot (v : D_2, \mathbf{return}) \cdot (z : D_1, D_2, \mathbf{return})$.

The history corresponding to this trace is:

$$\mathbf{enq}_i^t(x) \cdot \mathbf{enq}_i^u(y) \cdot \mathbf{deq}_i^v \cdot \mathbf{deq}_i^w \cdot \mathbf{deq}_r^v(x) \cdot \mathbf{deq}_i^z \cdot \mathbf{deq}_r^z(y)$$

where we have used the thread identifiers without subscripts as unique event identifiers. After completing the history with responses \mathbf{enq}_r^t and \mathbf{enq}_r^u of the pending enqueues, and removing the pending invocation \mathbf{deq}_i^w , the history may be linearized as follows:

$$\mathbf{enq}_r^t(x) \cdot \mathbf{enq}_r^t \cdot \mathbf{enq}_i^u(y) \cdot \mathbf{enq}_r^u \cdot \mathbf{deq}_i^v \cdot \mathbf{deq}_r^v(x) \cdot \mathbf{deq}_i^z \cdot \mathbf{deq}_r^z(y)$$

and corresponds to the (legal) behavior $\mathbf{enq}^t(x) \cdot \mathbf{enq}^u(y) \cdot \mathbf{deq}^v(x) \cdot \mathbf{deq}^z(y)$. An execution trace is *complete* if its induced history is complete. An implementation is identified with the set of execution traces it generates. When clear from the context, we will refer to the induced history of an execution trace as a history of the implementation.

3. ALTERNATIVE CHARACTERIZATION OF LEGAL QUEUE BEHAVIOURS

We start with some terminology. 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 . When c is a complete history, we define the value of an event e , written $Val_c(e)$, to be the value enqueued or dequeued by that event.

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

Definition 3.1. A queue behavior b has a *sequential witness* if there is a total mapping μ_{seq} from $Deq(b)$ to $Enq(b) \cup \{\perp\}$ such that

- (i) $\mu_{\text{seq}}(d) = e$ implies $Val_b(d) = Val_b(e)$,
- (ii) $\mu_{\text{seq}}(d) = \perp$ iff $Val_b(d) = \mathbf{NULL}$,
- (iii) $\mu_{\text{seq}}(d) = \mu_{\text{seq}}(d') \neq \perp$ implies $d = d'$,
- (iv) $\mu_{\text{seq}}(d) = e$ implies $e \prec_b d$,
- (v) $e \prec_b \mu_{\text{seq}}(d')$ implies $\mu_{\text{seq}}^{-1}(e) \prec_b d'$,
- (vi) $\mu_{\text{seq}}(d) = \perp$ implies $|\{e \in Enq(b) \mid e \prec_b d\}| = |\{d' \in Deq(b) \mid d' \prec_b d \wedge \mu_{\text{seq}}(d') \neq \perp\}|$.

To illustrate this definition, consider the following legal queue behavior:

$$b \stackrel{\text{def}}{=} \mathbf{enq}^t(x) \cdot \mathbf{enq}^u(y) \cdot \mathbf{deq}^v(x) \cdot \mathbf{deq}^z(y) \cdot \mathbf{deq}^w(\mathbf{NULL})$$

We can pick μ_{seq} such that $\mu_{\text{seq}}(v) = t$, $\mu_{\text{seq}}(z) = u$ and $\mu_{\text{seq}}(w) = \perp$. The constraints of Definition 3.1 are satisfied because:

- $t \prec_b u$ implies $\mu_{\text{seq}}^{-1}(t) = v \prec_b z = \mu_{\text{seq}}^{-1}(u)$.
- $|\{e \in Enq(b) \mid e \prec_b w\}| = |\{t, u\}| = |\{v, z\}| = |\{d' \in Deq(b) \mid d' \prec_b w \wedge \mu_{\text{seq}}(d') \neq \perp\}|$.

To show that a behavior is legal iff it has a sequential witness, we need a number of auxiliary definitions and lemmas.

We say that two queue behaviors b_1 and b_2 are *observationally equivalent*, written $b_1 \equiv_{obs} b_2$, if the sequences of enqueue events and those of dequeue events agree in both behaviors. Formally, $b_1 \equiv_{obs} b_2$ iff $b_1|Enq = b_2|Enq$ and $b_1|Deq = b_2|Deq$.

We define a special subset of queue behaviors, the *canonical* subset \mathcal{Q}_c , in which each enqueue event is immediately followed by its matching dequeue event, in case it exists. Formally, the canonical queue behaviors are given by the following regular expression:

$$\mathcal{Q}_c \stackrel{\text{def}}{=} ((\text{deq}(\text{NULL}))^* \cdot \sum_{x \in \mathbb{N}} \text{enq}(x) \cdot \text{deq}(x))^* \cdot (\text{deq}(\text{NULL}))^* \cdot (\sum_{x \in \mathbb{N}} \text{enq}(x))^*$$

A run r of $\text{LTS}_{\mathcal{Q}}$ is called *canonical* if the trace of r is canonical.

Note that every canonical behavior is legal. Consider a canonical behavior $b \in \mathcal{Q}_c$ and split it into $b = b' \cdot \text{enq}(v_1)\text{enq}(v_2) \dots \text{enq}(v_n)$ such that b' does not end with an enq . This behavior can be generated by the following run of the $\text{LTS}_{\mathcal{Q}}$.

$$\begin{aligned} & ((\epsilon \text{ deq}(\text{NULL}))^* \epsilon \sum_{x \in \mathbb{N}} \text{enq}(x) x \text{ deq}(x))^* (\epsilon \text{ deq}(\text{NULL}))^* \epsilon \\ & \text{enq}(v_1)v_1 \text{enq}(v_2)(v_1v_2) \cdots (v_1 \dots v_{n-1}) \text{enq}(v_n)(v_1 \dots v_{n-1}v_n) \end{aligned}$$

As the following result shows, canonical queue behaviors represent all legal behaviors, up to observational equivalence.

Lemma 3.2. *Let $b \in \mathcal{Q}$ be a legal queue behavior. Then there exists a canonical behavior $b_c \in \mathcal{Q}_c$ such that $b \equiv_{obs} b_c$.*

Proof. By induction on the length of b . The base case, where $b = \epsilon$ trivially satisfies the condition since $\epsilon \in \mathcal{Q}_c$. Now assume the claim holds for b , and we have to prove it for $b \cdot a$. By the induction hypothesis, there is a canonical behavior $b_c \equiv_{obs} b$. We observe that since b_c and b are legal and $b_c \equiv_{obs} b$, their runs end in the same final state. Therefore, the fact that $b \cdot a$ is legal implies that $b_c \cdot a$ is also legal. We proceed by a case analysis of a .

- $a = \text{enq}(x)$. Then $b_c \cdot \text{enq}(x)$ is trivially also canonical.
- $a = \text{deq}(\text{NULL})$. We know that b_c cannot end in an enqueue event, or else the queue would not be empty. Therefore $b_c \cdot \text{deq}(\text{NULL})$ is canonical.
- $a = \text{deq}(y)$, with $y \neq \text{NULL}$. We know that b_c must be of the form $b_c^1 \cdot \text{enq}(y) \cdot b_c^2$ where b_c^1 does not end in a enqueue event and b_c^2 contains only enqueue events. Then, we take the behavior $b_c^1 \cdot \text{enq}(y) \cdot \text{deq}(y) \cdot b_c^2$, which is both canonical and observationally equivalent to $b \cdot \text{deq}(y)$. \square

Moreover, canonical behaviors have a straightforward sequential witness.

Lemma 3.3. *Every canonical behavior $b \in \mathcal{Q}_c$ has a sequential witness μ_b .*

Proof. Let b be a canonical behavior. We construct μ_b by mapping all $\text{deq}(\text{NULL})$ to \perp , and each $\text{deq}(x)$ with $x \in \mathbb{N}$ to its immediate predecessor. By the definition of canonical behavior, each $\text{deq}(x)$ in b is immediately preceded by $\text{enq}(x)$. Thus, the first four conditions are trivially satisfied. If $\text{enq}(y) \prec_b \text{enq}(x)$ and $\text{deq}(x)$ is in b , then by the definition of canonical behavior, we must have

$$b = b_1 \cdot \text{enq}(y) \cdot \text{deq}(y) \cdot b_2 \cdot \text{enq}(x) \cdot \text{deq}(x) \cdot b_3$$

for some sequences b_1, b_2, b_3 . This implies that condition (v) is satisfied. Finally, if $b = b_1 \cdot \text{deq}(\text{NULL}) \cdot b_2$, consider the sequence b'_1 obtained by projecting out all $\text{deq}(\text{NULL})$ events

from b_1 . That is,

$$b'_1 \stackrel{\text{def}}{=} b_1 | (\text{Enq} \cup \text{Deq} \setminus \{\text{deq}(\text{NULL})\})$$

Then, by the definition of canonical behavior, we have $b'_1 \in (\sum_{x \in \mathbb{N}} \text{enq}(x) \cdot \text{deq}(x))^*$. In other words, b'_1 has an equal number of **enq** and **deq** symbols, such that by construction $\mu_b(\text{deq}(x)) = \text{enq}(x) \neq \perp$. This implies that condition (vi) is satisfied. \square

Lemma 3.4. *If $b = b_1 \cdot \text{deq}(x) \cdot b_2$ is a legal queue behavior, then $\text{enq}(x) \prec_b \text{deq}(x)$.*

Proof. By the definition of LTS_Q , $\text{deq}(x)$ can happen at a state q if $q = x \cdot q'$ for some sequence q' . Again by definition, all runs of LTS_Q reaching q must have a transition with label $\text{enq}(x)$; otherwise, x cannot occur in q . Since all legal behaviors have a corresponding run, $\text{enq}(x) \prec_b \text{deq}(x)$ must hold. \square

Next, we show that observationally equivalent legal queue behaviors cannot reorder their $\text{deq}(\text{NULL})$ events.

Lemma 3.5. *If b and b' are observationally equivalent legal queue behaviors, then $b\langle i \rangle = \text{deq}(\text{NULL})$ iff $b'\langle i \rangle = \text{deq}(\text{NULL})$.*

Proof. Since \equiv_{obs} is a symmetric relation, we prove only one direction. (\Rightarrow) Consider the subsequences $b_e = b\langle 1 : i - 1 \rangle | \text{Enq}$, $b_d = b\langle 1 : i - 1 \rangle | \text{Deq}$, and their duals b'_e and b'_d for b' . Note that each enqueue event increases by one the length of the sequence representing the state, each dequeue event decreases by one the length of the sequence representing the state, and $\text{deq}(\text{NULL})$ can only happen when the length of the sequence is zero ($q = \varepsilon$). Then, the number of enqueue events in b_e and the number of non-NULL dequeue events in b_d must be equal; let us call it k .

Assume first that b_d is a proper prefix of b'_d . This implies that b'_e is a proper prefix of b_e . The $(|b_d| + 1)^{\text{th}}$ symbol in b'_d is $\text{deq}(\text{NULL})$ because $b \equiv_{\text{obs}} b'$. Then, the number of non-NULL dequeue events preceding this $\text{deq}(\text{NULL})$ is k , but the number of enqueue events preceding it, contained in b'_e , which is a proper prefix of b_e , is strictly less than k . This contradicts the assumption that b' is a legal queue behavior. The case where b_e is a proper prefix of b'_e follows a similar argument.

Now assume that $b'_d = b_d$ and $b'_e = b_e$. Further assume $b'\langle i \rangle = \text{enq}(x)$ for some $x \in \mathbb{N}$. Because $b \equiv_{\text{obs}} b'$, the next dequeue event in b' is necessarily $\text{deq}(\text{NULL})$. This, however, contradicts that the fact that b' is legal, because in a legal behavior $\text{deq}(\text{NULL})$ cannot immediately follow an enqueue event. Therefore $b'\langle i \rangle = \text{deq}(y)$ for some y and because $b \equiv_{\text{obs}} b'$, we have that $y = \text{NULL}$, as required. \square

Next, we show that given two observationally equivalent behaviors and a sequential witness for the first behavior, we can build a sequential witness for the other.

