A Simple Way to Verify Linearizability of Concurrent Stacks

TANGLIU WEN¹ and LAN SONG²

¹Gannan University of Science and Technology& Jiangxi University of Science and Technology, Ganzhou, China
²State Key Lab of Software Engineering&School of Computer, Wuhan University, Wuhan 430072, China
Email: wqtlglk@163.com

Linearizability is a commonly accepted correctness criterion for concurrent data structures. However, verifying linearizability of highly concurrent data structures is still a challenging task. In this paper, we present a simple and complete proof technique for verifying linearizability of concurrent stacks. Our proof technique reduces linearizability of concurrent stacks to establishing a set of conditions. The conditions are based on happened-before order of operations, intuitively express the “LIFO” semantics and can be proved by simple arguments. Verifiers of concurrent data structures can easily and quickly learn to use the proof technique. We have successfully applied the method to several challenging concurrent stacks: the TS stack, the HSY stack, and the FA stack.

Keywords: Concurrent Data Structures; Reduction; Linearizability; Verification

Received 00 January 2009; revised 00 Month 2009

1. INTRODUCTION

Linearizability[1-3] is a commonly accepted correctness criterion for concurrent data structures. Intuitively, linearizability requires that (1) each concurrent execution of a concurrent data structure is equivalent to a legal sequential execution and (2) the sequential execution preserves the order of non-overlapping operations (called the happened-before order, for two operations \( op, op' \) if \( op \prec op' \), if \( op \) returns before \( op' \) begins to execute). To achieve high performance, concurrent data structures often employ sophisticated fine-grained synchronization techniques[4-8]. This makes it more difficult to verify the linearizability of concurrent data structures.

Henzinger et al.[9] propose a simple method to the linearizability verification of concurrent queues, which reduces the problem of verifying linearizability of concurrent queues to checking four simple properties. The key one of these properties is based on the happened-before order of operations: If for two non-overlapping enqueue operations \( enq_1 \) and \( enq_2 \) in a concurrent execution, \( enq_1 \) precedes \( enq_2 \), then the value inserted by \( enq_2 \) cannot be removed earlier than the one inserted by \( enq_1 \), i.e., \( \text{deq}_2 \) cannot precede \( \text{deq}_1 \), where \( \text{deq}_2/\text{deq}_1 \) removed the value inserted by \( enq_2/enq_1 \). Their method is not be necessary to find the entire linearization order to show that an execution is linearizable. For instance, for two non-overlapping push operations, either of the two values pushed by the two push operations is possible to be popped first.

Intuitively, a pop operation must return the value pushed by the latest operation among the push operations whose effects can be observed by the pop operation. If a push operation precedes a pop operation in a concurrent execution, then the pop operation can observe the effect of the push operation. However, if a pop operation is interleaved with a push operation, the pop operation may observe the effect of the push operation or not. For example, consider the following execution of a stack shown in Figure 1. The pop operation can return either \( x \) or \( y \). This depends on whether the pop operation observes the effect of the operation \( \text{push}(y) \). Whichever the pop operation returns, the execution is always linearizable.

![Figure 1. which value does the pop operation return?](image-url)

If a push operation is interleaved with a pop operation
and the pop operation returns the value pushed by the push operation, then we call them an elimination pair.

In this paper, we prove that the elimination pairs do not violate linearizability of concurrent stacks, i.e., for a history $H$ of a concurrent stack, let $H'$ be a subsequence of $H$ obtained by deleting the elimination pairs of $H$; if $H'$ is linearizable, then $H$ is also linearizable. This enables verifiers to ignore the elimination pairs and focus on the “common” operations when they verify the linearization of concurrent stacks.

Any “common” pop operation pops the value pushed by the push operation which precedes the pop operation. Thus these pop operations do not exist the above nondeterministic problem. In this paper, we present a stack theorem for verifying concurrent stacks, this theorem presents a set of conditions based on the happened-before order which are sufficient and necessary to establish linearization of the “common” operations. Because of avoiding the above nondeterminism, the theorem intuitively characterizes the “Last in First Out” (“LIFO”) semantics. Informally, the stack theorem says: a concurrent execution of a stack is linearizable iff there exists a linearization of the pop operations, such that the pop operations always remove the latest values in the stack in the linear order of pop operations.

