Journal of Computers Vol. 35 No. 5, October 2024, pp. 91-103
doi: 10.53106/199115992024103505007

Proving Linearizability of Concurrent Queues

Jie Peng[1,2], Tangliu Wen[1*], and Dongming Jiang[3]

[1] Department of Information Engineering, Gannan University of Science and Technology, Ganzhou, China
gnkj2024@163.com.cn

[2] Ganzhou Cloud Computing and Big Data Research Center, Ganzhou, China
wqtlglk@163.com.cn

[3] School of Big Data, Jiangxi Science & Technology Normal University, Nanchang, China
cctjdm@jxust.edu.cn

Received 16 April 2024; Revised 30 May 2024; Accepted 6 August 2024

Abstract. Linearizability is a commonly accepted correctness criterion for concurrent data structures. Concurrent queues are among the most fundamental concurrent data structures. In this paper, we present necessary and sufficient conditions for proving linearizability of concurrent queues, which make use of linearization of dequeue operations. The verification conditions intuitively express the "FIFO" semantics of concurrent queues and can be verified just by reasoning about the happened-before order of operations. We have successfully applied the proof technique to prove several challenging concurrent queues. We believe that our proof technique can be extended to the concurrent data structures which have the ordering requirements when their elements are removed.

Keywords: concurrent queues, linearizability, verification, happened-before order

1 Introduction

The development of computer hardware technology has led to an era of multi-core CPU. Concurrent data structures are widely used to provide high performance for applications running on multi-core computers. Linearizability [1] is the standard correctness criterion for concurrent data structures. It requires that every concurrent execution of a concurrent data structure is equivalent to a legal sequential execution of an abstract object (called specification). This means that every method appears to take effect atomically at some point, during its execution interval. To obtain multi-core scalability and performance, highly-concurrent data structures employ sophisticated synchronization techniques, which makes proving their linearizability more difficult.

The typical proof of linearizability is based on forward or backward simulations. The proofs involve constructing an abstraction map, which relates the state of the implementation to the state of the specification and showing that their executions stay in correspondence under forward or backward program steps. Forward simulation alone is not sufficient in general to verify linearizability, which cannot handle with sophisticated concurrent data structures, such as the LCR queue. Backward simulation is a complete proof method for verifying linearizability. However, the proofs are difficult to understand intuitively and require considerable expertise.

In this paper, we propose a sound proof system that simplifies reasoning about linearizability of concurrent queues. Our basic idea is that for any linearizable execution of a concurrent queue, every dequeue operation must remove one of the values inserted by the oldest push operations. Building on this idea, we first present a set of necessary and sufficient conditions for proving linearizability of concurrent queues. The verification conditions are based on linearization of dequeue operations and intuitively characterize the "FIFO" semantics of concurrent queues. Informally, the conditions say that an execution of a concurrent queue is linearizable if there exists a linearization of dequeue operations such that every dequeue operation removes one of the values inserted by the oldest push operations. We need to construct the appropriate linearization of dequeue operations when applying the conditions to linearizability verification of concurrent queues. We observed an interesting phenomenon: for all concurrent queues [1-6] we have verified, their dequeue operations have the atomic actions which logically or physically remove elements of the queues. We further show that the removing actions can be used to construct

* Corresponding Author

such linearization.

Our proof technique is intuitive, easy-to-use, and verifiers just need to reason about the happened-before order of operations. We have successfully applied the proof technique to several challenging concurrent queues, including the TS queue, the LCR queue. To summarize, our contributions are:

1. We present a set of necessary and sufficient conditions for proving linearizability of concurrent queues.
2. We show that the removing actions of dequeue operations can be used to construct linearization of dequeue operations.
3. We apply our proof technique to prove linearizability of several challenging concurrent queues.

The rest of the paper is structured as follows: Section 2 recalls the definition of linearizability and partially ordered sets; Section 3 formalizes the aforementioned properties, and proves that they are necessary and sufficient conditions for proving linearizability of concurrent queues; Section 4 explains that the removing actions of dequeue operations can be used to construct linearization of dequeue operations. Section 5 returns to the LCR queue and the time-stamped queue examples and presents a detailed manual proof of their correctness; Section 6 discusses related work, and Section 7 concludes the paper.

2 Preliminaries

In this section, we formalize linearizability of a concurrent data structure with respect to its specification [7-9], and introduce basic notations of partially ordered sets.

2.1 Linearizability

We refer to an execution of a method as an operation. The calling of a method m with argument v is represented as an invocation event $inv_o m(v)$, and the return of a method with a return value v by a response event $ret_o(v)$, where o is an operation identifier. A thread executing a method starts with the invocation event, executes its internal atomic actions until the final response event. We denote a concurrent execution as a finite sequence of totally ordered atomic events. A history of a concurrent data structure is a sequence of invocation and response events generated by an execution of the data structure. An invocation event matches a response event if they belong to the same operation. A history is sequential if every invocation event, except possibly the last, is immediately followed by its matching response event. A history is complete if every invocation event has a matching response event. An invocation event is pending in a history if there is no matching response event to it. For an incomplete history H, a completion of H, is a complete history gained by adding some matching response events to the end of H and removing some pending invocation events within H. Let $Compl(H)$ be the set of all completions of the history H.

Let $\prec_H$ denote the happened-before order of operations in the history H. For any two operations op_1 and op_2 of H, we say that op_1 precedes op_2, denoted $op_1 \prec_H op_2$, if the response event of op_1 precedes the invocation event of op_2 in H; we say that op_1 is interleaved with op_2, denoted by $op_1 \simeq_H op_2$, if $op_1 \not\prec_H op_2$ and $op_2 \not\prec_H op_1$. We sometimes omit the subscripts when the histories are clear from the context.

A sequential history is legal with respect to a sequential specification if it satisfies the sequential specification behavior. For example, the sequence: $enq(a), enq(b), deq(a), enq(c), deq(b)$, is legal with respect to the standard sequential "FIFO" queue specification, where $enq(a)$ denotes an enqueue operation with an input parameter a; $deq(a)$ denotes a dequeue operation with a return value a.

A history H is linearizable with respect to a sequential specification [1] if there exists a complete history $C \in Compl(\mathrm{H})$ and a legal sequential history S such that (1) S is a permutation of C; (2) for any two operations op_1, op_2, if $op_1 \prec_C op_2$, then $op_1 \prec_S op_2$. S is called a linearization of H. The second condition above requires that S preserves the happened-before orders of the operations in H. A concurrent data structure is linearizable with respect to its sequential specification if every history of the concurrent data structure is linearizable with respect to the sequential specification.

Generally, the standard sequential "FIFO" queue is used to characterize the sequential specification of concurrent queues. For a linearizable queue with respect to the standard sequential specification, we sometimes omit the

sequential specification for simplicity.