Lemma 3.6. *Let $b \equiv_{\text{obs}} b'$ and μ_b be a sequential witness for b . Then, there exists a sequential witness for b' .*

Proof. Let π denote a permutation from b to b' such that $\text{deq}(\text{NULL})$ events are not shuffled. That is, $\pi(i) = j$ means that $b\langle i \rangle = b'\langle j \rangle$ and whenever $b\langle i \rangle = \text{deq}(\text{NULL})$, we set $\pi(i) = i$. By the definition of obs-equivalence and Lemma 3.5, this permutation is well-defined. Note that because $b \equiv_{\text{obs}} b'$, the ordering among dequeue events is preserved by π . That is, if $b\langle i \rangle, b\langle j \rangle \in \text{Deq}$ and $i < j$, then $\pi(i) < \pi(j)$. The same holds for the ordering among enqueue events. We now pick the mapping $\mu(i) \stackrel{\text{def}}{=} \pi(\mu_b(\pi^{-1}(i)))$ and show that it is a sequential witness for b' . Conditions (i) and (ii) are satisfied by construction. Condition

(iii) is satisfied because π is a bijection. Condition (iv) is satisfied by Lemma 3.4 and by the construction of μ . Condition (v) is satisfied by because π is a bijection and preserves the ordering between dequeues and enqueues. Condition (vi) is satisfied by Lemma 3.5. \square

Lemma 3.7. *Let b be a queue behavior with a sequential witness μ .*

- (1) *Let d and e be dequeue and enqueue events such that $\mu(d) = e$, and let b' be the behavior obtained after removing both d and e from b . Then, the restriction of μ to b' is a sequential witness for b' .*
- (2) *Let $d = \mathbf{deq}(\text{NULL})$, and b' by obtained by removing d from b . Then, the restriction of μ to b' is a sequential witness for b' .*

Proof. In both cases, let us denote the restriction of μ on b' with μ' ; we have to show that μ' satisfies the six conditions of a sequential witness.

(1) Since μ' is a restriction, it satisfies conditions (i), (ii) and (iii). Observe that for any two events e_1 and e_2 in b' , we have $e_1 \prec_b e_2$ iff $e_1 \prec_{b'} e_2$. This implies that μ' satisfies conditions (iv) and (v). Finally, we have to show that there cannot be a dequeue event $d' = \mathbf{deq}(\text{NULL})$ such that $e \prec_b d' \prec_b d$. Assume the contrary, then since the number of non-NULL dequeue events and the number of enqueue events preceding d' must be equal, there must be a dequeue event $d_y = \mathbf{deq}(y)$ whose matching $e_y = \mathbf{enq}(y)$ comes after d' . This implies that $d_y \prec_b d' \prec_b e_y$ and $\mu(d_y) = e_y$, contradicting condition (iii). Thus, condition (vi) is also satisfied by μ' .

(2) As in the previous case, conditions (i) to (v) are satisfied by μ' . The condition (vi) is satisfied because removing d' does not affect the cardinality of either set; thus, if $d' = \mathbf{deq}(\text{NULL})$ is in b' , then the number of enqueue events and non-NULL dequeue events that precede d' in b' is the same as those that precede d' in b . \square

Lemma 3.8. *Let b be a queue behavior and let μ be a sequential witness for b . Then, there exists a canonical behavior b_c such that $b \equiv_{\text{obs}} b_c$ and for all i , $b\langle i \rangle = \mathbf{deq}(\text{NULL})$ iff $b_c\langle i \rangle = \mathbf{deq}(\text{NULL})$.*

Proof. Let $\text{can}(b, \mu)$ denote the canonical behavior of b whose sequential witness is μ . We will prove, by induction on the length of b , that can is a well-defined total function.

For the base, consider all sequences b of length 1 or less which have a sequential witness.

- If $b = \varepsilon$, then the empty mapping is the only sequential witness for b ; by definition, b is a canonical behavior. The second condition is vacuously satisfied.
- If $b = \mathbf{deq}(\text{NULL})$, then μ which maps $\mathbf{deq}(\text{NULL})$ to \perp is the only sequential witness for b ; by definition, b is a canonical behavior. Since $b_c = b$, the second condition is satisfied.
- If $b = \mathbf{enq}(x)$ for some $x \in \mathbb{N}$, then the empty mapping is the only sequential witness for b ; by definition, b is a canonical behavior. Since there is no $\mathbf{deq}(\text{NULL})$ event, the second condition is vacuously satisfied.

Observe that the sequence $b = \mathbf{deq}(x)$ of length 1 cannot have a sequential witness, because any sequential witness has to map $\mathbf{deq}(x)$ to a matching enqueue which does not exist.

Assume that the claim holds for all sequences of length k or less. Let b be a sequence of length $k + 1$ and μ be a sequential witness for b . Consider the two sub-sequences of b , $b_d = b|_{\mathbf{Deq}}$ and $b_e = b|_{\mathbf{Enq}}$, with lengths n_d and n_e , respectively. Observe that b is an interleaving of b_d and b_e . In particular, $b\langle 1 \rangle$ is either $b_d\langle 1 \rangle$ or $b_e\langle 1 \rangle$. We will do a case analysis on the possible values for $d = b_d\langle 1 \rangle$.

- $d = \mathbf{deq}(\text{NULL})$. Then, we set $b_c = \text{can}(b, \mu) = d \cdot \text{can}(b', \mu')$, with b' obtained by removing the first $\mathbf{deq}(\text{NULL})$ from b (note that this is d) and μ' obtained by restricting μ to b' . By

Lemma 3.7, μ' is a sequential witness for b' . By inductive hypothesis, $b'_c = \text{can}(b', \mu')$ is a canonical behavior observationally equivalent to b' . Since b'_c is a canonical behavior, so is $b_c = d \cdot b'_c = \text{deq}(\text{NULL}) \cdot b'_c$. Since $b'_c \equiv_{\text{obs}} b'$, we have $b_c|_{\text{Deq}} = d \cdot b'_c|_{\text{Deq}} = b_d = b|_{\text{Deq}}$, $b_c|_{\text{Enq}} = b_e = b|_{\text{Enq}}$. Thus, $b_c \equiv_{\text{obs}} b$. The second condition is satisfied, because both b and b_c have $\text{deq}(\text{NULL})$ in their first position and b'_c preserves the positions of NULL -dequeue events by inductive hypothesis.

- $d = \text{deq}(x)$ for some $x \in \mathbb{N}$. By the assumption that μ is a sequential witness for b implies that there exists $e = \text{enq}(x)$ such that $\mu(d) = e$ (conditions (i) and (ii)) and $d \prec_b e$ (condition (iv)). Then, $e = b_e\langle 1 \rangle = \text{enq}(x)$ must hold. Assume contrary, that is $b_e\langle 1 \rangle = \text{enq}(y)$ for some $y \neq x$. As noted above, $b\langle 1 \rangle$ is either $b_d\langle 1 \rangle$ or $b_e\langle 1 \rangle$. If the former, then $d \prec_b e$ cannot hold since d is minimal with respect to \prec_b , violating condition (iv) which contradicts the assumption that μ is a sequential witness for b . If the latter, that is $e' = b\langle 1 \rangle = b_e\langle 1 \rangle = \text{enq}(y)$, then $e' \prec_b e$, and either there is no $d' = \text{deq}(y)$ or if it exists, $d \prec_b d'$, violating condition (v) which contradicts the assumption that μ is a sequential witness for b . Thus, $e = b_e\langle 1 \rangle$. We set $b_c = \text{can}(b, \mu) = e \cdot d \cdot \text{can}(b', \mu')$, with b' obtained by removing d and e from b and μ' to be the restriction of μ on b' . By Lemma 3.7, μ' is a sequential witness for b' . By inductive hypothesis, $b'_c = \text{can}(b', \mu')$ is a canonical behavior obs-equivalent to b' . Since b'_c is a canonical behavior, so is $b_c = e \cdot d \cdot b'_c = \text{enq}(x) \cdot \text{deq}(x) \cdot b'_c$. Finally, since $b'_c \equiv_{\text{obs}} b'$, we have $b_c|_{\text{Deq}} = d \cdot b'_c|_{\text{Deq}} = b_d = b|_{\text{Deq}}$ and $b_c|_{\text{Enq}} = e \cdot b'_c|_{\text{Enq}} = b_e = b|_{\text{Enq}}$. Thus, $b_c \equiv_{\text{obs}} b$. By the proof of Lemma 3.7, we know that for any $d' = \text{deq}(\text{NULL})$ either both d and e precede it in b or neither does. Since d is the first event in b_d , the latter cannot happen; i.e. $d \prec_b d'$ and $e \prec_b d'$. This implies that the position of d' is the same in b_c and $e \cdot d \cdot b'_c$ by the inductive hypothesis. Thus the second condition is satisfied. \square

Lemma 3.9. *Let b_c be a canonical queue behavior. Let b be a queue behavior such that $b \equiv_{\text{obs}} b_c$, for every $\text{deq}(x)$ in b there is $\text{enq}(x) \prec_b \text{deq}(x)$, and for every i , $b\langle i \rangle = \text{deq}(\text{NULL})$ iff $b_c\langle i \rangle = \text{deq}(\text{NULL})$. Then, b is legal.*

Proof. We prove by induction on the length of b that b has a run in LTS_Q . The base case where $b = \varepsilon$ is trivial. Assume that the claim holds for all sequences of length k or less. Let b be a sequence of length $k + 1$. By the inductive hypothesis, there is a run r in LTS_Q with trace $b\langle 1 : k \rangle$. Let q denote the state reached after this run. It is enough to show that there is a transition in LTS_Q of the form $q \xrightarrow{b\langle k+1 \rangle} q'$, for some q' . We do a case analysis on $b\langle k + 1 \rangle$.

- $b\langle k + 1 \rangle = \text{enq}(x)$. Then the desired transition is $q \xrightarrow{\text{enq}(x)} q \cdot x = q'$.
- $b\langle k + 1 \rangle = \text{deq}(x) = d$. By the assumption on b , $e = \text{enq}(x) \prec_b d$. By observational equivalence to b_c , if $d = (b|_{\text{Deq}} \setminus \{\text{deq}(\text{NULL})\})\langle i \rangle$ for some i , then $e = b|_{\text{Enq}}\langle i \rangle$. Together they imply that there are exactly $i - 1$ many non- NULL dequeue events and at least i many enqueue events that precede d in b . This in turn implies that q must be of the form $x \cdot q'$. Then the desired transition is $x \cdot q' \xrightarrow{\text{deq}(x)} q'$.
- $b\langle k + 1 \rangle = \text{deq}(\text{NULL}) = d$. By the assumption on b and b_c , we have $b_c\langle k + 1 \rangle = \text{deq}(\text{NULL})$. This implies that the number of enqueue events that occur in $b_c\langle 1 : k \rangle$ is equal to the number of non- NULL dequeue events in $b_c\langle 1 : k \rangle$. Since $b \equiv_{\text{obs}} b_c$, for any dequeue event d' we have $d' \prec_{b_c} d$ iff $d' \prec_b d$. These in turn imply that for any enqueue event e we have $e \prec_{b_c} d$ iff $e \prec_b d$. Overall, we then have $q = \varepsilon$ and $\varepsilon \xrightarrow{\text{deq}(\text{NULL})} \varepsilon = q'$ is the desired transition. \square

Theorem 3.10. *A queue behavior b is legal iff b has a sequential witness.*

Proof. (\Rightarrow) Let b be a legal queue behavior. By Lemma 3.2, there is a canonical behavior b_c such that $b_c \equiv_{obs} b$. By Lemma 3.3, b_c has a sequential witness. By Lemma 3.6, b has a sequential witness.

(\Leftarrow) Let b be a queue behavior and μ be a sequential witness for b . By Lemma 3.8 and Lemma 3.9, b is legal. \square

4. CONDITIONS FOR QUEUE LINEARIZABILITY

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 an **enq** event such that the value removed by the former is inserted by the latter, implying that elements are inserted exactly once and removed at most once. A safe mapping is *ordered* if it additionally respects the ordering of events in c . Finally, an ordered mapping is a *linearization witness* if all NULL returning **deq** events see at least one state where the queue is logically empty. Below, we formalize these notions.

Definition 4.1 (Safe Mapping). A total mapping $Match$ from $Deq(c)$ to $Enq(c) \cup \{\perp\}$ is *safe* for complete history c if

- (1) for all $d \in Deq(c)$, if $Match(d) \neq \perp$, then $Val_c(d) = Val_c(Match(d))$;
- (2) for all $d \in Deq(c)$, $Match(d) = \perp$ iff $Val_c(d) = \text{NULL}$; and
- (3) for all $d, d' \in Deq(c)$, if $Match(d) = Match(d') \neq \perp$, then $d = d'$.

Definition 4.2 (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 $e \in Enq(c)$ and $d' \in Deq(c)$, if $e \prec_c Match(d')$, then there exists $d \in Deq(c)$ such that $e = Match(d)$ and $d' \not\prec_c d$.

Intuitively, the first condition states that an enqueue event cannot start after the completion of the dequeue event that removed the value inserted by the former. The second condition states that if two enqueue events e and e' are ordered such that $e \prec_c e'$ and the value inserted by e' is removed by some d' , then there must exist a dequeue event d removing what e has inserted and d' cannot complete before d starts.