Although our proof technique requires a linear order of pop operations, we are surprised to discover (1) the above linearization of pop operations is probably difference from the final linearization of the pops operations which is extracted from the linearization of the execution. (2) for all concurrent stacks we have met, the atomic write actions of pop methods which logically or physically remove the values of stacks can be chosen as their linearization points to construct the initial linearization. Thus it is not a difficult task to construct the initial linearization of pop operations.

The conditions stated in the stack theorem can be easily verified by the properties based on happened-before order of operations, verifiers do not need to know other proof techniques, and can easily and quickly learn to use the proof technique. We have successfully applied the method to several challenging concurrent stacks: the TS stack, the HSY stack, the FA stack, etc.

## 2. LINEARIZABILITY

A history of a concurrent data structure is a sequence of invocation and response events[1,9]. We refer to a method call as an operation. An invocation event $(t, \text{inv}(m, v), o)$ represents a method $m$ with an argument value $v$ which is performed by a thread $t$ and is identified by an operation identifier $o$. A response event $(t, \text{ret}(v), o)$ represents an operation $o$ with a return value $v$. An invocation event matches a response event if they are associated with the same operation. A history is sequential if every invocation event, except possibly the last, is immediately followed by its matching response event. A sequential history of a concurrent data structure is legal if it satisfies the sequential specification of the concurrent data structure.

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 $\text{Compl}(H)$ be the set of all completions of the history $H$. For a history $h$ and a thread $t$, let $h[t]$ denote the maximal subsequence of $h$ consisting of the events performed by the thread $t$. Let $\prec_H$ denote the happened-before order of operations in the history $H$; for any two operations $o$ and $o'$ of $H$, $o \prec_H o'$ if the response event of $o$ precedes the invocation event of $o'$. We say the operation $o$ precedes the operation $o'$ in the history $H$ if $o \prec_H o'$. For any two operations $o$ and $o'$ of $H$, $o$ is interleaved with $o'$, i.e., $o \not\prec_H o'$ and $o' \not\prec_H o$, denoted by $o \approx_H o'$. We omit the subscripts when the histories are clear from the context.

A history $H$ is linearizable with respect to a sequential specification[1,11] if there exists $C \in \text{Compl}(H)$ and a legal sequential history $S$ such that $\forall t. C[t] = S[t]$; if for any two operations $o_1, o_2$, $o_1 \prec_C o_2$, then $o_1 \prec_S o_2$. $S$ is called a linearization of $H$. 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 “LIFO” stack is used to characterize the sequential specification of a concurrent stack. For a linearizable stack with respect to the standard specification, we sometimes omit the standard specification for simplicity.

In this paper, we only consider complete histories. As Henzinger et al. [9] have shown, a purely-blocking data structure is linearizable iff every complete history of the concurrent data structure is linearizable. The notion of purely blocking is a very weak liveness property, and most of the concurrent data structures satisfy the liveness property. In particular, all concurrent data structures verified in this paper are purely blocking.

## 3. PARTIALLY ORDERED SETS

A strict partial order on a set is an irreflexive, antisymmetric, and transitive relation. Obviously, the happened-before order of operations is a strict partial order on the set of operations. An element $x$ in the set $S$ with a strict partial order $\prec$ is called maximal if $\forall y \in A. x \not\prec y$, and $x$ is called minimal if $\forall y \in A. y \not\prec x$. We say $y$ is bigger than $x$ with respect to a strict partial order $\prec$ if $x \prec y$. Let $\prec_1$ and $\prec_2$ be two partial orders on a set $S$. The partial order $\prec_1$ is called an extension of partial order $\prec_2$ if, whenever $a \prec_1 b$, then $a \prec_2 b$. 
Proposition 3.1. Let \( \prec \) be a strict partial order on the set \( S \), let \( L_1, L_2, \ldots, L_n \) be a sequence which preserves the partial order \( \prec \) (i.e., \( \forall x, y. 1 \leq x \leq n, 1 \leq y \leq n, x < y \Rightarrow L_y \not\prec L_x \)). Then for any element \( L' \in S \), \( L' \) can be inserted into the sequence such that it still preserves the partial order \( \prec \).