In this paper, we only consider complete histories. As Henzinger et al. have shown [10], a purely-blocking data structure is linearizable if every complete history of the concurrent data structure is linearizable. Purely blocking is a very weak liveness property, and most of concurrent data structures satisfy the liveness property. All concurrent queues verified in this paper are purely blocking.

2.2 Partially Ordered Sets

A strict partial order on a set is an irreflexive, antisymmetric, and transitive relation. Obviously, the happened-before order $\prec_H$ is a strict partial order on the set of the operations of H. We say that y is bigger than x with respect to a strict partial order $\prec$ if $x \prec y$. Let $\prec$ be a strict partial order on the set S and $\mathrm{x} \in \mathrm{S}$; x is a maximal element of S if $\forall \mathrm{y} \in \mathrm{S}. x \not\prec y$; x is a minimal element of S if $\forall y \in S. y \not\prec x$; x is the greatest element of S if $\forall y \in S - x. y \prec x$; x is the smallest element of S if $\forall y \in S - x.\ x \prec y$. Let $\prec_1$ and $\prec_2$ be two partial orders on a set S; the partial order $\prec_2$ is called an extension of partial order $\prec_1$ if, whenever $\mathrm{a} \prec_1 b$, then $\mathrm{a} \prec_2 b$.

A total order $\prec$ is a linear order if $\forall \mathrm{x}, y \in S. x \prec y \ \vee \ y \prec \mathrm{x}$. If a total order is an extension of a partial order, then it is called a linear extension of the partial order.

Lemma 1 Let $\prec$ be a strict partial order on the set S, assume the sequence $L_1, L_2, \ldots, L_n$ (where $L_i \in S,\ 1 \leq i \leq n$) preserves the partial order $\prec$ (i.e., $\forall x, y.\ 1 \leq x \leq n \wedge 1 \leq y \leq n \wedge x < y \Longrightarrow L_y \not\prec L_x$). Then for any element $L' \in S$, L' can be inserted into the sequence such that the new sequence still preserves the partial order $\prec$.

In Appendix A, we show an algorithm by which L' can be inserted into a proper position such that the new sequence preserves the partial order.

3 Conditions for Linearizability of Concurrent Queues

In this section, we first give the basic technical setting including a formal operational definition of safe matching and linearization of dequeue operations. Then, we present our main theorem which gives necessary and sufficient conditions for proving linearizability of concurrent queues.

3.1 Safe Matching and Linearization of Dequeue Operations

Let $Enq(H)$ and $Deq(H)$ denote the sets of all enqueue and dequeue operations in a history H of a concurrent queue, respectively. For simplicity, we assume that all values which are added by enqueue operations are unique. We map each dequeue operation to the enqueue operation whose value is removed by the dequeue operation, or to ϵ if the dequeue operation returns $empty$. We say that a mapping is safe if a dequeue operation always returns the value which is added by an enqueue operation or $empty$; the value which is added by an enqueue operation is removed at most once. Obviously, for any history of concurrent queues, if there does not exist a safe matching, then the history is not linearizable. Every linearizable history has a unique safe matching. We formalize the notion as follows.

Definition 1. A mapping $Match$ from $Deq(H)$ to $Enq(H) \cup \epsilon$ is safe if

1. $\forall dequeue \in Deq(H)$. if $Match(dequeue) \neq \epsilon$, then the return value of the dequeue operation is added by the enqueue operation $Match(dequeue)$.
2. $\forall dequeue \in Deq(H)$. if $Match(dequeue) = \epsilon$, then the dequeue operation returns $empty$.
3. $\forall\ dequeue \in Deq(H),\ \forall\ dequeue' \in Deq(H)$. if $dequeue \neq dequeue' \wedge Match(dequeue) \neq \epsilon$, then $Match(dequeue) \neq Match(dequeue')$.

A linearization of dequeue operations is a sequence of dequeue operations which preserves the happened-before order of non-overlapping operations in the original execution. Our main theorem is based on linearizations of dequeue operations, which is defined as follows.

Definition 2. For a history H of a concurrent queue, the sequence $deq_1, \ldots, deq_n$ is a linearization of dequeue operations of H, if $Deq(H) = \{deq_1, \ldots, deq_n\} \wedge x < y \Longrightarrow deq_y \prec_H deq_x$.

3.2 Conditions for Linearizability of Concurrent Queues

Any history comprising only the events of enqueue operations is always linearizable. The reason is that linearizability is a property of externally observable behaviors (i.e., histories) and the return value of an enqueue operation is always $null$ or the signal ok. Such histories can be ignored when we verify linearizability of concurrent queues. Our queue theorem is stated below.

Theorem 1. Let H be a complete history of a concurrent queue containing the events of dequeue operations. H is linearizable with respect to the standard sequential queue specification iff there exists a linearization of dequeue operations $deq_1, deq_2, \ldots, deq_n$, and a safe mapping $Match$, such that:

1. $1 \leq \forall i \leq n$. if $Match(deq_i) \neq \epsilon$, let the set $PPA = Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}$. then $\forall enq_x \in PPA.\ enq_x \prec_H Match(deq_i)$ and $i \leq \forall j \leq n.\ deq_j \prec_H Match(deq_i)$;
2. $1 \leq \forall i \leq n$. If $Match(deq_i) = \epsilon$, let $PPN = \{enq \mid enq \simeq deq_i \wedge enq \in Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}\}$, then (1) $\forall enq.\ enq \prec_H deq_i \Longrightarrow enq \in \{Match(deq_1), \ldots, Match(deq_{i-1})\}$; (2) $\forall enq \in PPN, x \leq i - 1.\ enq \prec_H deq_x$.

For a dequeue operation deq_i, PPA is a set of the enqueue operations whose values have not been removed by the dequeue operations ahead of deq_i (i.e., the dequeue operations: $deq_x, 1 \leq x < i$). Informally, the first condition requires that each non-empty dequeue operation (i.e., it does not return $empty$) deq_i always removes the value of a minimal enqueue operation (w.r.t. $\prec_H$) in the set PPA.

PPN is a set of the enqueue operations which are interleaved with deq_i and whose values have not been removed by the dequeue operations ahead of deq_i. The second condition requires that for any enqueue operation which precedes the empty dequeue operation deq_i, the value of the enqueue operation is removed by a dequeue operation ahead of deq_i; for any enqueue operation in the set PPN, the enqueue operation does not precede any dequeue operation ahead of deq_i.

The following proof of Theorem 1 is written in a hierarchically structured style as advocated by Lamport [11].

Proof ($\Longrightarrow$). We first prove that the theorem holds when H does not contain the empty dequeue operations, then further extend the result to the case where H contains this kind of dequeue operations.