Let c be a complete history and $Match$ be ordered for c . Let $d_\perp \in Deq(c)$ be a dequeue event returning NULL; that is, $Val_c(d_\perp) = \text{NULL}$. Define $Bad(c, d_\perp) \subseteq Enq(c)$ as the smallest set consisting of all enqueue events e in c such that either if the matching dequeue d for e exists (i.e. $Match(d) = e$), then d is after d_\perp , or there is another e' in $Bad(c, d_\perp)$ which precedes either e or the matching dequeue event d of e . Formally, the definition is given inductively as follows:

$$\begin{aligned} Bad_0(c, d_\perp) &= \{e \in Enq(c) \mid d_\perp \prec_c e \vee \forall d \in Deq(c). Match(d) = e \Rightarrow d_\perp \prec_c d\} \\ Bad_{i+1}(c, d_\perp) &= \{e \in Enq(c) \mid \exists e_i \in Bad_i(c, d_\perp). e_i \prec_c e \\ &\quad \vee \exists d \in Deq(c). Match(d) = e \wedge e_i \prec_c d\} \end{aligned}$$

with $Bad(c, d_\perp) = \cup_{i \in \mathbb{N}} Bad_i(c, d_\perp)$.

Intuitively, the set $Bad(c, d_\perp)$ contains all enqueue events after the completion of which d_\perp cannot observe an empty queue. In other words, if $e \in Bad(c, d_\perp)$ and if e completes before d_\perp does, then the state of the queue is guaranteed to be non-empty after e completes until the completion of d_\perp .

Definition 4.3 (Linearization Witness). An ordered mapping $Match$ for c is a *linearization witness* if for any $d \in Deq(c)$ with $Val_c(d) = \text{NULL}$, we have $Bad(c, d) \cap Before(c, d) = \emptyset$.

In the proofs that follow, we sometimes use the following result to prove that a given ordered mapping is a linearization witness.

Lemma 4.4. *Let c be a complete history, $Match$ be an ordered mapping for c and $d_\perp \in Deq(c)$ be such that $Match(d_\perp) = \perp$. Then, $Bad(c, d_\perp) \cap Before(c, d_\perp) = \emptyset$ iff there exist subsets $D_{d_\perp} \subseteq Deq(c)$ and $E_{d_\perp} \subseteq Enq(c)$ such that $(D_{d_\perp} \cup E_{d_\perp}) \cap After(c, d_\perp) = \emptyset$, $D_{d_\perp} \cup E_{d_\perp}$ is closed under \prec_c , and $Before(c, d_\perp) \cap Enq(c) \subseteq E_{d_\perp} \subseteq Match(D_{d_\perp})$.*

Proof.

(\Rightarrow) Assume that $Bad(c, d_\perp) \cap Before(c, d_\perp) = \emptyset$. Set

$$\begin{aligned} E_{d_\perp} &\stackrel{\text{def}}{=} \{e \in Enq(c) \mid e \notin (After(c, d_\perp) \cup Bad(c, d_\perp))\} \\ D' &\stackrel{\text{def}}{=} \{d' \in Deq(c) \mid \exists e \in E_{d_\perp} . Match(d') = e\} \\ D_{d_\perp} &\stackrel{\text{def}}{=} D' \cup \{d' \in Deq(c) \mid Match(d') = \perp \wedge \exists a \in E_{d_\perp} \cup D' . d' \prec_c a\} \end{aligned}$$

We have to show that E_{d_\perp} and D_{d_\perp} satisfy the three constraints.

- If $e \in E_{d_\perp}$, then it cannot be in $After(c, d_\perp)$ by construction. If $d \in D_{d_\perp}$, then either d belongs to D' or it is an event that precedes another event in $E_{d_\perp} \cup D'$. If $d \in D'$, then by construction its matching $e = Match(d)$ cannot be in $Bad(c, d_\perp)$. This implies that $d_\perp \not\prec_c d$, hence $d \notin After(c, d_\perp)$. If $d \notin D'$, then it is in D_{d_\perp} and there is some d' such that $d \prec_c d'$ and $d_\perp \not\prec_c d'$ which imply that $d_\perp \not\prec_c d$, hence $d \notin After(c, d_\perp)$.
- Let $a' \in E_{d_\perp} \cup D_{d_\perp}$ and $a \prec_c a'$. We do case analysis on a .
 - If $a = d \in Deq(c)$ with $Match(d) = \perp$, then by the construction of D_{d_\perp} , $d \in D_{d_\perp}$.
 - If $a = d \in Deq(c)$ with $Match(d) \neq \perp$, then if there is $e \in E_{d_\perp}$ such that $Match(d) = e$, then $d \in D_{d_\perp}$. Assume that $Match(d) = e \notin E_{d_\perp}$. This can happen when either $e \in After(c, d_\perp)$ or $e \in Bad(c, d_\perp)$. If e is in $After(c, d_\perp)$, which by the assumption that $Match$ is ordered implies that d must complete after e starts ($e \not\prec_c d$ must hold). This in turn implies that a' , beginning after d completes must be in $After(c, d_\perp)$, which contradicts the assumption that $a' \in E_{d_\perp}$. If e is in $Bad(c, d_\perp)$, then there must exist $e' \in Bad(c, d_\perp)$ such that either $e' \prec_c e$ or $e' \prec_c d$. Because $Match$ is ordered, we have $d \not\prec_c e$. Together with the assumption that $d \prec_c a'$, these imply $e' \prec_c a'$. Now, if $a' \in Enq(c)$, then $a' \in Bad(c, d_\perp)$ which contradicts the assumption that $a' \in E_{d_\perp}$. If $a' \in Deq(c)$ with $Match(a') \neq \perp$, that $e' \in Bad(c, d_\perp)$ and $e' \prec_c a'$ hold means that $Match(a') \in Bad(c, d_\perp)$ which in turn contradicts the assumption that $a' \in D_{d_\perp}$. Finally, if $a' \in Deq(c)$ with $Match(a') = \perp$, then because $a' \in D_{d_\perp}$ there is some $d'' \in D'$ such that $a' \prec_c d''$ which leads to the same contradiction as the previous case.
 - If $a = e \in Enq(c)$, a' is either an enqueue event e' or there is a dequeue event d' such that $e \prec_c d'$ and $Match(d') \neq \perp$. For the latter claim, observe that either $Match(a') \neq \perp$ and we take $d' = a'$ or $Match(a') = \perp$ and by definition of D_{d_\perp} there exists d' such that $a' \prec_c d'$ which by transitivity of \prec_c implies $e \prec_c d'$. If $e \notin E_{d_\perp}$, then either $e \in Bad(c, d_\perp)$ or $e \in After(c, d_\perp)$. If $e \in Bad(c, d_\perp)$ and $e \prec_c e'$ hold, then e' must

also be in $Bad(c, d_\perp)$ contradicting the assumption that $e' \in E_{d_\perp}$. If $e \in Bad(c, d_\perp)$ and $e \prec_c d'$ hold, then $Match(e')$, which exists because $Match$ is safe, must be in $Bad(c, d_\perp)$ which contradicts the assumption that $d' \in D_{d_\perp}$. If $e \in After(c, d_\perp)$, then $e \prec_c d'$ implies that $d' \in After(c, d_\perp)$ contradicting the assumption that $d' \in E_{d_\perp} \cup D_{d_\perp}$.

Thus, we conclude that $a \in E_{d_\perp} \cup D_{d_\perp}$ whenever $a \prec_c a'$ for some $a' \in E_{d_\perp} \cup D_{d_\perp}$.

- Let $e \in Before(c, d_\perp) \cap Enq(c)$. By the assumption that $Bad(c, d_\perp) \cap Before(c, d_\perp) = \emptyset$, $e \notin Bad(c, d_\perp)$. Thus, by construction $e \in E_{d_\perp}$, establishing $Before(c, d_\perp) \cap Enq(c) \subseteq E_{d_\perp}$. Since $e \notin Bad(c, d_\perp)$, there exists d such that $Match(d) = e$ and $d \in D'$, establishing $E_{d_\perp} \subseteq Match(D') \subseteq Match(D_{d_\perp})$.

(\Leftarrow) Assume that there exist $D_{d_\perp} \subseteq Deq(c)$ and $E_{d_\perp} \subseteq Enq(c)$ such that all three conditions are satisfied. We now show that the sets $Bad(c, d_\perp)$ and $Before(c, d_\perp)$ are disjoint. We show by induction that there is no index i such that $Bad_i(c, d_\perp) \cap Before(c, d_\perp) \neq \emptyset$. If $i = 0$, $e \in Bad_0(c, d_\perp)$ implies that there does not exist d such that $Match(d) = e$ and $d_\perp \not\prec_c d$. By the assumption that D_{d_\perp} and $After(c, d_\perp)$ are disjoint, we have $d \notin D_{d_\perp}$. But by the assumption that $Enq(c) \cap Before(c, d_\perp) \subseteq E_{d_\perp}$, we must have $e \in E_{d_\perp}$. This contradicts the assumption that $E_{d_\perp} \subseteq Match(D_{d_\perp})$.

Assume that for all indices less than or equal to k , for some $k > 0$, the claim holds: $i \leq k$ implies that $Bad_i(c, d_\perp)$ and $Before(c, d_\perp)$ are disjoint. Consider the index $k + 1$. Assume that there is $e \in Bad_{k+1}(c, d_\perp) \cap Before(c, d_\perp)$. Then there exists $e_k \in Bad_k(c, d_\perp)$ such that either $e_k \prec_c e$ or there is $d \in Deq(c)$ with $Match(d) = e$ and $e_k \prec_c d$. The former case, $e_k \prec_c e$, is not possible since that would imply that $e_k \in Before(c, d_\perp)$ and contradict that $Bad_k(c, d_\perp) \cap Before(c, d_\perp) = \emptyset$. By the assumption that $E_{d_\perp} \cup D_{d_\perp}$ is closed under \prec_c , $d \in D_{d_\perp}$ and $e_k \prec_c d$, we must have $e_k \in E_{d_\perp}$. By the assumption that $E_{d_\perp} \subseteq Match(D_{d_\perp})$ and $e_k \in E_{d_\perp}$, there must be $d_k \in D_{d_\perp}$ such that $Match(d_k) = e_k$. But if $e_k \in Bad_k(c, d_\perp)$ and $d_k \in D_{d_\perp}$, then there must be $e_{k-1} \in Bad_{k-1}(c, d_\perp)$ such that $e_{k-1} \prec_c e_k$ or $e_{k-1} \prec_c d_k$. Applying the same arguments as above, we arrive, after k iterations, to the conclusion that there must be some $e_0 \in Bad_0(c, d_\perp)$ which is also in E_{d_\perp} . But by definition, d_0 with $Match(d_0) = e_0$ cannot be in D_{d_\perp} (if d_0 exists, then $d_0 \in After(c, d_\perp)$). This contradicts the assumption that $E_{d_\perp} \subseteq Match(D_{d_\perp})$. \square

Definition 4.5. Let c be a complete history with a linearization witness $Match$. Call two events a and a' in c *overlapping* if neither $a \prec_c a'$ nor $a' \prec_c a$ holds. We define a relation $\ll_{c, Match}$ over $Enq(c)$. For two enqueue events e_1 and e_2 , we have $e_1 \ll_{c, Match}^1 e_2$ if $e_1 \neq e_2$ and one of the following holds:

- (1) $e_1 \prec_c e_2$.
- (2) e_1 and e_2 are overlapping, there exists d_1 such that $Match(d_1) = e_1$, but there does not exist d_2 such that $Match(d_2) = e_2$.
- (3) e_1 and e_2 are overlapping, and there exist d_1 and d_2 such that $Match(d_1) = e_1$, $Match(d_2) = e_2$, and $d_1 \prec_c d_2$.
- (4) e_1 and e_2 are overlapping, there exist d_1, d_2 such that $Match(d_1) = e_1$, $Match(d_2) = e_2$, and there exists $d \in Deq(c)$ such that $Val_c(d) = \text{NULL}$, $e_1 \notin Bad(c, d)$ and $e_2 \in Bad(c, d)$.

Let $\ll_{c, Match}$, called the *enq-order*, denote the transitive closure of $\ll_{c, Match}^1$. We will drop the subscripts when the history c and its linearization witness either are clear from the context or do not matter.