Obviously, the following algorithm does the job and it also is used in the proofs of Lemma 4.1 and Theorem 5.1.

Algorithm 1: a linear extension

\[
\begin{align*}
\text{if} & \quad L_n \prec L' \text{ then } \\
& \quad L' \text{ is inserted to the right of } L_n; \\
\text{else if} & \quad L_i \prec L' \text{ then } \\
& \quad L' \text{ is inserted between } L_i \text{ and } L_{i+1}; \\
\text{else if} & \quad L_1 \prec L' \text{ then } \\
& \quad L' \text{ is inserted between } L_1 \text{ and } L_2; \\
\text{else } & \quad L' \text{ is inserted into the left of } L_1;
\end{align*}
\]

4. ELIMINATION MECHANISM

Elimination is an essential optimization technique for concurrent stacks\([12-14]\). For example, both the HSY stack and the TS stack apply the parallelization optimization mechanism. The elimination mechanism is based on the fact that if a push operation followed by a pop operation is performed on a stack, the stack’s state remains unchanged. In a concurrent execution, if a push operation is interleaved with a pop operation and the pop operation returns the value inserted by the push operation, then they are called an elimination pair. In order to reduce frequency of shared-data accesses and increase the degree of parallelism of stack, the elimination mechanism allows the elimination pairs to exchange their values without accessing the shared stack structure.

We now prove that the elimination mechanism does not violate linearizability of concurrent stacks. In other words, we can ignore the elimination pairs when proving linearizability of concurrent stacks.

Lemma 4.1. For a history \( H \) of a concurrent stack, let \( H' \) be a subsequence of \( H \) obtained by deleting an elimination pair of \( H \). If \( H' \) is linearizable with respect to the standard “LIFO” specification, then \( H \) is also linearizable with respect to the specification.

Proof. Let \( \text{push}(x) \) denote the push operation with an input parameter \( x \). Let \( \text{pop}(x) \) denote the pop operation with a return value \( x \). Assume that \( \text{push}(x) \) and \( \text{pop}(x) \) are an elimination pair of \( H \). \( H' \) is obtained from \( H \) by deleting the elimination pair. \( H' \) is a linearization of \( H' \). Consider two cases: In the first case, \( \text{push}(x) \) begins to execute earlier than \( \text{pop}(x) \), as shown in Figure 2. In the second case, \( \text{pop}(x) \) begins to execute earlier than \( \text{push}(x) \). In the following, we prove that the lemma holds in the first case. The proof for the second case is similar.

In the first case, there are the following properties:

Property 1. \( \forall \text{op}', \text{pop}(x) \prec_H \text{op}' \Rightarrow \text{push}(x) \prec_H \text{op}' \).

Property 2. \( \forall \text{op}', \text{op}' \cong_H \text{pop}(x) \Rightarrow \text{op}' \not\prec_H \text{push}(x) \).

Property 3. \( \forall \text{op}', \text{op}' \prec_H \text{pop}(x) \Rightarrow \text{push}(x) \not\prec_H \text{op}' \).

\[
\begin{align*}
T_1 \xrightarrow{\text{push}(x)} & \xrightarrow{\text{pop}(x)} \\
T_2 \xrightarrow{\text{op}} & \xrightarrow{\text{pop}(x)}
\end{align*}
\]

Figure 2. \( \text{push}(x) \) begins to execute earlier than \( \text{pop}(x) \)

Using Algorithm 1, we insert \( \text{pop}(x) \) into \( H'_1 \). If there exist operations which precede \( \text{pop}(x) \) in \( H' \), then the left operation of \( \text{pop}(x) \) precedes \( \text{pop}(x) \). Assume the left operation is \( \text{op} \).

\[
\ldots, \text{op}, \text{pop}(x), \ldots
\]

Since \( \text{op} \prec_H \text{pop}(x) \), the operation \( \text{op} \) must either be interleaved with \( \text{push}(x) \), or precede \( \text{push}(x) \). The former is shown in Figure 3. In either case, there is the following property:

Property 4. \( \forall \text{op}', \text{op}' \prec \text{op} \Rightarrow \text{push}(x) \not\prec_H \text{op}' \).