1. H is linearizable when H does not contain the empty dequeue operations.

Proof. We construct a sequential history H' by inserting every enqueue operation of H into the sequence $deq_1, deq_2, \ldots, deq_n$, and show that H' is a linearization of H. H' is constructed by the following steps:

Step 1. $1 \leq \forall i \leq n$, let $enq_i = Match(deq_i)$, we first insert $enq_1, enq_2, \ldots, enq_n$, one after another, into the sequence $deq_1, deq_2, \ldots, deq_n$. For enq_1, we insert it before deq_1. Since for all dequeue operation deq, $deq \prec_H enq_1$ (by the first condition of Theorem 1), the new sequence preserves the happened-before order $\prec_H$ after the inserting operation.

We can insert enq_2 between enq_1 and deq_2, by using Algorithm 1 (in Appendix A), i.e., if $deq_1 \prec_H enq_2$, insert it into the right of deq_1 (i.e., $enq_1, deq_1, enq_2, deq_2$); otherwise, insert it into the left of deq_1 (i.e., $enq_1, enq_2, deq_1, deq_2$). Since $enq_2 \prec_H enq_1$ and $\forall x \geq 2.\ deq_x \prec_H enq_2$, after the inserting operation, the new sequence (either $enq_1, deq_1, enq_2, deq_2, deq_3, \ldots$ or $enq_1, enq_2, deq_1, deq_2, deq_3, \ldots$) preserves the happened-before order $\prec_H$.

Similarly, for each enq_i, $2 < i \leq n$, we insert enq_i between enq_{i-1} and deq_i, by using Algorithm 1. After inserting enq_i, we get the following properties.

(1) In terms of the first condition of Theorem 1, we get that $enq_i \prec_H enq_{i-1}, \ldots, enq_2 \prec_H enq_1$. Thus, in the new sequence, any two enqueue operations do not violate the happened-before order $\prec_H$ (i.e., for any two enqueue operations op_1 and op_2, if op_1 precedes op_2 in the new sequence, then $op_2 \prec_H op_1$).

(2) enq_i does not violate the happened-before order with the dequeue operations on the left of enq_{i-1} (if any). The reason for this is as follows. Assume deq_y is a dequeue operation on the left of enq_{i-1}. By

Algorithm 1, there exists an enqueue operation enq_x, $x \le i-1$ such that $deq_y \prec_H enq_x$. Since $enq_i \not\prec_H enq_x$, $enq_i \not\prec_H deq_y$.

(3) In terms of the first condition of Theorem 1, we get that $\forall j \ge i.\ deq_j \not\prec_H enq_i$, thus, enq_i does not violate the happened-before order with the dequeue operations on the right of deq_i. Thus, after the inserting actions, the new sequence preserves the happened-before order $\prec_H$.

Step 2. Assume that the sequence $enq_{(n+1)}, \dots, enq_{(n+m)}$ is a linear extension of the partial order $\prec_H$ on the rest of $Enq(H)$ (i.e., there is no dequeue operation which is mapped to $enq_{(n+i)}$). We insert the enqueue operations $enq_{(n+1)}, \dots, enq_{(n+m)}$, one after another, into the new sequence constructed by step 1.

For each $1 \le i \le m$, we insert $enq_{(n+i)}$ between $enq_{(n+i-1)}$ and the end of the sequence, by using Algorithm 1. Since $enq_{(n+m)} \not\prec_H enq_{(n+m-1)}, \dots, enq_2 \not\prec_H enq_1$, and $enq_{(n+i)}$ does not violate the happened-before order with the dequeue operations on the left of $enq_{(n+i-1)}$ (we can get it, similar to the above proof), after the inserting action, the new sequence preserves the happened-before order $\prec_H$.

By the process of constructing H', H' preserves the happened-before order $\prec_H$, and satisfies the "FIFO" sequential semantics. Thus, the sequential history H' is a linearization of H.

2. H is linearizable when H contains the empty dequeue operations.

Proof. We construct the linearization of H by the following process. If $Match(deq_i) = \epsilon$, let A denote the linearization of $deq_1, \dots, deq_{i-1}$ and their matching enqueue operations (constructed by the above method), let B denote the linearization of the other operations of H. Let $H' = A^\frown deq_i{}^\frown B$, where the notation $^\frown$ denotes the concatenation of sequences. Obviously, any two dequeue operations of H' do not violate the happened-before order $\prec_H$.

In the following, we show that in H' (1) any two enqueue operations do not violate the happened-before order $\prec_H$ and (2) any dequeue operation does not violate the happened-before order $\prec_H$ with any enqueue operation. Let enq_x and deq_x be an enqueue operation and a dequeue operation in A, respectively. Let enq_y and deq_y be an enqueue operation and a dequeue operation in B, respectively. Since $enq_{(n+m)} \not\prec_H, \dots, enq_2 \not\prec_H enq_1$, $enq_y \not\prec_H enq_x$. By the first condition of Theorem 1, we get $deq_y \not\prec_H enq_x$. If $enq_y \simeq deq_i$, by the second condition of Theorem 1, $enq_y \not\prec_H deq_x$. If $deq_i \prec_H enq_y$, obviously, $enq_y \not\prec_H deq_x$.

Proof ($\Longleftarrow$). Since H is linearizable, there exists a safety mapping $Match$ from $Deq(H)$ to $Enq(H) \cup \epsilon$. We assume that H' is a linearization of H. Let $deq_1, deq_2, \dots, deq_n$ be the maximal subsequence of H' consisting of dequeue operations. Obviously, it is a linearization of the dequeue operations of H. Based on the linearization of the dequeue operations and the safety mapping $Match$, we show that the two conditions of Theorem 1 hold.

1. $1 \le \forall i \le n$. if $Match(deq_i) \ne \epsilon$, let the set $PPA = Enq(H) - \{Match(deq_1), \dots, Match(deq_{i-1})\}$, then $\forall enq_x \in PPA.\ enq_x \not\prec_H Match(deq_i)$ and $i \le \forall j \le n.\ deq_j \not\prec_H Match(deq_i)$.

Proof. Since H' is a sequential execution and satisfies the "FIFO" semantics, $Match(deq_i) \prec_{H'} enq_x$. Since the linear order $\prec_{H'}$ is a linear extension of $\prec_H$, $enq_x \not\prec_H Match(deq_i)$. Since $Match(deq_i) \prec_{H'} deq_i$, and $deq_i \prec_{H'} deq_j$, $deq_j \not\prec_H Match(deq_i)$.

2. $1 \le \forall i \le n$. If $Match(deq_i) = \epsilon$, let $PPN = \{enq \mid enq \simeq deq_i \wedge enq \in Enq(H) - \{Match(deq_1), \dots, Match(deq_{i-1})\}\}$, then (1) $\forall enq.\ enq \prec_H deq_i \Longrightarrow enq \in \{Match(deq_1), \dots, Match(deq_{i-1})\}$; (2) $\forall enq \in PPN, x \le i-1.\ enq \not\prec_H deq_x$.