Lemma 4.6. *Let c be a complete history with linearization witness $Match$. Then, the induced enq-order $\ll_{c, Match}$ is a partial order over $Enq(c)$.*

Proof. We have to show that there does not exist a sequence e_1, \dots, e_{k+1} of enqueue events such that $e_i \ll^1 e_{i+1}$ for $i \in [1, k]$ and $e_{k+1} = e_1$. The proof is done by induction on k , the number of enqueue events in the sequence. In the base case, we note that $e_1 \ll^1 e_1$ is impossible by definition. Assume that there is no such sequence of length k or less. Consider the sequence e_1, \dots, e_{k+1} . For convenience, we will use d_i to denote the dequeue event in c such that $Match(d_i) = e_i$. If no such dequeue event exists for e_i , we will say that d_i *does not exist*. We make the following observations about this sequence:

- (1) If d_i does not exist, then d_{i+1} cannot exist. Assume the contrary and that for some i , we have $e_i \ll^1 e_{i+1}$, d_i does not exist and d_{i+1} exists. By the definition of \ll^1 , $e_i \ll^1 e_{i+1}$ cannot be due to conditions 2-4, because they all require the existence of d_i . Then, we must have $e_i \prec_c e_{i+1}$. On the other hand, since $Match$ is a linearization witness for c , by condition 2 of ordered mapping, the existence of d_{i+1} implies the existence of d_i , which contradicts the assumption that d_i does not exist. Because the sequence represents a cycle and \prec_c is a partial order, all d_i exist.
- (2) There cannot be two distinct pairs of events (e_i, e_{i+1}) and (e_j, e_{j+1}) such that $e_i \prec_c e_{i+1}$ and $e_j \prec_c e_{j+1}$ for some $i < j$. If there were, then we would have $e_i \prec_c e_{j+1}$ or $e_j \prec_c e_{i+1}$. If $e_i \prec_c e_{j+1}$, then $e_1 \ll^1 e_2 \dots \ll^1 e_i \ll^1 e_{j+1} \ll^1 \dots \ll^1 e_{k+1}$ hold and this sequence does not contain e_{i+1} . If $e_j \prec_c e_{i+1}$, then $e_{i+1} \ll^1 \dots \ll^1 e_j \ll^1 e_{i+1}$ hold and this sequence does not contain e_i . Thus, both sequences have less than $k + 1$ events, which contradict the inductive hypothesis.

We first show that none of the orderings in the cycle can be due to condition (4); i.e. there is no i such that $e_i \ll^1 e_{i+1}$ because there is some $d \in Deq(c)$ such that $e_i \notin Bad(c, d)$ and $e_{i+1} \in Bad(c, d)$. We assume the contrary and, without loss of generality, assume that $e_1 \ll^1 e_2$ is due to condition (4). Then, there is $d \in Deq(c)$ such that $e_1 \notin Bad(c, d)$ and $e_2 \in Bad(c, d)$. Observe that for all other enqueue events e_j in the sequence, $e_j \notin Bad(c, d)$ as otherwise, $e_1 \ll^1 e_j$ which results in a shorter cycle contradicting the inductive hypothesis. In particular, $e_3 \notin Bad(c, d)$, but this immediately leads to $e_3 \ll^1 e_2$. This implies that e_2, e_3, e_2 is also a cycle. Thus, if any consecutive events in the cycle are ordered due to condition (4), then $k \leq 2$. Clearly $e \ll^1 e$ can never hold due to condition (4), leading to the conclusion that if $e_1 \ll^1 e_2$ is due to condition (4), then $k = 2$.

Now, assume by contradiction that $e_1 \ll^1 e_2 \ll^1 e_1$ exists and there is d such that $e_1 \notin Bad(c, d)$ and $e_2 \in Bad(c, d)$. Since $e_1 \notin Bad(c, d)$, d_1 exists. By the first observation above, d_2 also exists. So, $e_2 \ll^1 e_1$ cannot be due to condition (2). We do a case analysis on the possible justifications for $e_2 \ll^1 e_1$.

- Assume that $e_2 \prec_c e_1$ (condition (1)). By the assumption that $e_2 \in Bad(c, d)$, we have $e_1 \in Bad(c, d)$, which contradicts the assumption that $e_1 \notin Bad(c, d)$.
- Assume that e_2 and e_1 are overlapping and $d_2 \prec_c d_1$ (condition (3)). Because $e_2 \in Bad(c, d)$, either $d \prec_c d_2$ or $d \prec_c e_2$ or there is an enqueue event $e' \in Bad(c, d)$ such that either $e' \prec_c e_2$ or $e' \prec_c d_2$ holds. If $d \prec_c d_2$ holds, then by transitivity $d \prec_c d_1$ also holds. If $d \prec_c e_2$ holds, then because $d_2 \prec_c e_2$ cannot hold ($Match$ is ordered), $d \prec_c d_1$ must hold. If $e' \prec_c e_2$ holds, then because $d_2 \not\prec_c e_2$ holds (due to $Match$ being ordered) we must have $e' \prec_c d_1$. Finally, if $e' \prec_c d_2$ holds, then by transitivity $e' \prec_c d_1$ also holds. All four cases contradict the assumption that $e_1 \notin Bad(c, d)$.
- Assume that there exists $d' \in Deq(c)$ such that $Match(d') = \perp$, $e_2 \notin Bad(c, d')$ and $e_1 \in Bad(c, d')$ (condition (4)). We do a case analysis on the possible justifications of $e_1 \in Bad(c, d')$ and $e_2 \in Bad(c, d)$ holding:

- $d' \prec_c e_1$, and $d \prec_c e_2$. Then either $d' \prec_c e_2$ or $d \prec_c e_1$ holds.
 - $d' \prec_c e_1$, and $d \prec_c d_2$. Then either $d' \prec_c d_2$ or $d \prec_c e_1$ holds.
 - $d' \prec_c e_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c e_2$. Then either $d' \prec_c e_2$ or $e_{b,d} \prec_c e_1$ holds.
 - $d' \prec_c e_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c d_2$. Then either $d' \prec_c d_2$ or $e_{b,d} \prec_c e_1$ holds.
 - $d' \prec_c d_1$, and $d \prec_c e_2$. Then either $d' \prec_c e_2$ or $d \prec_c d_1$ holds.
 - $d' \prec_c d_1$, and $d \prec_c d_2$. Then either $d' \prec_c d_2$ or $d \prec_c d_1$ holds.
 - $d' \prec_c d_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c e_2$. Then either $d' \prec_c e_2$ or $e_{b,d} \prec_c d_1$ holds.
 - $d' \prec_c d_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c d_2$. Then either $d' \prec_c d_2$ or $e_{b,d} \prec_c d_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c e_1$, and $d \prec_c e_2$. Then either $e_{b,d'} \prec_c e_2$ or $d \prec_c e_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c e_1$, and $d \prec_c d_2$. Then either $e_{b,d'} \prec_c d_2$ or $d \prec_c e_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c e_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c e_2$. Then either $e_{b,d'} \prec_c e_2$ or $e_{b,d} \prec_c e_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c e_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c d_2$. Then either $e_{b,d'} \prec_c d_2$ or $e_{b,d} \prec_c e_1$.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c d_1$, and $d \prec_c e_2$. Then either $e_{b,d'} \prec_c e_2$ or $d \prec_c d_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c d_1$, and $d \prec_c d_2$. Then either $e_{b,d'} \prec_c d_2$ or $d \prec_c d_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c d_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c e_2$. Then either $e_{b,d'} \prec_c e_2$ or $e_{b,d} \prec_c d_1$ holds.
 - There is $e_{b,d'} \in \text{Bad}(c, d')$ such that $e_{b,d'} \prec_c d_1$, and there is $e_{b,d} \in \text{Bad}(c, d)$ such that $e_{b,d} \prec_c d_2$. Then either $e_{b,d'} \prec_c d_2$ or $e_{b,d} \prec_c d_1$.
- In all cases the former implication contradicts $e_2 \notin \text{Bad}(c, d')$ and the latter implication contradicts $e_1 \notin \text{Bad}(c, d)$.

Thus, if e_1, e_2, \dots, e_{k+1} is a cycle in \ll^1 , none of the pairwise orderings can be due to condition (4).

Now consider the case where all consecutive events are overlapping; that is, e_i and e_{i+1} are overlapping for all $i \in [1, k]$. Then, by the definition of \ll^1 and the first observation, we must have $d_i \prec_c d_{i+1}$. But this would imply by the transitivity of \prec_c that $d_1 \prec_c d_{k+1} = d_1$ which is impossible due to \prec_c being a partial order.

So, there must be exactly one pair e_i and e_{i+1} of events ordered by \prec_c . Without loss of generality assume that $e_1 \prec_c e_2$. By the second observation, e_2 is overlapping with all e_{i+1} for $i \in [2, k]$. In particular, e_2 and $e_{k+1} = e_1$ must be overlapping. That contradicts the assumption that $e_1 \prec_c e_2$. Thus no sequence of length $k + 1$ can have a cycle in the \ll^1 relation. \square

The main result of this section is stated below.

Theorem 4.7. *A set of histories C is linearizable with respect to queue iff every $c \in C$ has a completion $\hat{c} \in \text{Compl}(c)$ that has a linearization witness.*

Proof.

(\Rightarrow) 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 Theorem 3.10, 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 . In particular, μ_{seq} is safe because conditions (i) to (iii) of sequential witness imply conditions (1) to (3) of safe mapping. It is ordered because

- By condition (iv) of sequential witness, $\mu_{\text{seq}}(d) = e$ implies $e \prec_s d$ and definition of linearizability implies that $d \not\prec_c e$, which is condition (1) of ordered mapping,
- Assume that there exist d', e', e such that $e' = \mu_{\text{seq}}(d')$ and $e \prec_c e'$. Then by definition of linearization, $e \prec_s e'$. By condition (v) of sequential witness, $d = \mu_{\text{seq}}^{-1}(e)$ exists and $d \prec_s d'$. By definition of linearization, this in turn implies that $d' \not\prec_c d$, which is condition of (2) of ordered mapping.

Assume $d = \text{deq}(\text{NULL}) \in \text{Deq}(c)$. Define the sets $D_d = \{d' \in \text{Deq}(s) \mid d' \prec_s d\}$, $E_d = \{e \in \text{Enq}(s) \mid e \prec_s d\}$. Observe that $(D_d \cup E_d) \cap \text{After}(d, c) = \emptyset$ because for any $a \in \text{After}(d, c)$, by definition we have $d \prec_c a$, which implies $d \prec_s a$, which in turn implies $a \notin D_d \cup E_d$. Assume there is $e \in \text{Before}(d, c) \cap \text{Enq}(c)$. Then by definition of linearization, $e \prec_s d$. By construction, $\text{Before}(d, c) \cap \text{Enq}(c) \subseteq E_d$. Let i denote the position of d in s ; i.e. $s\langle i \rangle = d$. Because s is legal, it has an obs-equivalent canonical behavior, s' . By Lemma 3.8 $s'\langle i \rangle = d$. By definition of canonical behavior, each enqueue event in $s'\langle 1 : i - 1 \rangle$ has a matching dequeue event in $s'\langle 1 : i - 1 \rangle$. Since s and s' are obs-equivalent, then each enqueue event in $s\langle 1 : i - 1 \rangle$ has a matching dequeue event in $s\langle 1 : i - 1 \rangle$. Thus, $E_d \subseteq \text{Match}(D_d)$, the inclusion being proper in case D_d contains a NULL-dequeue event (distinct from d since $d \notin D_d$). Thus, $\text{Before}(d, c) \cap \text{Enq}(c) \subseteq E_d \subseteq \text{Match}(D_d)$. Since all conditions of linearization witness per Lem. 4.4 are satisfied for D_d and E_d , μ_{seq} is a linearization witness.

(\Leftarrow) Let c be a complete history with a linearization witness Match . Let $<$ denote a total order extension of \ll . That is, $<$ is a total order over $\text{Enq}(c)$ such that whenever $e \ll e'$, we have $e < e'$. Let e^* denote the $<$ -maximal enqueue event over $<$. That is, for any $e \in \text{Enq}(c)$, we have $e < e^*$ whenever $e \neq e^*$.

In order to prove the if-direction (\Leftarrow), we will make use of $<$ to construct a sequence s with sequential witness μ . We actually prove a stronger property, which also requires that if $e < e'$ in c then $e \prec_s e'$. By Theorem 3.10, the result follows.

The construction is given by induction on the number of (completed) events in c . In the base case, there are no events and ε with empty mapping is the desired sequence. Assume that the claim holds for all complete concurrent histories with k events or less. Let c be a complete concurrent history with $k + 1$ events and Match be a linearization witness for c . We first choose an event.

Call event $a \in \text{Enq}(c) \cup \text{Deq}(c)$ *maximal* (relative to $<$), if there is no event a' such that $a \prec_c a'$ and one of the following holds:

- (1) $a = e^* \in \text{Enq}(c)$, there is no d^* such that $\text{Match}(d^*) = e^*$.
- (2) $a = d \in \text{Deq}(c)$ with $\text{Match}(d) \neq \perp$, there is no d' such that $\text{Match}(d) < \text{Match}(d')$.
- (3) $a = d_\perp \in \text{Deq}(c)$ with $\text{Match}(d_\perp) = \perp$, and $\text{Bad}(c, d_\perp) = \emptyset$.

Let c be a non-empty complete history and Match be its linearization witness. We first show that there is at least one event in c that is maximal relative to $<$. First, observe that if $\text{Enq}(c) = \emptyset$, then any $d \in \text{Deq}(c)$ must return NULL; otherwise, Match cannot be safe. Then, any d such that no $d' \in \text{Deq}(c)$ with $d \prec_c d'$ exists is maximal. Since \prec_c is a

partial-order, such d must exist. If conversely we assume that $Enq(c) \neq \emptyset$ and $Deq(c)$ is empty, then e^* is maximal.