Proof. If an enqueue operation $enq \prec_H deq_i$, then $enq \prec_{H'} deq_i$. Since H' satisfies the "FIFO" semantics, the value inserted by the enqueue operation enq is removed by a previous dequeue operation, i.e., $enq \in \{Match(deq_1), \dots, Match(deq_{i-1})\}$.

If an enqueue operation $enq \in PPN$, then $deq_i \prec_{H'} enq$. Since $\forall x \le i-1,\ deq_x \prec_{H'} deq_i$, $deq_x \prec_{H'} enq$. Thus, $enq \not\prec_H deq_x$ (since $\prec_{H'}$ is a linear extension of $\prec_H$).

Example 1 Consider the following history H. Next, we will prove linearizability of the execution using the above theorem. For simplicity, let $e_i(x)$ denote the invocation event of an enqueue operation e_i with an input parameter x, and e_i' be its matching response event; d_i () denote the invocation event of a dequeue operation d_i, $d_i(y)'$ be its matching response event with a return value y.

$(e_1(a), e_2(b), e_2', d_1(), d_2(), d_1(a)', d_3(), e_3(c), e_1', e_4(d), e_3', e_4', d_2(b)', d_3(d)')$

The history has a unique safe matching, $Match(d_1) = e_1, Match(d_2) = e_2, Match(d_3) = e_4$. We choose the linearization of dequeue operations, d_1, d_2, d_3 to verify the linearizability. d_1 removes the value inserted by e_1 (i.e., $Match(d_1) = e_1$), e_1 is minimal in the set $\{e_1, e_2, e_3, e_4\}$ and $d_1 \prec_H Match(d_1)$. Thus, for the dequeue operation d_1, the first condition is satisfied. Since $d_2 \prec_H Match(d_1) = e_2$ and e_2 is minimal in the set $\{e_2, e_3, e_4\}$, the first condition is satisfied in this case. Similarly, d_3 also satisfies the first condition. Thus, the history is linearizable.

4 Constructing Linearization of Dequeue Operations

We need to construct appropriate linearization of dequeue operations when applying the queue theorem to linearizability verification of concurrent queues. For all concurrent queues we have verified [1-6], the atomic actions of dequeue operations which logically or physically remove values can be used to construct such linearizations. If there exists a logical removing action in a non-empty dequeue operation, then the removing action is chosen for constructing linearization; otherwise, the physical removing action is chosen. The physical removing action of a dequeue operation physically removes a value in the queue. The logical removing action of a dequeue operation only fixes a value in the queue, after the execution of the logical removing action, other dequeue operations cannot logically remove the value. A linearization of dequeue operations constructed by using their removing actions is a sequence where the dequeue operations are arranged in the execution order of these removing actions. Obviously, the initial linearization of the dequeue operations can be easily constructed in terms of the removing actions. We show that in general case the removing actions of non-empty dequeue operations can be used to construct such linearization.

For simplicity, we only consider the executions containing two dequeue operations where their removing actions cannot be used to construct such linearization. Two basic example executions are shown in Fig. 1 and Fig. 2.

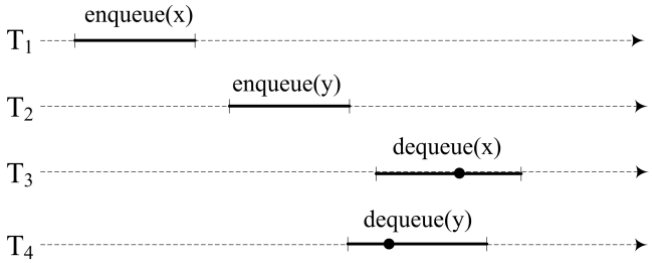

Fig. 1. *dequeue*(*x*) begins to execute before the removing action of *dequeue*(*y*)

In these figures, the black circles of the dequeue operations stand for the logical or physical removing actions; $enqueue(x)/enqueue(y)$ denotes the enqueue operation with an input parameter x/y; $dequeue(x)/dequeue(y)$ denotes the dequeue operation with a return value x/y.

In Fig. 1, $dequeue(x)$ begins to execute before the removing action of $dequeue(y)$. The only linearization of the execution is $enqueue(x), enqueue(y), dequeue(x), dequeue(y)$. The linearization of the two dequeue operations constructed in terms of the two removing actions is $dequeue(y), dequeue(x)$. Under the linearization of the two dequeue operations, the first condition of Theorem 1 is not satisfied. Thus, the two removing actions cannot be used to construct linearization of dequeue operations.

If there is no $dequeue(x)$, then the dequeue operation of the thread T_4 will remove the value x inserted by $enqueue(x)$, to make the execution linearizable. Thus, the $dequeue(x)'s$ actions before the $dequeue(x)'s$ removing action affect the execution of the dequeue operation of the thread T_4, and prevent it observing the value inserted by $enqueue(x)$. Such dequeue methods are uncommon. Generally, for a lock-free concurrent queue,

except for the logical and physical removing actions, the actions do not prevent the values of the queue from being removed by other dequeue operations. For most of concurrent queues we have verified, the actions before the removing action of a dequeue operation either read the shared state or access (read or write) the local state, and do not affect the executions of other dequeue operations.

In Fig. 2, $dequeue(y)$ removes the value y before $dequeue(x)$ begins to execute. If there is no $dequeue(x)$, then $dequeue(y)$ also removes the value y. In this case, the execution of the three operations is not linearizable. Thus, such dequeue algorithms are basically nonexistent.

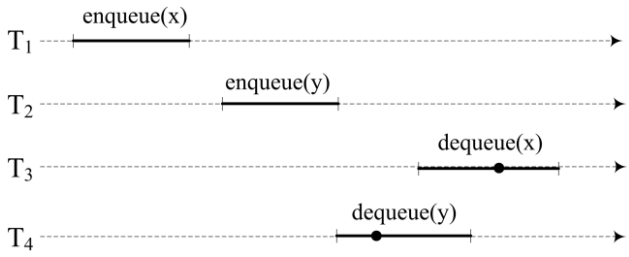

Fig. 2. *dequeue*(*x*) begins to execute after the removing action of *dequeue*(*y*)

5 Examples

We illustrate our technique on the LCR queue and the TS queue. Proving linearizability of the two queues are challenging because their enqueue methods do not have fixed linearization points. The linearization points of the two enqueue methods depend on future executions of dequeue operations.

5.1 Verifying the LCR Queue

The LCR queue [4] shown in Fig. 3 is represented as an infinite size array, *item*, and two markers, $head$ and tail, pointing to the head and end of the interval of the array that may contain values, respectively. The queue reserves two special values $\perp$ and $\top$ that are distinct from any element enqueued by the enqueue operations. Each cell $items[i]$ is initialized to the reserved value $\perp$.

An enqueue operation first obtains a cell index t by performing a Fetch-and-Add (FAA) atomic instruction (line L_7) on $tail$, which returns the value t of $tail$ and changes $tail's$ value to $t + 1$. Then it updates the value of $items[t]$ to x by the Swap atomic instruction (line L_8), which returns the value of $items[t]$ and changes $items[t]'s$ value to x. If the Swap instruction returns $\perp$, the enqueue operation returns ok; otherwise (this means that the Swap instruction returns $\top$), the enqueue operation tries again.

```
L0  class LCRQueue{
L1    int head=1;
L2    int tail=1;
L3    int[ ] items;
L4  }
L5  void enqueue(x: Object){
L6    while(true) {
L7      t=FAA(&tail, 1)
L8      if(Swap(items[t], x)==⊥)
L9        return OK;
L10   }
L11 }
```

```
L12 int dequeue( ){
L13   while(true) {
L14     h=FAA(&head, 1)
L15     x=Swap(items[h], ⊤)
L16     if(x ≠ ⊥)
L17       return x;
L18     if(tail ≤ h + 1)
L19       return EMPTY;
L20   }
L21 }
```

Fig. 3. the LCR queue

A dequeue operation first obtains a cell index h by performing a FAA atomic instruction on $head$. Then it updates the value of $items[h]$ to $\top$ by the Swap instruction. If the Swap instruction returns a value $x \neq \perp$, then the dequeue operation completes and returns x; otherwise (this means that the Swap instruction returns $\perp$), (1) if $tail \leq h+1$, the dequeue operation returns empty; (2) otherwise, the dequeue operation tries again.

If an enqueue operation deposits a value x into the cell which does not contain $\perp$ (this means that the cell contains $\top$, and has been visited by some dequeue operation), the value x stored in the cell will not be dequeued by any dequeue operation. Thus, an enqueue operation deposits only one value which may be dequeued. Since a cell is updated by no more than one dequeue operation, the value in a cell is dequeued at most once. Thus, for a complete history H of the LCR queue, there is a safe mapping $Match$ from $Deq(H)$ to $Enq(H)$ and ϵ.

Let $enq.Faa/deq.Faa$ denote the last FAA instruction of the enqueue/dequeue operation enq/deq. Let $\prec_o$ denote the happened-before order of atomic actions in an execution. For the enqueue/dequeue operation enq^x / deq^x, let the superscript x denote the return value of $enq.Faa/deq.Faa$. Obviously, there are the following properties:

For two enqueue operations enq^x and enq^y in a history H, if $x < y$, then $enq^y \prec_H enq^x$; for two dequeue operations deq^x and deq^y in a history H, if $x < y$, then $deq^y \prec_H deq^x$; if $Match(deq^y) = enq^x$, then $x = y$.

The last FAA action $h := FAA(\&head, 1)$ of a dequeue operation is a logical removing action. After the last FAA action of a non-empty dequeue operation is executed, $items[h]$ is logically removed, i.e., other dequeue operations cannot logically or physically remove $items[h]$ again.

Theorem 2. Every complete history H of the LCR queue is linearizable with respect to the standard sequential queue specification.

Proof. Assume $deq_1^{h_1}, deq_2^{h_2}, \ldots, deq_n^{h_n}$ is the linearization of the dequeue operations constructed in terms of their logical removing actions. Based on the linearization and the safe mapping $Match$, we show that the LCR queue satisfies the two conditions of Theorem 1.

1. $1 \leq \forall i \leq n$. if $Match(deq_i) \neq \epsilon$, let the set $PPA = Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}$. then $\forall enq_x \in PPA.\ enq_x \prec_H Match(deq_i)$ and $i \leq \forall j \leq n.\ deq_j \prec_H Match(deq_i)$.

Proof. By the removing actions of dequeue operations, we get $h_1 <, \ldots, < h_n$. By $(deq_i)^{h_i}$, we get the return value of the last FAA instruction of $Match(deq_i)$ is h_i. Assume the return value of the last FAA instruction of enq_x is t_x. Consider the following two cases: If $enq_x^{t_x} \in \{Match(deq_{i+1}^{h_{i+1}}), \ldots, Match(deq_n)^{h_n}\}$, then $h_{i+1} \leq t_x$. By $h_i < h_{i+1} \leq t_x$, we get $enq_x \prec_H Match(deq_i)$. If $enq_x^{t_x} \in Enq(H) - \{Match(deq_1), \ldots, Match(deq_n)\}$, then $h_n < t_x$. By $h_i < h_n < t_x$, we get $enq_x \prec_H Match(deq_i)$.

By $h_i < h_j$, we can get $Match(deq_i).Faa \prec_o Match(deq_j).Faa$. Let $deq_j.Swap$ denote the last Swap instruction of deq_j. By $Match(deq_j).Faa \prec_o deq_j.Swap$, we can get $Match(deq_i).Faa \prec_o deq_j.Swap$. Thus, $deq_j \prec_H Match(deq_i)$.

2. $1 \leq \forall i \leq n$. If $Match(deq_i) = \epsilon$, let $PPN = \{enq \mid enq \simeq deq_i \wedge enq \in Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}\}$, then (1) $\forall enq.\ enq \prec_H deq_i \Longrightarrow enq \in \{Match(deq_1), \ldots, Match(deq_{i-1})\}$; (2) $\forall enq \in PPN,\ x \leq i-1.\ enq \prec_H deq_x$.

Proof. Since $Match(deq_i) = \epsilon$, $tail \leq h_i + 1$ is true at the time point while the last statement if (tail ≤ h + 1) of deq_i is executed. Assume $enq^{t_x} \prec_H deq_i$. Since at the above time point, $t_x < tail \leq h_i + 1$, $t_x \leq h_i$. Since $enq^{t_x} \prec_H deq_i$ and $Match(deq_i) = \epsilon$, $t_x \neq h_i$. Thus, $t_x < h_i$. Since all cells from $itmes[1]$ to $items[h_i]$ have been updated by dequeue operations before deq_i, $enq^{t_x} \in \{Match(deq_1), \ldots, Match(deq_{i-1})\}$.

Assume $enq_y^{t_y} \in PPN$, we can get $t_y > h_i$. If $enq_y^{t_y}.Faa(\&tail, 1) \prec_o deq_i^{h_i}.Faa(\&head, 1)$, then $tail > t_y + 1 > h_i + 1$ at the time point while the last statement if (tail ≤ h + 1) of deq_i is executed. This contradicts the fact: $tail \leq h_i + 1$ is true at the above time point. Thus, $deq_i^{h_i}.Faa(\&head, 1) \prec_o enq_y^{t_y}.Faa(\&tail, 1)$. Since $deq_x^{h_x}.Faa(\&head, 1) \prec_o deq_i^{h_i}.Faa(\&head, 1)$, $deq_x^{h_x}.Faa(\&head, 1) \prec_o enq_y^{t_y}.Faa(\&tail, 1)$. Thus $enq_y^{t_y} \prec_H deq_x$.