Assume that $Enq(c)$ and $Deq(c)$ are non-empty. If e^* is not maximal, it must be because there is $d^* \in Deq(c)$ such that $Match(d^*) = e^*$. Then, by definition of $<$ and the assumption that e^* is $<$ -maximal, there cannot be $d' \in Deq(c)$ such that $d^* \prec_c d'$ if $Match(d') \neq \perp$. So, d^* is not maximal only if there is $d_\perp \in Deq(c)$, $Match(d_\perp) = \perp$ and $d^* \prec_c d_\perp$. Furthermore, the definition of \ll^1 , that e^* is $<$ -maximal and d^* exists imply that for all $e' \in Enq(c)$, there is $d' \in Deq(c)$ such that $Match(d') = e'$. In particular, this means that for $d_\perp \in Deq(c)$ such that no $d' \in Deq(c)$ with $d_\perp \prec_c d'$ exists and $d^* \prec_c d_\perp$ with $Bad(c, d_\perp) = \emptyset$, setting d_\perp as a maximal element. Thus, the set of maximal events in any non-empty history is non-empty.

Let A denote the set of maximal elements relative to $<$. If A contains a dequeue event d such that $Match(d) = \perp$, then we choose d . Otherwise, if A contains a dequeue event d^* such that $Match(d^*) \neq \perp$, then we choose d^* . If neither condition holds, we choose e^* .

We now show that if c is a non-empty history with linearization witness $Match$, the history c' obtained by removing the chosen event from c has $Match'$, which is $Match$ restricted to the remaining events in c' , as a linearization witness. Before we do a case analysis on the type of the chosen event, we make two observations. If c' is obtained from c by removing an event a and a mapping is safe for c , then it is also safe for c' when restricted to the $Deq(c')$. Second, removing a from c does not change the relative ordering among the remaining events. So $b \prec_c d$ holds iff $b \prec_{c'} d$ holds. In particular, if $a \in Deq(c)$ and a mapping is ordered for c , then it is ordered for c' .

We have three cases to consider for the chosen event:

- The chosen event is d_\perp with $Match(d_\perp) = \perp$. Let $d' \in Deq(c)$ be such that $Match(d') = \perp$. Since $Match(d_\perp) = \perp$, after removing d_\perp we have $Enq(c') = Enq(c)$ and thus $Bad(c, d') = Bad(c', d')$. Additionally, $Before(c, d')$ is the same as $Before(c', d')$ when both are restricted to $Enq(c) = Enq(c')$. Then, we have

$$\begin{aligned}
& Before(c, d') \cap Bad(c, d') \\
&= Before(c, d') \cap Bad(c, d') \cap Enq(c) && [Bad(c, d') \subseteq Enq(c)] \\
&= Before(c, d') \cap Bad(c, d') \cap Enq(c') && [Enq(c) = Enq(c')] \\
&= Before(c, d') \cap Bad(c', d') \cap Enq(c') && [d_\perp \notin Bad(c, d')] \\
&= Before(c', d') \cap Bad(c', d') \cap Enq(c') && [Before(c, d') \cap Enq(c) = Before(c', d') \cap Enq(c')] \\
&= Before(c', d') \cap Bad(c', d') && [Bad(c', d') \subseteq Enq(c')]
\end{aligned}$$

establishing that $Before(c', d') \cap Bad(c', d') = \emptyset$. Thus, $Match'$ is a linearization witness for c' .

- The chosen event is $d^* \in Deq(c)$. Observe that $Match(d^*)$ is the $<$ -maximal enqueue event e^* relative to $<$. By the second observation above, $Match'$ is ordered for c' . We have to show that for any $d_\perp \in Deq(c)$, $Match'(d_\perp) = \perp$ is justified; that is, $Bad(c', d_\perp) \cap Before(c', d_\perp) = \emptyset$. By the assumption that $Match$ is a linearization witness for c , we have $Bad(c, d_\perp) \cap Before(c, d_\perp) = \emptyset$. If $d_\perp \prec_c e^*$, then $e^* \in Bad(c', d_\perp)$ by definition. If $d_\perp \prec_c d^*$, then $e^* \in Bad_0(c, d_\perp)$ and $e^* \in Bad_0(c', d_\perp)$, so $Bad(c', d_\perp) = Bad(c, d_\perp)$.

Then, the interesting case is when $e^* \notin Bad(c, d)$. First observe that $Bad(c, d) \neq \emptyset$ iff $e^* \in Bad(c, d)$. For the only-if (\Rightarrow) direction, assume that there is some $e' \in Bad(c, d)$. By the definition of \ll^1 , if $e^* \notin Bad(c, d)$ then $e^* \ll^1 e'$ contradicting the $<$ -maximality of e^* . The if (\Leftarrow) direction is trivial. This implies that $Bad(c, d) = \emptyset$ because $e^* \notin Bad(c, d_\perp)$.

If there are several such NULL-returning dequeues, choose d_{\perp} such that for any $a \in \text{Deq}(c)$ with $d_{\perp} \prec_c a$ implies $\text{Match}(d) \neq \perp$. Intuitively, d_{\perp} is the \prec -maximal among dequeue events returning NULL.

Now since d^* was chosen, we know that there must be at least one a such that $d_{\perp} \prec_c a$, since otherwise d_{\perp} would have been chosen. By the assumption about d_{\perp} , $a \notin \text{Deq}(c)$ with $\text{Match}(a) = \perp$. If $a = e \in \text{Enq}(c)$, then $e \in \text{Bad}(c, d_{\perp})$ contradicting the assumption that $\text{Bad}(c, d_{\perp}) = \emptyset$. So $a = d \in \text{Deq}(c)$ with $\text{Match}(d) \neq \perp$. But then $\text{Match}(a)$, which must exist because Match is safe, is in $\text{Bad}(c, d_{\perp})$, again contradicting the assumption that $\text{Bad}(c, d_{\perp}) = \emptyset$. So, by contradiction we conclude that there is no such d_{\perp} for which $\text{Bad}(c, d_{\perp}) = \emptyset$ and $\text{Bad}(c', d_{\perp}) \neq \emptyset$ hold.

- The chosen event is $e^* \in \text{Enq}(c)$. By the assumption about the chosen event, d^* does not exist, so $\text{Deq}(c) = \text{Deq}(c')$ and Match' is safe because Match is safe. Because d^* does not exist, if d_{\perp} is such that $\text{Match}(d_{\perp}) = \perp$, then $e^* \in \text{Bad}(c, d_{\perp})$. Then, for every such d_{\perp} , $\text{Bad}(c', d_{\perp}) \subseteq \text{Bad}(c, d_{\perp})$, which means that $\text{Bad}(c, d_{\perp}) \cap \text{Before}(c, d_{\perp}) = \emptyset$ implies $\text{Bad}(c, d_{\perp}) \cap \text{Before}(c', d_{\perp}) = \emptyset$. So, Match' is a linearization witness for c' .

Now, we know that Match' is a linearization witness for c' which has exactly k events. By the inductive hypothesis, c' is linearizable with respect to queue. That is, there is a linearization s' of c' which is a legal queue behavior. By Theorem 3.10, s' has a sequential witness μ' . We claim that $s = s' \cdot a$, where a is the chosen element in c as described above, is a legal queue behavior. Additionally, we will also show that for any two enqueue events e and e' both in $\text{Enq}(c)$, $e < e'$ implies $e \prec_s e'$.

Assume that the chosen element was $a = d_{\perp} \in \text{Deq}(c)$ such that $\text{Match}(d_{\perp}) = \perp$. We set $\mu = \mu'[a \mapsto \perp]$. Observe that by the assumption that d_{\perp} is a chosen element, we must have $\text{Bad}(c, d_{\perp}) = \emptyset$. This implies that for all $e \in \text{Enq}(c)$, there is $d \in \text{Deq}(c)$ such that $\text{Match}(d) = e$; as otherwise, e would be in $\text{Bad}(c, d_{\perp})$. Since all events of c' are the same as the events of s' , the sets $\{e \in \text{Enq}(c) \mid e \prec_s d_{\perp}\} = \text{Enq}(c)$ and $\{d \in \text{Deq}(c) \mid d \prec_s d_{\perp} \wedge \mu(d) \neq \perp\}$ have the same cardinality. These along with the inductive hypothesis that μ' is a sequential witness for s' imply that all six conditions of a sequential witness are satisfied for μ and s . Because the relative ordering of events in $\text{Enq}(c)$ in s' remains the same in s , $e \prec_{s'} e'$ implies $e \prec_s e'$, and by induction hypothesis this can happen only when $e < e'$.

Assume that the chosen element was $a = d^* \in \text{Deq}(c)$ such that $\text{Match}(d^*) = e^*$. We set $\mu = \mu'[a \mapsto e^*]$. Because Match was safe for c , e^* exists and μ is well-defined. By the inductive hypothesis, e^* is in s' and hence $e^* \prec_s d^*$. Again by the inductive hypothesis, for any $e \in \text{Enq}(c)$, we have $e \prec_s e^*$. Since d^* is the last event in s , no event can follow d^* in s . In particular, there is no $d' \in \text{Deq}(c)$ such that $d^* \prec_s d'$. These along with the inductive hypothesis imply that μ is a sequential witness for s . Similar to the previous case, $e \prec_{s'} e'$ implies $e \prec_s e'$ and by inductive hypothesis this can happen only when $e < e'$.

Assume that the chosen element was $a = e^*$. We take $\mu = \mu'$. Because e^* is chosen, d^* does not exist in c . Furthermore, since e^* is the last event in s , no other event can follow e^* in s . These observations along with the inductive hypothesis imply that μ is a sequential witness for s . Observe also that e^* , being the last element in s , also satisfies the condition that it should not precede any other enqueue event in s , satisfying the condition that $e < e'$ implies $e \prec_s e'$. \square

Necessary and Sufficient Conditions for Complete Histories. We now focus on complete histories, namely ones with no pending events. We observe that whether a history is not linearizable can always be determined by examining the dequeued values. Let c be a complete history. In order to simplify the technical presentation we assume that each value is enqueued at most once.² The possible violations in c are:

- (VFresh): A dequeue event returns a value not previously inserted by any enqueue event. Formally, there exists a value $x \neq \text{NULL}$ such that $\text{deq}(x) \in \text{Deq}(c)$ and either $\text{enq}(x) \notin \text{Enq}(c)$ or $\text{deq}(x) \prec_c \text{enq}(x)$.
- (VRepet): Two dequeue events return the value inserted by the same enqueue event. Formally, there exist two dequeue events $d, d' \in \text{Deq}(c)$ such that $\text{Val}_c(d) = \text{Val}_c(d') \neq \text{NULL}$.
- (VOrd): Two values are enqueued in a certain order, and a dequeue returns the later value before any dequeue of the earlier value starts. Formally, there exist values x, y such that $\text{enq}(y) \prec_c \text{enq}(x)$, $\text{deq}(x) \in \text{Deq}(c)$, and either $\text{deq}(y) \notin \text{Deq}(c)$ or $\text{deq}(x) \prec_c \text{deq}(y)$.
- (VWit): A dequeue event returning NULL even though the queue is never logically empty during the execution of the dequeue event. Formally, let $c = c_0 \cdot \text{deq}_i(\text{NULL}) \cdot c_d \cdot \text{deq}_r(\text{NULL}) \cdot c_3$, where c_0, c_d, c_3 represent subsequences of c . Then for any choice of c_1 and c_2 such that $c_d = c_1 \cdot c_2$, there exists an $\text{enq}(x) \in \text{Enq}(c)$ completed in $c_0 \cdot \text{deq}_i(\text{NULL}) \cdot c_1$ and $\text{deq}_i(x)$ does not occur in $c_0 \cdot \text{deq}_i(\text{NULL}) \cdot c_1$.

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

Proposition 4.8. *A complete history c is linearizable with respect to queue iff it has none of the VFresh, VRepet, VOrd, VWit violations.*

Proof.

(\Rightarrow) If c is linearizable with respect to queue, then by Theorem 4.7, $\hat{c} = c$ has a linearization witness Match . We show by contradiction that none of the four violations can happen in c .