5.2 Verifying the Time-Stamped Queue

Fig. 4 shows the pseudo code for the time-stamped (TS) queue [5]. We use $>_{ts}$ operator for timestamp comparison. For two timestamps t_1 and t_2, we say that t_2 is bigger than t_1, if $t_2 >_{ts} t_1$; t_1 and t_2 are incomparable, if $t_1 \not>_{ts} t_2$ and $t_2 \not>_{ts} t_1$. For two operations op_1 and op_2, $op_2 >_{ts} op_1$ if the timestamp generated by the operation op_2 is bigger than the one generated by the operation op_1. Let $\top$ denote the maximal value of timestamps. There are a number of implementations of the time stamping algorithm. All these implementations guarantee that (1) in a sequential execution of two calls to the algorithm, the latter returns a bigger timestamp than the former and (2) a concurrent and overlapping execution of two calls to the algorithm generates two incomparable timestamps.

This queue maintains an array $pools$ of singly-linked lists (i.e., instances of $SPPool$), one for each thread. Each node of the list contains a data value (field val), a timestamp (field $timestamp$), a next pointer (field $next$). Each list contains a $tail$ pointer which points to the end of the list, a $head$ pointer which points to the first node (a sentinel node) of the list. Initially both the $head$ and $tail$ pointers point to the sentinel node indicating that the list is empty. The $head$ pointer of a list is annotated with an ABA-counter to avoid the ABA-problem [12]. The methods of the $SPPool$ list are as follows.

1. insert(v) - inserts a node with a value v and a timestamp $\top$, to the end of the list and returns a reference to the new node.
2. getOldest - returns a reference to the node with the oldest timestamp, or $null$ if the list is empty, together with the top pointer of the list.
3. remove(node) - tries to remove the given node from the list. Returns $true$ and the value of the node if it succeeds, or returns $false$ and $null$ otherwise.

These methods of $SPPool$ are linearizable, and can be viewed as atomic actions. The Enqueue method first inserts an element into its associated list (line E_3), then generates a timestamp (line E_4) and sets the timestamp field of the new node to the new timestamp (line E_5).

The dequeue method first generates a timestamp $startTS$ (line D_4), attempts to remove an element by calling the method tryRem. The tryRem method traverses every list, searching for a node with a minimal timestamp to remove (line T_7-T_{20}). Note that the search starts from a random list, to make different threads more likely to pick different elements for removal and reduce data contention. If the timestamp $minimalTS$ of the candidate node $oldestNode$ is bigger than the timestamp $startTS$ (line T_{29}), the candidate is invalid, the tryRem method returns $false$ (line T_{30}), then the dequeue method restarts. If the candidate node $oldestNode$ is valid, the tryRem method tries to remove it (line T_{31}).

During the traversing of the tryRem method, if it finds an empty list, then the top pointer of the empty list is recorded in the array $emptyArr$ (line T_{10}-T_{13}). After the traversing, if no candidate node for removal is found (line T_{22}), then the tryRem method traverses all lists again to check whether their top pointers have changed (line T_{23}-T_{26}). If not, the tryRem method returns empty (line T_{27}), and then the dequeue method returns empty. Otherwise, the tryRem method returns false (line T_{25}), and then the dequeue method restarts. If all top pointers have not changed, all lists must have been empty between the first (line T_7) and second (line T_{23}) traversal (because the top pointers are annotated with ABA-counters).

Theorem 3 states that the TS queue is linearizable, and the following lemma is used in the proof of Theorem 3.

Lemma 2. For a non-empty dequeue operation, deq, let S_1 be the set of the nodes which are still in the lists while the final removing action T_{31} of deq is executed; let S_2 be the set of the enqueue operations which insert the nodes of S_1. $Match(deq)$ is minimal w.r.t. the happened-before order in the set S_2.

Proof. The final candidate node of deq is inserted by $Match(deq)$. During the final traversing of deq, there are two kinds of lists: the empty lists---are empty when deq visits them, and the non-empty lists---are not empty when deq visits them.

1. For a non-empty list, if the timestamp of the oldest node of the list is not $\top$ when deq visits it, then the enqueue operation inserting the oldest node does not precede $Match(deq)$.

Proof. The timestamp of the final candidate node of deq is not bigger than the timestamp of the oldest node of the non-empty list (by T_{15}). Thus, the enqueue operation inserting the oldest node does not precede $Match(deq)$.

2. For a non-empty list, if the timestamp of the oldest node of the list is ⊤ when deq visits it, then the enqueue operation inserting the oldest node does not precede $Match(deq)$.

Proof. Let enq_x be the enqueue operation inserting the oldest node. enq_x does not complete the action E_5 (sets the timestamp field of the oldest node) when deq visits it. Thus, the action D_4 (generating timestamp) of deq precedes the action E_5 of enq_x. Since the timestamp of $Match(deq)$ is not bigger than the timestamp of deq (by T_{29}), the action D_4 of deq does not precede the action E_4 (generating timestamp) of $Match(deq)$ $Match(deq)$. Thus, the action E_5 of enq_x does not precede the action E_4 of $Match(deq)$. Thus, enq_x does not precede $Match(deq)$.

3. For the empty lists, if some nodes are inserted into the empty lists after deq visits them, then the enqueue operations inserting the nodes do not precede $Match(deq)$.

Proof. Assume that an enqueue operation enq_x inserts a node into an empty list after deq visits the empty list. When deq visits the empty list, enq_x does not complete its inserting action. The timestamp of enq_x is generated by E_4 after the inserting action E_3. Thus, the timestamp of enq_x is bigger than the one of deq. By the statement T_{29}, the timestamp of $Match(deq)$ is not bigger than the one of deq. Thus, the timestamp of $Match(deq)$ is not bigger than the one of enq_x. Thus, the enqueue operation enq_x does not precede $Match(deq)$.

4. Q.E.D.

Proof. By 1 and 2, for any non-empty list, the enqueue operation inserting the oldest node does not precede $Match(deq)$. Thus, the enqueue operations inserting other nodes do not also precede $Match(deq)$. By 3, for any empty list, the enqueue operations inserting the nodes into the empty list do not precede $Match(deq)$.

```
L0 TSQueue{
L1   SPPool [maxThreads] pools;
L2   void Enqueue (int e);
L3   int Dequeue( );
L4 }
L5 Node{
L6   int val;
L7   Timestamp timestamp;
L8   Node next;
L9 }
L10 SPPool{
L11   Node insert;// insert pointer.
L12   Node remove;// remove pointer.
L13   ...// definitions of methods
L14 }
E0 void Enqueue(int e){
E1   Timestamp ts;
E2   SPPool pool=pools[threadID];
E3   Node node=pool.insert(e);
E4   ts=newTimestamp();
E5   node.timestamp=ts;
E6 }
D0 int Dequeue( ){
D1   Timestamp startTS;
D2   Bool success;
D3   while(true){
D4     startTS=newTimestamp( );
D5     (success, e)=tryRem(startTS);
D6     if(success)
D7       break;
D8   }
D9   return e;
D10}
```

```
T0 (bool,Val) tryRem(TimeStamp startTS){
T1   Timestamp minimalTS,CurNodeTS;
T2   SPPool candidateList;
T3   Node[maxThreads] emptyArr;
T4   Node CurNode, CurTop, oldestNode;
T5   oldestNode=NULL;
T6   minimalTS=⊤;
T7   for each (SPPool CurList in pools){
T8     (CurNode, CurTop)=CurList.getOldest();
T9   // recording empty lists (line T10 − T13)
T10    if(CurNode==NULL){
T11      emptyArr[CurList.id]=CurTop;
T12      continue;
T13    }
T14    CurNodeTS=CurNode.timestamp;
T15    if(minimalTS >ts CurNodeTS){
T16        oldestNode=CurNode;
T17        minimalTS=CurNodeTS;
T18        candidateList=CurList;
T19    }
T20  }
T21  // Emptiness check (line T22 − T28)
T22  if(oldestNode==NULL){
T23    for each(SPPool CurList in pools ){
T24      if(CurList.top ≠ emptyArr[CurList.id])
T25          return (false, NULL);
T26    }
T27    return (true, EMPTY);
T28  }
T29  if(minimalTS >ts startTS)
T30    return (false, NULL);
T31  return candidateList.remove(oldestNode);
T32 }
```

Fig. 4. The TS queue

An enqueue operation always inserts a node into a list; a dequeue method either removes a node from a list and returns the value of the node or returns empty; a node is removed at most once. Thus, for a complete history H of the TS queue, there is a safe mapping $Match$ from $Deq(H)$ to $Enq(H)$ and ϵ.

Theorem 3. Every complete history H of the TS queue is linearizable with respect to the standard sequential queue specification.

Proof. For a non-empty dequeue operation, we choose the successful removing node action of the tryRem method (T_{31}) to construct the linearization of dequeue operations; for an empty dequeue operation, we choose T_{22} (at the time point, all lists are empty). Assume $deq_1, deq_2, \ldots, deq_n$ is a linearization of the dequeue operations constructed in terms of these atomic actions. Based on the linearization and the safe mapping $Match$, we prove that the TS queue satisfies the two conditions of Theorem 1.

1. $1 \leq \forall i \leq n$. if $Match(deq_i) \neq \epsilon$, let the set $PPA = Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}$, then $\forall enq_x \in PPA.\ enq_x \prec_H Match(deq_i)$ and $\mathrm{i} \leq \forall \mathrm{j} \leq \mathrm{n}.\ deq_j \prec_H Match(deq_i)$.

Proof. Obviously, $Match(deq_i)$ completes its inserting action E_3 before the removing action T_{31} of deq_i. If enq_x does not complete its inserting action before the removing action of deq_i, then $enq_x \prec_H Match(deq_i)$. If enq_x completes its inserting action before the removing action of deq_i, then the node inserted by enq_x is not removed before the removing action of deq_i (by $enq_x \in PPA$). By Lemma 2, $enq_x \prec_H Match(deq_i)$. Since the inserting node action of $Match(deq_i)$ precedes the removing node action of deq_i, and the removing node action of deq_i precedes the removing node action of deq_j, $deq_j \prec_H Match(deq_i)$.

2. $1 \leq \forall i \leq n$. If $Match(deq_i) = \epsilon$, let $PPN = \{enq \mid enq \simeq deq_i \wedge enq \in Enq(H) - \{Match(deq_1), \ldots, Match(deq_{i-1})\}\}$, then (1) $\forall enq.\ enq \prec_H deq_i \Rightarrow enq \in \{Match(deq_1), \ldots, Match(deq_{i-1})\}$; (2) $\forall enq \in PPN, x \leq i-1.\ enq \prec_H deq_x$.

Proof. Since deq_i returns **empty**, all lists must have been empty at the time point when the statement T_{22} of deq_i is executed. Thus, for all enqueue operations which precede deq_i, the values inserted by them are removed by the previous dequeue operations (i.e., $deq_x,\ x < i$).

At the above time point, any enqueue operation $enq \in PPN$ has not completed the inserting action (E_3), any previous dequeue operation $deq_x,\ x < i$, has completed the removing action (T_{31}). Thus, $enq \prec_H deq_x$.

6 Related Work

There has been a great deal of work on linearizability verification [13-27]. Mainly, there are four kinds of verification techniques: refinement-based techniques, simulation-based techniques, reduction-based techniques, program-logic-based techniques. An interested reader may refer to the survey article [13]. However, proving linearizability of sophisticated concurrent data structures is still a challenging task.

Much work on proving linearizability is based on different kinds of *simulation proofs* [17-20]. As we explained in Section 1, forward simulation alone is not sufficient in general to verify linearizability. However, Schellhorn et al. prove that backward simulation alone *is* always sufficient. However, backward simulation proofs are difficult to understand intuitively and require considerable expertise.

Bouajjani et al. propose a forward simulation technique for proving linearizability [17]. They have successfully applied the method to prove the HW queue. In fact, for the HW queue, there does not exist a forward simulation to the standard sequential queue. They need to construct a deterministic atomic reference implementation (as an intermediate specification) for the concurrent queue, and the linearizability proof is reduced to showing that the HW queue is forward-simulated by the intermediate specification.

Schellhorn et al. propose a backward simulation technique for proving linearizability [20]. Their proof technique can deal with concurrent data structures where the linearization points are not fixed, but the proofs are conceptually more complex and less amenable to automation.

One related approach to ours is that of Henzinger et al. [10] (called Aspect-oriented proof technique), which reduces the task of proving linearizability of concurrent queues to establishing four basic properties, each of which can be proved independently. For the non-empty dequeue operations, their proof technique needs to verify the following key property: if for two non-overlapping enqueue operations enq_1 and enq_2, enq_1 precedes enq_2, then the value inserted by enq_2 cannot be removed earlier than the one inserted by enq_1 (i.e., deq_2 cannot precede deq_1 where deq_2/deq_1 removes the value inserted by enq_2/enq_1). For the empty dequeue operations, they propose a primitive verification condition, which requires that there exists a subset of enqueue operations containing the enqueue operations which precede the empty dequeue operation such that the empty dequeue

operation does not precede any of their matching dequeue operations and an enqueue operation which precedes any of their matching dequeues operation also belongs to the subset. In comparison with the Aspect-Oriented proof technique, our verification conditions make use of the removing actions of dequeue operations, intuitively characterize the "FIFO" semantics of concurrent queues and can be transformed into the following state-based invariant: when the removing action of a dequeue operation logically or physically removes a value, the value is the oldest value in the current queue.