- Assume that c has VFresh. Then there exists a dequeue event $d \in \text{Deq}(c)$ such that $\text{Val}_c(d) \neq \text{NULL}$ and either $e = \text{Match}(d)$ does not exist or $d \prec_c e = \text{Match}(d)$. That $e = \text{Match}(d)$ does not exist is impossible because by the second condition of safe mapping, $\text{Match}(d) \neq \perp$ and by the first condition of safe mapping $\text{Match}(d) \in \text{Enq}(c)$. That $d \prec_c e = \text{Match}$ holds is impossible because by the first condition of safe mapping, $d \not\prec_c \text{Match}(d) = e$.
- Assume that c has VRepet. Then there exist $d, d' \in \text{Deq}(c)$ with $\text{Val}_c(d) = \text{Val}_c(d') \neq \perp$. This is impossible by the third condition of safe mapping.
- Assume that c has VOrd. Then there exist $e, e' \in \text{Enq}(c)$, $d' \in \text{Deq}(c)$ such that $e \prec_c e' = \text{Match}(d')$ and either $d \in \text{Deq}(c)$ such that $\text{Match}(d) = e$ does not exist or such a d exists and $d' \prec_c d$. Both possibilities contradict the second condition of ordered mapping.
- Assume that c has VWit. Then c is of the form $c_0 \cdot \text{inv}(d_\perp) \cdot c_d \cdot \text{res}(d_\perp) \cdot c_3$ such that $\text{Match}(d_\perp) = \perp$, and for every possible partitioning of $c_d = c_1 \cdot c_2$, there is an enqueue event $e \in \text{Enq}(c_p)$ with $c_p = c_0 \cdot \text{inv}(d_\perp) \cdot c_1$ such that e is completed in c_p and there is no dequeue event d , pending or completed, in c_p such that $\text{Match}(d) = e$. First, observe that by choosing $c_2 = \varepsilon$ (resulting in $c_p = c_0 \cdot \text{inv}(d_\perp) \cdot c_d$), we conclude that there is at least one enqueue event $e_0 \in \text{Enq}(c_p)$ whose matching dequeue event d_0 is not in $\text{Enq}(c_p)$; that is, $d_\perp \prec_c d_0$ if $d_0 \in \text{Deq}(c)$. This implies that $e_0 \in \text{Bad}(c, d_\perp)$. Because Match is a linearization witness for c , we must have $\text{Bad}(c, d_\perp) \cap \text{Before}(c, d_\perp) = \emptyset$. In other words,

²In case there are multiple occurring values, this is akin to guessing the mapping Match ; it is enough that at least one guess satisfies the criteria (absence of violations).

all enqueue events $e \in \text{Bad}(c, d_\perp)$ must not belong to $\text{Before}(c, d_\perp)$. This implies that if $e \in \text{Bad}(c, d_\perp)$ then $\text{res}(e)$ must happen after $\text{inv}(d_\perp)$. Let $e \in \text{Bad}(c, d_\perp)$ be chosen such that for any other $e' \in \text{Bad}(c, d_\perp)$, $\text{res}(e)$ occurs before $\text{res}(e')$ in c . Let $c_d = c_1 \cdot c_2$ with $c_2 = \text{res}(e) \cdot c'_2$. By the assumption that there is a VWit violation for d_\perp , there must be an enqueue event e' in $c_p = c_0 \cdot \text{inv}(d_\perp) \cdot c_1$ such that if there is $d' \in \text{Deq}(c)$ with $\text{Match}(d') = e'$, then d' is neither completed nor pending in c_p . This implies that $\text{inv}(d')$ if it exists must occur after $\text{res}(e)$. Because e is not completed in c_p (it is completed in $c_p \cdot \text{res}(e)$), $e' \neq e$. These two facts imply that either $d' \notin \text{Deq}(c)$ or if $d' \in \text{Deq}(c)$ then $e \prec_c d'$ holds. But this implies that $e' \in \text{Bad}(c, d_\perp)$. This contradicts the assumption that $\text{res}(e)$ is the first enqueue event in $\text{Bad}(c, d_\perp)$ to complete in c . Such an e does not exist implies that there is at least one enqueue event e_b in $\text{Bad}(c, d_\perp)$ which is completed in c_0 , which implies that $e_b \in \text{Before}(c, d_\perp)$. Finally, this contradicts the assumption that $\text{Bad}(c, d_\perp)$ and $\text{Before}(c, d_\perp)$ are disjoint.

(\Leftarrow) Assume that there exists a complete history c in which none of the violations happen. We will show that the mapping that pairs events enqueueing and dequeuing the same value is a linearization witness for c .

Let $D_v = \{\text{deq}(x) \in \text{Deq}(c) \mid x \neq \text{NULL}\}$ denote the set of all non-NULL returning dequeue events of c . Similarly, let $D_n = \text{Deq}(c) \setminus D_v$ denote the set of all NULL returning dequeue events of c . Let M_v be the mapping from D_v to $\text{Enq}(c)$ such that $M_v(d) = e$ iff $\text{Val}_c(d) = \text{Val}_c(e)$. Let M_n be such that all $d \in D_n$ are mapped to \perp . We claim that Match defined as

$$\text{Match}(d) \stackrel{\text{def}}{=} \begin{cases} M_v(d) & \text{if } d \in D_v \\ \perp & \text{if } d \in D_n \end{cases}$$

is a linearization witness for c .

First, observe that M_v is a total mapping because c does not have VFresh. Furthermore, because c does not contain VRepet, Match is a safe mapping by construction. Match satisfies the first condition of an ordered mapping because c does not have VFresh. Match satisfies the second condition of an ordered mapping because c does not have VOrd. Thus, Match is also an ordered mapping.

Let $d_\perp \in D_n$ be a NULL-returning dequeue event in c . We have to show that $\text{Before}(c, d_\perp)$ and $\text{Bad}(c, d_\perp)$ are disjoint. Because c has no VWit violation, there must be a prefix $c_p = c_0 \cdot \text{inv}(d_\perp) \cdot c_1$ of c such that if e is an enqueue event is completed in c_p then its matching dequeue event d (i.e. $\text{Match}(d) = e$) is either pending or completed in c_p . In other words, if $\text{res}(e)$ occurs in c_p , then so does $\text{inv}(d)$. Let $e_j \in \text{Bad}(c, d_\perp)$ be such that $e_j \in \text{Bad}_j(c, d_\perp)$, for any $e_k \in \text{Bad}(c, d_\perp)$ we have $j \leq k$, and e_j is completed in c_p . If there is no such e_j , that is, if $\text{Bad}(c, d_\perp)$ is empty, then we are done. Otherwise, observe that $j \neq 0$ because by the absence of VWit, there is d_j such that $\text{Match}(d_j) = e_j$ and $d_\perp \not\prec d_j$; in particular, $\text{inv}(d_j)$ occurs in c_p . But $j > 0$ implies that there is $e_{j-1} \in \text{Bad}_{j-1}(c, d_\perp)$ such that either $e_{j-1} \prec_c e_j$ or $e_{j-1} \prec_c d_j$. Both cases imply that e_{j-1} must be completed in c_p , contradicting the assumption that j was minimal. Thus, there are no enqueue events in $\text{Bad}(c, d_\perp)$ which are completed in c_p . Since $\text{Before}(c, d_\perp)$ is contained in the set of completed events of c_p , we conclude that $\text{Before}(c, d_\perp)$ and $\text{Bad}(c, d_\perp)$ are disjoint.

This concludes the proof that Match is a linearization witness for c . \square

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 4.9. *Given an infinite sequence of complete histories c_1, c_2, \dots 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 $\text{enq}(x)$, then exists some c_j containing $\text{deq}(x)$.*

Proof. We prove this by contradiction. If there is no $\text{deq}(x)$ event, then $\text{enq}(x)$ is always in the queue, and so, from the absence of VWit violations, none of the dequeue events following $\text{enq}(x)$ 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 $\text{enq}(x)$ which completes in c_1 . This means that eventually one dequeue event has to return an element inserted by $\text{enq}(y)$ such that $\text{enq}(x) \prec_{c_j} \text{enq}(y)$, which is VOrd. \square

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 $\text{Val}_c(e_1) \neq \text{Val}_c(e_2)$, a dequeue event d_2 cannot return $\text{Val}_c(e_2)$ if $\text{Val}_c(e_1)$ is not removed in c or is removed by d_1 with $d_2 \prec_c d_1$.

Thus, to check this property, it suffices to come up with an overapproximation of all those executions satisfying the premise of POrd, and prove that such executions 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 .

Necessary and Sufficient Conditions for Purely-Blocking Queues. There is a subtle complication in the statement of Theorem 4.7. 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 the reachable states of the implementation. What we would ideally like to do is to claim that if the implementation violates linearizability, then there is a finite complete induced history of the implementation which has no witness. In other words, if the implementation contains an incomplete execution trace whose induced (incomplete) history has no witness, then that execution trace is the prefix of a complete execution trace of the implementation.

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

Theorem 4.10. *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. 4.8, it has no linearization, contradicting the assumption that the implementation is linearizable.

(\Leftarrow) Consider an arbitrary induced history c of the implementation. As the implementation is completable, there exists a completion $\hat{c} \in \text{Compl}(c)$ that is a valid induced history of the implementation. From our assumptions, \hat{c} cannot have a violation, and so by Prop. 4.8, \hat{c} has a linearization, and therefore so does c . \square

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. Formally, let $\tau = \tau_0 \cdot (t : \text{enter}(m)) \cdot \tau_1$ be an execution trace of the implementation in which m executed by t is pending, i.e. $(t : \text{exit}(m))$ does not occur in τ_1 . The pending method m is called *pure after* τ if for any sequence τ_e in which no action of m by t occurs and any sequence τ_m in which only actions of m by t occur, $\tau \cdot \tau_e$ is an execution trace of the implementation iff $\tau \cdot \tau_m \cdot \tau_e$ is an execution trace of the implementation. The execution trace τ is called *obstruction-free for* m if there is another execution trace $\tau' = \tau \cdot \tau_2 \cdot (t : \text{exit}(m))$ of the implementation such that all actions in τ_2 belong to m executed by t . Then, the implementation is purely-blocking if for each execution trace τ of the implementation and pending method m in τ , either τ is obstruction-free for m or m is pure after τ .

Proposition 4.11. *Every purely-blocking implementation is completable.*

Proof. Let τ be an execution trace of a purely-blocking implementation. We fix a total order of pending methods, and consider them in that order. For a pending method m executed by t , if running it in isolation terminates, then extend τ only with actions executed by t until $(t : \text{exit}(m))$ occurs. Otherwise, the execution of m does not modify any global state and so all actions executed by t beginning with the last occurrence of $(t : \text{enter}(m))$ can be removed from the execution trace without affecting its realizability. \square

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 [9] for an in depth exposition of these three notions.)

5. 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 $x \neq \text{NULL}$ at E_2 , in which case it immediately terminates. So from Prop. 4.11 and Theorem 4.10, 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 execution trace 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 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 `enq(x)` events in the induced 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 an execution trace τ where every method call enqueueing v_2 is preceded by some method call enqueueing some different value v_1 and there are no **deq**() calls returning v_1 (there may be arbitrarily many concurrent **enq**() and **deq**() calls enqueueing or dequeuing other values). The goal is to show that in this execution trace, 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 **enq**(v_2) incremented it at E_1 .

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

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

Note that since, by assumption, v_1 can never be dequeued, and any later **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 . Formally, to show this we use the following loop invariant at the beginning of **for** loop

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

and (*) for the while loop. With these invariants, it is immediate that the swap at line D_2 cannot read v_2 .

6. CHECKING THE CONDITIONS BY PROVING PROGRAM DIVERGENCE

In this section, we reduce proving the absence of **VFresh**, **VRepet** and **VOrd** violations to proving that certain programs always diverge. Towards the end of the section, we also discuss how the absence of **VWit** violations might be automatically checked for queue implementations whose **deq** method may return **NULL**.

Our proof technique relies heavily on instrumenting the **deq**() function with a prophecy variable ‘guessing’ the value that will be returned when calling it. That is, we construct a method, **deq**(v), such that the set of execution traces of $\bigsqcup_{x \in \mathbb{N} \cup \{\text{NULL}\}} \mathbf{deq}(x)$ is equal to the set of execution traces of **deq**(), where \bigsqcup stands for (demonic) non-deterministic choice: the set of traces of $\mathcal{T} \bigsqcup \mathcal{T}'$ is the union of the sets of traces of \mathcal{T} and \mathcal{T}' . A simple construction is to define **deq**(v) to behave exactly as **deq**() except that when **deq**() is about to return a value other than v , we make **deq**(v) diverge. That is, we prepend an **assume**($x = v$); statement to every **return** x statement in **deq**(). In Section 7, we describe a better construction.

Proving Absence of VFresh Violations. Generally, it is completely straightforward to prove the absence of **VFresh** violations. For example, it is sufficient for the queue implementation to be data independent [22].

This is because a data independent implementation cannot produce values ‘out of thin air.’ In other words, if a dequeue returns a value, it must have read that value from memory, and the only way for a value to get into memory is for an enqueue to be invoked with that value passed as an argument. Therefore, no **VFresh** violations can occur in data independent implementations.

Proving Absence of VRepet Violations. To prove the absence of VRepet violations, we use the following theorem.

Theorem 6.1. *A completable queue implementation has no VRepet violations iff for all values v and all $n, m, k \in \mathbb{N}$ such that $0 < n < m$, the program*

$$\text{Prg}(v, n, m, k) \stackrel{\text{def}}{=} \overbrace{(\text{enq}(v) \parallel \dots \parallel \text{enq}(v))}^{n \text{ times}} \parallel \overbrace{(\text{deq}(v) \parallel \dots \parallel \text{deq}(v))}^{m \text{ times}} \parallel \overbrace{C \parallel \dots \parallel C}^{k \text{ times}}$$

has no execution trace in which more than n $\text{deq}(v)$ threads terminate, where

$$C \stackrel{\text{def}}{=} \bigsqcup_{x \neq v} \text{enq}(x) \sqcup \bigsqcup_{x \neq v} \text{deq}(x).$$

Proof. (\Rightarrow) We argue by contradiction. Consider an execution trace τ of $\text{Prg}(v, n, m, k)$ where at least $n + 1$ of the $\text{deq}(v)$ threads terminate. The induced history $h(\tau)$ cannot have a safe matching because to satisfy condition (1) of Definition 4.1, each $\text{deq}(v)$ must be matched by some $\text{enq}(v)$, and from the pigeonhole principle multiple $\text{deq}(v)$ will have to be matched with the same $\text{enq}(v)$, thereby violating condition (3) of the Definition.

(\Leftarrow) Again, we argue by contradiction. Assume the queue implementation has an execution trace τ such that $h(\tau)$ has a VRepet violation. For each value v , let n_v be the number of invoked $\text{enq}(v)$ operations in τ and m_v be the number of invoked $\text{deq}(v)$ operations. Then, since there is a VRepet violation, for some v there are at least $n_v + 1$ completed $\text{deq}(v)$ operations in τ . Finally, observe that τ can be generated by a run of the program $\text{Prg}(v, n_v, m_v, k)$ (for some k) in which at least $n_v + 1$ of the $\text{deq}(v)$ threads terminate. \square

In case the queue implementation is data independent [22], we can simplify the VRepet check further. We say that a history is *differentiated*, if all the input arguments to invocations of the library's methods are pairwise different. Given a renaming function on data values, $f : \mathcal{D} \rightarrow \mathcal{D}$, we write $f(c)$ for applying the function to all the data values in the history c . An implementation is *data independent*, if the set of histories it generates, H , satisfies two properties: (1) for every $c \in H$, $f(c) \in H$; and (2) for every $c \in H$, there exists a differentiated history $c' \in H$ such that $c = f(c')$. To ensure data independence, it suffices to check that the implementation never performs any operations (such as testing for equality) on the value domain.

For data-independent programs, we can reduce reasoning about any number (say n and m where $m > n$) of $\text{enq}(v)$ and $\text{deq}(v)$ threads to a single $\text{enq}(v)$ and multiple $\text{deq}(v)$ threads. To see why a data independence condition is necessary, consider the following incorrect $\text{enq}(v)$ and $\text{deq}()$ implementations:

$$\begin{aligned} \text{enq}(v) &\stackrel{\text{def}}{=} \text{atomic (if } v \in Q \text{ then } Q := Q \cdot v \cdot v \text{ else } Q := Q \cdot v) \\ \text{deq}() &\stackrel{\text{def}}{=} \text{atomic (match } Q \text{ with } \epsilon \rightarrow \text{block} \mid v \cdot Q' \rightarrow Q := Q'; \text{return } v) \end{aligned}$$

Observe that for all $m > 1$, the program $\text{Prg}(v, 1, m, 0)$ never terminates whereas the program $\text{Prg}(v, 2, 3, 0)$ has a terminating execution: the serial execution where both enqueues take place before all the dequeues.

Theorem 6.2. *A data-independent completable queue implementation has no VRepet violations iff for all values v , all $m > 1$ and all $k \in \mathbb{N}$, the program $\text{Prg}(v, 1, m)$ (as defined in Theorem 6.1) has no execution in which more than one $\text{deq}(v)$ threads terminate.*

Proof. By Theorem 6.1, it suffices to show that if for all v , m and k , $\text{Prg}(v, 1, m, k)$ has no execution trace with more than one terminating $\text{deq}(v)$, then for all v , n , m and k , no execution trace of the program $\text{Prg}(v, n, m, k)$ can have more than n terminating $\text{deq}(v)$ threads. Now, as enq and deq do not perform any value-dependent operations, we can replace the v being enqueued by distinct fresh v_i values. Doing so will naturally affect the return values of the dequeue operations that were returning v , but because of data independence, nothing else. Hence, the program

$$\overbrace{\text{enq}(v_1) \parallel \dots \parallel \text{enq}(v_n)}^{n \text{ threads}} \parallel \overbrace{\text{deq}(r_1) \parallel \dots \parallel \text{deq}(r_m)}^{m \text{ threads}} \parallel \overbrace{C \parallel \dots \parallel C}^{k \text{ times}}$$

must have an execution trace where at least $n + 1$ of the $\text{deq}(r_i)$ threads terminate with $r_i \in \{v_1, \dots, v_n\}$ for $0 < i < m$. So, by the pigeonhole principle, there exists some value v_i that gets dequeued multiple times, say m' . This, however, contradicts our assumption that $\text{Prg}(v_i, 1, m', -)$ has at most one terminating $\text{deq}(v_i)$ thread. \square

Proving Absence of VOrd Violations. We move on to the POrd property, which as we have seen in the manual proof of the HW queue, is often more complicated to prove. 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 $\text{deq}(v)$ definition: $\text{deq}()$ cannot return a result x in an execution precisely if $\text{deq}(x)$ cannot terminate in that same execution.

Theorem 6.3. *A completable queue implementation has no VFresh and VOrd violations iff for all $k \in \mathbb{N}$ and for all v_1 and v_2 such that $v_1 \neq v_2$, the $\text{deq}(v_2)$ thread does not terminate in the program*

$$\text{Prg}(k) \stackrel{\text{def}}{=} b \leftarrow \text{false}; (\text{deq}(v_2) \parallel (\text{enq}(v_1); b \leftarrow \text{true})) \parallel \overbrace{C \parallel \dots \parallel C}^{k \text{ threads}})$$

where

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

Proof. (\Rightarrow) We argue by contradiction. Consider an execution trace τ of $\text{Prg}(k)$ in which the $\text{deq}(v_2)$ thread terminates. If $\text{enq}(v_2)$ is not invoked in τ , then as there are no VFresh violations, we know that no $\text{deq}()$ in τ can return v_2 , contradicting our assumption that $\text{deq}(v_2)$ terminates in τ . Otherwise, if $\text{enq}(v_2)$ is invoked in τ , then at some earlier point $\text{assume}(b)$ was executed, and since initially b was set to false , this means that $b \leftarrow \text{true}$ was executed and therefore $\text{enq}(v_1) \prec_{h(\tau)} \text{enq}(v_2)$. Consequently, from POrd, if there is $\text{deq}()$ in τ returns v_2 , there must be a $\text{deq}()$ in τ that can be completed to return v_1 , contradicting our assumption that $\text{deq}(v_2)$ terminates in τ .

(\Leftarrow) We have two properties to prove. For VFresh, it suffices to consider the restricted parallel context that never enqueues v_2 . In this restricted context, $\text{deq}(v_2)$ does not terminate, and so $\text{deq}()$ cannot return v_2 . For VOrd, consider an execution trace in which every $\text{enq}(v_2)$ happens after some enqueue of a different value, say $\text{enq}(v_1)$, and in which there is no $\text{deq}(v_1)$. Such an execution trace can easily be produced by the unbounded parallel composition of C , and so $\text{deq}(v_2)$ also does not terminate, as required. \square

Showing Absence of VWit Violations. Here, we have to show that any dequeue event cannot return NULL if it never goes through a state where the queue could be 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_\perp which returns NULL and does not precede any other event in c .

Let c' be some prefix of c and let $e \in \text{Enq}(c')$ be a completed enqueue event in c' . We will call e *alive* after c' if there is a matching dequeue event d in $\text{Deq}(c)$, i.e. $d = \text{deq}(\text{Val}_c(e))$, then d is neither pending nor completed in c' . In other words, e is alive after c' if its matching dequeue d , if it exists, is not invoked in c' .

For the following, let d_i denote the dequeue event which removes the element inserted by the enqueue event e_i ; that is, $d_i = \text{deq}(\text{Val}_c(e_i))$. A sequence $e_0 e_1 \dots e_n$ of enqueue events in $\text{Enq}(c)$ is *covering* for d_\perp in c if the following holds:

- e_0 is alive at c' where c' is the maximal prefix of c in which $\text{inv}(d_\perp)$ does not occur.
- For all $i \in [1, n]$, e_i starts before d_\perp 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, with the only exception of d_n , which does not exist (the last condition). Then, the sequence is covering for d_\perp if d_0 does not start before d_\perp 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_\perp , 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 .

Lemma 6.4. *A (complete) concurrent history c has VWit iff it does not satisfy PWit.*

Proof. (\Rightarrow) Let c have VWit. By Prop. 4.8, there is $d_\perp \in \text{Deq}(c)$ such that $\text{Val}_c(d_\perp) = \text{NULL}$ and $\text{Bad}(c, d_\perp) \cap \text{Before}(c, d_\perp) \neq \emptyset$. We construct a covering sequence $e_0 \dots e_n$ for d_\perp such that for all $0 \leq i < n$ the response of e_i occurs before the response of e_{i+1} , if j_i and j_{i+1} are minimal indices for which $e_i \in \text{Bad}_{j_i}(c, d_\perp)$ and $e_{i+1} \in \text{Bad}_{j_{i+1}}(c, d_\perp)$ hold, then $j_{i+1} < j_i$, and $e_n \in \text{Bad}_0(c, d_\perp)$, and if $e \in \text{Bad}_k(c, d_\perp)$ with $k < j_{i+1}$, then $e \not\prec_c d_i$.

(Base): By the assumption there is an enqueue event in $\text{Bad}(c, d_\perp) \cap \text{Before}(c, d_\perp)$. Set e_0 an enqueue event in $\text{Bad}_{j_0}(c, d_\perp)$ such that for any other enqueue event $e' \in \text{Bad}_k(c, d_\perp) \cap \text{Before}(c, d_\perp)$, we have $j_0 \leq k$.

(Inductive): Let e_i be in $\text{Bad}_{j_i}(c, d_\perp)$ with $j_i > 0$. Let E' be the set of all $e' \in \text{Bad}(c, d_\perp)$ such that either $e' \prec_c e_i$ or $e' \prec_c d_i$, where d_i is the matching dequeue event for e_i . Observe that E' is non-empty. Choose $e_{i+1} \in E'$ to be an enqueue event with minimal index in E' . That is, if j_{i+1} is the smallest index for which $e_{i+1} \in \text{Bad}_{j_{i+1}}(c, d_\perp)$ holds, then for any $e' \in E'$, $e' \in \text{Bad}_k(c, d_\perp)$ implies $j_{i+1} \leq k$. Observe that $j_{i+1} < j_i$. This implies that by construction it cannot be the case that $e_{i+1} \prec_c d_{i-1}$ since it would contradict the assumption that e_i was chosen as an enqueue event with minimal index among those that precede d_{i-1} . But again by construction we have $e_i \prec_c d_{i-1}$ which implies that the response event of e_{i+1} occurs after the response event of e_i . This also means that because $e_{i+1} \not\prec_c e_i$, we must have $e_{i+1} \prec_c d_i$.

```

procedure deq( $v : val$ )
  while true do
     $\langle range \leftarrow q.back - 1 \rangle$ 
    for  $i = 0$  to  $range$  do
       $\left( \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \text{assume}(x = v \wedge x \neq \text{NULL}); \end{array} \right\rangle; \right) \sqcup \left\langle \begin{array}{l} x \leftarrow q.items[i]; \\ \text{assume}(x = \text{NULL}); \\ q.items[i] \leftarrow \text{NULL} \end{array} \right\rangle$ 
      return  $x$ 

```

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

Since the sequence of indices j_i is strictly decreasing, to show that the construction terminates with $j_n = 0$, we only have to show that there is $e_n \in \text{Bad}_0(c, d_\perp)$ completed before d_\perp is completed; i.e. the response of e_n occurs before the response of d_\perp in c . By the definition of VWit, taking $c_p = c_0 \cdot \text{inv}(d_\perp) \cdot c_d$, we know that there must be at least one enqueue event e in c_p such that e is completed in c_p and its matching dequeue is neither pending nor completed in c_p . But this immediately implies that $e \in \text{Bad}_0(c, d_\perp)$ and e is completed before d_\perp is completed.

(\Leftarrow) Let $e_0 \dots e_n$ be a covering sequence for d_\perp . Then, $e_n \in \text{Bad}(c, d_\perp)$ because d_n if it exists is preceded by d_\perp , i.e. $d_\perp \prec_c d_n$. Furthermore, for every $i \in [1, n]$, since we have $e_i \prec_c d_{i-1}$, all $e_i \in \text{Bad}(c, d_\perp)$. Finally, $e_0 \in \text{Before}(c, d_\perp)$. Thus, $\text{Bad}(c, d_\perp)$ and $\text{Before}(c, d_\perp)$ are not disjoint if there is a covering for d_\perp . By Prop. 4.8 this implies the existence of VWit. \square

We will actually restate the same property in a simpler way by making the following observation.