Khyzha et al. propose a verification technique based on partial orders [14] that is related to our work. The key idea of their technique is to incrementally construct an abstract history—a partially ordered history of operations; the linearizability proof is reduced to establish a simulation between its execution and a growing abstract history. They formalize the technique as a program logic based on rely-guarantee reasoning, have applied it to verify the HW queue, the TS queue and the optimistic set [27]. Their proof technique is generic and can handle concurrent data structures with non-fixed linearization points. However, the proof technique relies on program logic and needs to construct a partially ordered history.

7 Conclusion

We present a simple and complete proof technique for verifying linearizability of concurrent queues. Our proof technique reduces the problem of proving linearizability of concurrent queues to establishing a set of conditions based on the happened-before orders of operations. The verification conditions can be easily verified, designers can easily and quickly learn to use the proof technique. We have successfully applied the proof technique to several concurrent queues: the TS queue and the LCR queue, etc. However, our proof technique is limited to concurrent data queues. We believe that our proof technique can be extended to prove the concurrent data structures which have the ordering requirements when their elements are removed, such as priority queues. We plan to pursue this direction in future work.

8 Acknowledgement

This work is supported by National Natural Science Foundation of China [No. 62341204] and Science and Technology Research Project of Jiangxi Province Educational Department [No. GJJ2203609].

References

[1] M.P. Herlihy, J.M. Wing, Linearizability: A correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12(3)(1990) 463–492.
[2] M.M. Michael, M.L. Scott, Simple, fast, and practical non-blocking and blocking concurrent queue algorithms, in: Proc. 1996 the fifteenth annual ACM symposium on Principles of distributed computing, 1996.
[3] M. Hoffman, O. Shalev, N. Shavit, The baskets queue, in: Proc. 2007 International Conference on Principles of Distributed Systems, 2007.
[4] A. Morrison, Y. Afek, Fast concurrent queues for x86 processors, in: Proc. 2013 Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming, 2013.
[5] A. Haas, Fast concurrent data structures through timestamping, [dissertation] Salzburg, Austria: University of Salzburg, 2015.
[6] E. Ladan-Mozes, N. Shavit, An optimistic approach to lock-free fifo queues, in: Proc. 2004 International Symposium on Distributed Computing, 2004.
[7] M. Herlihy, N. Shavit, V. Luchangco, M. Spear, The art of multiprocessor programming, Newnes, Burlington, 2020.
[8] H. Liang, X. Feng, Modular verification of linearizability with non-fixed linearization points, in: Proc. 2013 Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2013.
[9] T.L. Wen, L. Song, Z. You, Proving linearizability using reduction, The Computer Journal 62(9)(2019) 1342–1364.
[10] T.A. Henzinger, A. Sezgin, V. Vafeiadis, Aspect-oriented linearizability proofs, in: Proc. 2013 International Conference on Concurrency Theory, 2013.
[11] L. Lamport, How to write a 21st century proof, Journal of Fixed Point Theory and Applications 11(1)(2012) 43–63.
[12] D. Dechev, The aba problem in multicore data structures with collaborating operations, in: Proc. 2011 7th International Conference on Collaborative Computing, 2011.

[13] B. Dongol, J. Derrick, Verifying linearisability: A comparative survey, ACM Computing Surveys 48(2)(2015) 1–43.
[14] A. Khyzha, M. Dodds, A. Gotsman, M. Parkinson, Proving linearizability using partial orders, in: Proc. 2017 European Symposium on Programming, 2017.
[15] V. Singh, I. Neamtiu, R. Gupta, Proving concurrent data structures linearizable, in: Proc. 2016 IEEE 27th International Symposium on Software Reliability Engineering, 2016.
[16] S. Krishna, N. Patel, D. Shasha, T. Wies, Verifying concurrent search structure templates, in: Proc. 2020 Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation, 2020.
[17] A. Bouajjani, M. Emmi, C. Enea, S.O. Mutluergil, Proving linearizability using forward simulations, in: Proc. 2017 International Conference on Computer Aided Verification, 2017.
[18] P. Jayanti, S. Jayanti, U. Yavuz, L. Hernandez, A Universal, Sound, and Complete Forward Reasoning Technique for Machine-Verified Proofs of Linearizability, in: Proc. 2024 Proceedings of the ACM on Programming Languages, 2024.
[19] Q. Jia, Y. Lv, P. Wu, B. Zhan, J. Hao, H. Ye, C. Wang, Verilin: A linearizability checker for large-scale concurrent objects, in: Proc. 2023 International Symposium on Theoretical Aspects of Software Engineering, 2023.
[20] G. Schellhorn, H. Wehrheim, J. Derrick, How to prove algorithms linearizable, in: Proc. 2012 Computer Aided Verification: 24th International Conference, 2012.
[21] S.F. Vindum, D. Frumin, L. Birkedal, Mechanized verification of a fine-grained concurrent queue from meta's folly library, in: Proc. 2022 Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2022.
[22] Y.M. Feldman, A. Khyzha, C. Enea, A. Morrison, A. Nanevski, N. Rinetzky, S. Shoham, Proving highly-concurrent traversals correct, in: Proceedings of the ACM on Programming Languages 4 (OOPSLA), 2020.
[23] B.K. Ozkan, R. Majumdar, F. Niksic, Checking linearizability using hitting families, in: Proc. 2019 Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming, 2019.
[24] C. Wang, C. Enea, S.O. Mutluergil, G. Petri, Replication-aware linearizability, in: Proc. 2019 Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, 2019.
[25] G. Sela, E. Petrank, Concurrent size, in: Proc. 2022 Proceedings of the ACM on Programming Languages, 2022.
[26] G. Smith, Model checking simulation rules for linearizability, in: Proc. 2016 International Conference on Software Engineering and Formal Methods, 2016.
[27] P.W. O'Hearn, N. Rinetzky, M.T. Vechev, E. Yahav, G. Yorsh, Verifying linearizability with hindsight, in: Proc. 2010 Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing, 2010.

Appendix A: Algorithm 1

By using Algorithm 1, L' can be inserted into a proper position such that the new sequence preserves the partial order. The algorithm is also used in the proof of Theorem 1.

Algorithm 1: preserving the partial order $\prec$

if $L_n \prec L'$ **then**
    L' is inserted to the right of L_n;
...
else if $L_i \prec L'$ **then**
    L' is inserted between L_i and L_{i+1};
...
else if $L_1 \prec L'$ **then**
    L' is inserted between L_1 and L_2;
else
    L' is inserted into the left of L_1;