Proposition 6.5. *There is a covering for d_\perp in c iff at every prefix c' of c such that d_\perp is pending in c' , 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 $\text{Val}_c(e_1) \neq \text{Val}_c(e_2)$, a dequeue event cannot return $\text{Val}_c(e_2)$ if e_1 is alive at c .

7. AUTOMATION WITHIN CAVE

To automate the linearizability proof of the HW queue, we have mildly adapted the implementation of CAVE [19], a sound but incomplete thread-modular concurrent program verifier that can handle dynamically allocated linked list data structures and 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 [20] with a rich shape-value abstract domain [18] that can remember invariants indicating 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 [19] for details), but this is not applicable to the HW queue and therefore irrelevant for our purposes.

Overview of Action Inference. In brief, CAVE’s action inference algorithm first determines the part of the heap-allocated memory that is private to a thread and the part that is shared. The main heuristic employed in this decision is that newly allocated memory cells are deemed to be private until they become reachable from some global variable, from which point onwards they are deemed shared.

Next, the algorithm computes a binary relation R on program states overapproximating the effects of all atomic statements of the program to the shared part of the heap. Syntactically, it represents R as the union of a set of more primitive binary relations, which are called *actions*. Moreover, it remembers which atomic program statements correspond to which actions of the set. Thus, for example, if we want to compute an overapproximation of a program C in a parallel context, C' , we can run action inference on $C||C'$ and from the total set of actions return only those corresponding to C .

As part of this overapproximation, any information about the program’s control flow is lost except when the program explicitly records it in some global variable. This property is common to most thread-modular reasoning techniques, and is necessary for scalability. Thus, for instance, the programs C , C^* , and $C||C$ generate the same set of actions.

In the process of computing the set of actions, CAVE proves that the program is memory safe and does not violate any assertions in it. To do so, it constructs a proof in RGSep, which is an adaptation of Jones’ rely-guarantee method suitable for pointer-manipulating programs [12, 21]. To construct these proofs, it calculates via abstract interpretation an invariant that holds after every atomic program statement. These invariants describe the shapes of the heap allocated data structures (e.g., that there is a linked list from x to y via the field `next`), and some very simple facts about the values stored in them (e.g., that the sequences of values stored in two list segments are equal, or that the sequence of values stored in one list segment is sorted).

Finally, we note that action inference is incremental. Typically, action inference is run starting with an initial empty set of actions, to which set it adds any new actions it generates until a fixpoint is reached. When, however, we want to verify $C||C'$ and we already know a sound abstraction of C (under the assumption that C' can be run in parallel), it suffices to perform action inference only on C' but starting with the set of actions of C' as the initial set of actions. To this set, action inference will add any further actions C produces.

Summary of Changes. The modifications we had to perform to CAVE were:

- (1) To add code that instruments `deq()` methods with a prophecy argument guessing its return value, thereby generating `deq(v)`;
- (2) To add some glue code that constructs the verification conditions of Theorems 6.2 and 6.3 and runs the underlying prover to verify them;
- (3) 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 Section 5; and
- (4) When checking the absence of `VRepet` violations, to instrument the inferred actions so as to work around the fact that action inference abstracts over control flow information.

The first two changes are clearly tool-independent, the third item is very CAVE-specific, whereas the fourth item is fairly generic. The problem that we are working around here is common to almost all thread-modular verification approaches, and our instrumentation should work for other tools as well. To use a different tool from CAVE, the tool must be able to express invariants such as the aforementioned (*) invariant.

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, VRepet, and VOrd violations. (As the HW dequeues never return NULL, the algorithm also trivially has no VWit violations.)

Prophetic Instrumentation of Dequeues. In order to be able to use the theorems in the previous section, we must first construct the method $\mathbf{deq}(v)$ that records the result of the $\mathbf{deq}()$ function in its arguments which acts like a prophecy variable. In essence, the $\mathbf{deq}(v)$ we construct must be such that the set of traces of $\bigsqcup_{x \in \mathbb{N} \cup \{\text{NULL}\}} \mathbf{deq}(x)$ is equal to the set of traces of $\mathbf{deq}()$, where \sqcup stands for non-deterministic choice. Figure 2 shows the resulting automatically-generated instrumented definition of $\mathbf{deq}(v)$ 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:

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

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

Instrumentation for Checking Absence of VRepet Violations. Observe that the HW queue implementation is data independent as the operations on the shared locations in the \mathbf{enq} and \mathbf{deq} methods do not depend on the value of argument. Therefore, using Theorem 6.2, we have to prove that in the context where only one $\mathbf{enq}(v)$ can happen in parallel, $\mathbf{deq}(v)$ cannot terminate if another $\mathbf{deq}(v)$ has terminated.

One slight complication is that we cannot use RGSep action inference [19] directly to prove this property because we have to keep track of the exact number of occurrences of particular shared memory operation (such as the enqueues of v). In rely-guarantee, operations on shared variables are abstracted by *actions*, which typically do not contain any control flow within them. Hence after the initial action generation, we have to augment the shared state and the actions with auxiliary variables that (a) record the termination of parallel $\mathbf{deq}(v)$ and (b) ensure that only one parallel $\mathbf{enq}(v)$ call is accounted for. Our implementation therefore proceeds as follows:

- (1) It infers an initial set of RGSep actions, R , by performing symbolic execution of the \mathbf{enq} and \mathbf{deq} methods, and refine this set of actions to record information about the arguments of $\mathbf{enq}()$ and the result of the $\mathbf{deq}()$ functions wherever possible. Let $R_{\mathbf{enq}}$ be the actions generated by \mathbf{enq} method and $R_{\mathbf{deq}}$ be those generated by \mathbf{deq} .

- (2) For each action that is executed at most once by an $\text{enq}(v)$ invocation, it generates a fresh auxiliary variable, e_i , and records that e_i changes from 0 to 1 by performing that action. Formally, we define:

$$\begin{aligned} E &\stackrel{\text{def}}{=} \{(\ell, A) \in R_{\text{enq}} \mid \ell \text{ occurs at most once on every path through } \text{enq}\} \\ R' &\stackrel{\text{def}}{=} \{(\ell, A \wedge e_\ell = 0 \wedge e'_\ell = 1) \mid (\ell, A) \in E\} \cup (R_{\text{enq}} \setminus E). \end{aligned}$$

writing e_ℓ and e'_ℓ for the freshly generated variables in the action's pre- and post-states. (The purpose of this instrumentation is to ensure that the E actions will not interfere more than once with $\text{deq}(v)$ below.)

- (3) Record each action that must be performed by a completed $\text{deq}(v)$ event using a fresh auxiliary variable, d_i . Formally,

$$\begin{aligned} D &\stackrel{\text{def}}{=} \{(\ell, A) \in R_{\text{deq}} \mid \ell \text{ must occur on every path through } \text{deq}\} \\ R'' &\stackrel{\text{def}}{=} \{(\ell, A \wedge d'_\ell = 1) \mid (\ell, A) \in D\} \cup (R_{\text{deq}} \setminus D). \end{aligned}$$

where d'_ℓ are the freshly generated variables in the action's post-state. (The purpose of this instrumentation is to be able to detect whether a deq operation has terminated.)

- (4) Running action inference with the following initial set of actions (the rely condition)

$$R'[v/arg] \cup R''[v/res] \cup \bigcup_{v' \neq v} (R_{\text{enq}}[v'/arg] \cup R_{\text{deq}}[v'/res]),$$

verify the Hoare triple

$$\{e_1 = \dots = e_n = d_1 = \dots = d_m = 0\} \text{ deq}(v) \{\exists i. d_i = 0\}.$$

The postcondition ensures that no other $\text{deq}(v)$ has terminated, because if it had, it must have set each $d_i = 1$.

8. RELATED WORK

Linearizability was first introduced by Herlihy and Wing [10], 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, 17], and which Schellhorn et al. [17] 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. [17] devote two full pages to explain it.

As a result, most work on automatically verifying linearizability (e.g. [2, 18, 19, 1, 6]) and some manual verification efforts (e.g., [4, 3]) have 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 [19]. Most recently, with their technique based on (automatically) rewriting implementations Dragoi et al. [6] have succeeded to extend this approach to some implementations with helping. Similar to their precursors, however, their approach also assumes the existence of static linearization points, i.e. instructions in the program code that when executed invariably correspond to the linearization of one or more methods. Thus, there are many implementations, as mentioned in the Introduction, that cannot be handled by this approach.

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.

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 [10], and (2) a mechanized backward simulation proof by Schellhorn et al. [17].

Both proofs are manually constructed. In comparison, our new proof is simpler, more modular, and automatically generated. 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.

9. 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 simpler correctness proof of the Herlihy and Wing queue [10], and one which can be produced automatically.

We believe that our new property-oriented approach to linearizability proofs will be applicable to other kinds of concurrent shared data structures, such as stacks, sets, and maps. The generalization, however, is not entirely straightforward. In the case of stacks, the violations are similar to that of queues, but not exactly dual. The main difference is that the ordering violation for stacks is similar to `VWit` and not to `VOrd` as one might

expect. Similarly, the violations for set implementations are also not as simple as dropping the ordering constraint. Instead, we need to count the number of successful insertions and deletions to express what can go wrong. It remains to be seen, however, whether such counting arguments can yield an automatic verification technique.

ACKNOWLEDGMENTS

We would like to thank the CONCUR'13 reviewers for their feedback. The research was supported by the EC FET FP7 project ADVENT, by the Austrian Science Fund NFN RISE (Rigorous Systems Engineering), by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling), and by the EPSRC Grants EP/H005633/1 and EP/K008528/1.

REFERENCES

- [1] P. A. Abdulla, F. Haziza, L. Holík, B. Jonsson, and A. Rezine. An integrated specification and verification technique for highly concurrent data structures. In N. Piterman and S. A. Smolka, editors, *TACAS 2013*, volume 7795 of *LNCS*, pages 324–338. Springer, 2013.
- [2] D. Amit, N. Rinetzky, T. Reps, M. Sagiv, and E. Yahav. Comparison under abstraction for verifying linearizability. In W. Damm and H. Hermanns, editors, *CAV 2007*, volume 4590 of *LNCS*, pages 477–490. Springer, 2007.
- [3] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *ENTCS*, 137(2):93–110, 2005.
- [4] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In M. Butler and W. Schulte, editors, *FM 2011*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.
- [5] S. Doherty and M. Moir. Nonblocking algorithms and backward simulation. In I. Keidar, editor, *DISC 2009*, volume 5805 of *LNCS*, pages 274–288. Springer, 2009.
- [6] C. Dragoi, A. Gupta, and T. A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV 2013*, pages 174–190. Springer-Verlag, 2013.
- [7] D. Hendler, I. Ince, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010*, pages 355–364. ACM, 2010.
- [8] T. A. Henzinger, A. Sezgin, and V. Vafeiadis. Aspect-oriented linearizability proofs. In P. R. D’Argenio and H. C. Melgratti, editors, *CONCUR 2013*, volume 8052 of *LNCS*, pages 242–256. Springer, 2013.
- [9] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.
- [10] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [11] M. Hoffman, O. Shalev, and N. Shavit. The baskets queue. In E. Tovar, P. Tsigas, and H. Fouchal, editors, *OPODIS 2007*, volume 4878 of *LNCS*, pages 401–414. Springer, 2007.
- [12] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.
- [13] E. Ladan-Mozes and N. Shavit. An optimistic approach to lock-free FIFO queues. In R. Guerraoui, editor, *DISC 2004*, volume 3274 of *LNCS*, pages 117–131. Springer, 2004.
- [14] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model checking linearizability via refinement. In A. Cavalcanti and D. Dams, editors, *FM 2009*, volume 5850 of *LNCS*, pages 321–337. Springer, 2009.
- [15] N. A. Lynch and F. W. Vaandrager. Forward and backward simulations: I. Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.
- [16] M. Moir, D. Nussbaum, O. Shalev, and N. Shavit. Using elimination to implement scalable and lock-free FIFO queues. In *SPAA 2005*, pages 253–262. ACM, 2005.
- [17] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In P. Madhusudan and S. A. Seshia, editors, *CAV 2012*, volume 7358 of *LNCS*, pages 243–259. Springer, 2012.
- [18] V. Vafeiadis. Shape-value abstraction for verifying linearizability. In N. D. Jones and M. Müller-Olm, editors, *VMCAI 2009*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.
- [19] V. Vafeiadis. Automatically proving linearizability. In T. Touili, B. Cook, and P. Jackson, editors, *CAV 2010*, volume 6174 of *LNCS*, pages 450–464. Springer, 2010.

- [20] V. Vafeiadis. RGSep action inference. In G. Barthe and M. V. Hermenegildo, editors, *VMCAI 2010*, volume 5944 of *LNCS*, pages 345–361. Springer, 2010.
- [21] V. Vafeiadis and M. J. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR 2007*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.
- [22] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *POPL 1986*, pages 184–193. ACM, 1986.