Property 5. \( \forall \text{op}', \text{op}' \cong \text{op} \Rightarrow \text{push}(x) \not\prec_H \text{op}' \).

\[
\begin{align*}
T_1 \xrightarrow{\text{push}(x)} & \xrightarrow{\text{pop}(x)} \\
T_2 \xrightarrow{\text{op}} & \xrightarrow{\text{pop}(x)}
\end{align*}
\]

Figure 3. the operation \( \text{op} \) is interleaved with \( \text{push}(x) \)

After inserting \( \text{pop}(x) \), we insert \( \text{push}(x) \) between \( \text{op} \) and \( \text{pop}(x) \), as shown below.

\[
\ldots, \text{op}, \text{push}(x), \text{pop}(x), \ldots
\]

By Property 1 and Property 2, any operation on the right of \( \text{pop}(x) \) does not precede \( \text{push}(x) \). By Property 3, \( \text{push}(x) \) does not precede \( \text{op} \). By Property 4 and Property 5, \( \text{push}(x) \) does not precede any operation on the left of \( \text{op} \).

Thus, after inserting \( \text{push}(x) \), the new sequence preserves the happened-before order. Obviously, the new sequence satisfies the “LIFO” semantics. Thus, the new sequence is a linearization of \( H \).
..., op, push(x), pop(x),...

If there is no any operation which precedes pop(x) in H', by Algorithm 1, we insert pop(x) in the front of H'. Then we insert push(x) into the left of pop(x), as shown below.

\[ \text{push}(x), \text{pop}(x) \in H'_1 \]

By Property 1 and Property 2, the final sequence preserves the happened-before order. Obviously, the sequence is a linearization of H. □

By the above lemma, we can get the following theorem.

**Theorem 4.1.** For a history H of a concurrent stack, let H' be a subsequence of H obtained by deleting all elimination pairs of H. If H' is linearizable with respect to the standard “LIFO” specification, then H is also linearizable with respect to the specification.

### 5. CONDITIONS FOR STACK LINEARIZABILITY

For a history H of a stack, let \( \text{PUSH}(H) \) and \( \text{POP}(H) \) denote the sets of all push and pop operations in h, respectively. We map each push operation of the history to the standard “LIFO” specification iff there exists a linearization sequence.

#### Definition 5.1.

For a complete history H, a mapping \( \text{Match} \) from \( \text{POP}(H) \) to \( \text{PUSH}(H) \) and \( \epsilon \) is safe if:

1. If \( \forall \text{pop} \in \text{POP}(H), \text{Match}(\text{pop}) \neq \epsilon \), then the value returned by the operation pop is inserted by the operation \( \text{Match}(\text{pop}) \).
2. If \( \forall \text{pop} \in \text{POP}(H), \text{Match}(\text{pop}) = \epsilon \), then the pop operation returns empty.
3. If \( \forall \text{pop}, \text{pop}' \in \text{POP}(H), \text{Match}(\text{pop}) \neq \epsilon \land \text{Match}(\text{pop}') \neq \epsilon \), then \( \text{Match}(\text{pop}) \neq \text{Match}(\text{pop}') \).

The conditions stated in the following theorem characterize the “LIFO” semantics of concurrent stacks.

**Theorem 5.1.** Let H be a subsequence of a concurrent stack’s history obtained by deleting elimination pairs of the history. H is linearizable with respect to the standard “LIFO” specification iff there exists a linearization \( \text{Pop}_1, \text{Pop}_2, \ldots, \text{Pop}_n \) of all pop operations in \( H \) (i.e., \( \text{POP}(H) = \{\text{Pop}_1, \text{Pop}_2, \ldots, \text{Pop}_n\} \land x < y \Rightarrow \text{Pop}_y \prec_H \text{Pop}_x \)) and a safety mapping \( \text{Match} \) such that:

1. If \( \text{Match}(\text{Pop}_i) = \text{Push}_i \neq \epsilon \), let \( \text{PUSH}' = \{\text{push} \mid \text{push} \in (\text{PUSH}(H) - \{\text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1})\}) \land \text{push} \prec_H \text{Pop}_i\} \), then \( \text{Push}_i \in \text{PUSH}' \) and \( \forall \text{push} \in \text{PUSH}', \text{Push}_i \neq \epsilon \).
2. If \( \text{Match}(\text{Pop}_i) = \epsilon \), then (a) \( \forall \text{push} \in \text{PUSH}(H) \land \text{push} \prec_H \text{Pop}_i \Rightarrow \text{push} \in \{\text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1})\} \); (b) let \( \text{PPN} = \{\text{push} \mid \text{push} \simeq_H \text{Pop}_i \land \text{push} \in (\text{PUSH}(H) - \{\text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1})\})\} \), \( \forall \text{push} \in \text{PPN} \Rightarrow \text{push} \notin \text{Match}(\text{Pop}_i) \).

Informally, the first condition requires that each non-empty pop operation always pops the value pushed by the latest push operation among the push operations which precede the pop operation and are not mapped to its previous pop operations (with respect to the linear order of pop operations). In other words, the pop operations always remove the latest values in the stack in the linear order of pop operations. The second condition requires that if a pop operation returns an empty value, then for any push operation which precedes the pop operation, the values pushed by them are removed by the previous pop operations of the pop operation; for any push operation which is interleaved with the pop operation and is not mapped to the previous pop operation of the pop operation, the push operation does not preceed the push operations mapped to the previous pop operations of the pop operation.

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

1. H is linearizable when \( H \) does not contain the pop operations returning empty.

The proof is done in two stages. Firstly, we construct a linearization of push operations. Secondly, we insert \( \text{Pop}_1, \text{Pop}_2, \ldots, \text{Pop}_n \), one after another, into the linearization sequence.

**Step 1:** Construct a linearization of push operations.

Assume the number of push operations in \( \text{PUSH}(H) \) is \( m \). We construct a linearization of push operations by the following rules: First, choose the first element from the maximal push operations with respect to the partial order \( \prec_H \) such that its matching pop operation is the smallest with respect to the linear order of pop operations. In other words, the pop operations always remove the latest values in the stack in the linear order of pop operations.

"Step 2:" Linearize the pop operations.

For any pop operation \( \text{Pop}_i \) which precedes the pop operation \( \text{Pop}_j \), we map it to the position where \( \text{Pop}_j \) is the smallest with respect to the linear order of pop operations. If \( \text{Pop}_i \) is the first step, choose an element \( \text{Push}_i \), then we insert \( \text{Push}_i \) into the left of \( \text{Pop}_i \), as shown below.

\[ \text{push(x)}, \text{pop(x)} \in H'_1 \]

By Property 1 and Property 2, the final sequence preserves the happened-before order. Obviously, the sequence is a linearization of H.
rest of \( PUSH(H) \); and insert it before \( Push_m \) (i.e., \( Push_{m-1}, Push_m \)). Third, continue to construct the sequence in the same way until all push operations of \( PUSH(H) \) are chosen.

By construction, the final sequence is \( Push_1, \ldots, Push_m \). The sequence is a linearization of all push operations, and have the following properties:

**Property 1.**
For any push operation \( Push_x \), \( Push_x \) is a maximal element (w.r.t. \( <_H \)) among the left elements of \( Push_z \), and if \( Push_x \) has a matching pop operation, its matching pop operation is the smallest (w.r.t. \( <_L \)) among the matching pop operations of the maximal push operations.

**Step 2:** We insert \( Pop_1, Pop_2, \ldots, Pop_n \), one after another, into the sequence \( Push_1, \ldots, Push_m \).

**Step 2.1:** Using Algorithm 1, we insert \( Pop_1 \) into the linearization sequence of push operations. In terms of the property of Algorithm 1, we can get that (1) the left push operation of \( Pop_1 \) precedes \( Pop_1 \), (2) after the insertion operation, the new sequence preserves the order \( <_H \). Let \( Push_x \) denote the left element of \( Pop_1 \). According to Property 1 and the first condition of the theorem, \( Match(Pop_1) = Push_x \).

**Step 2.2:** Similar to \( Pop_1 \), for any \( Pop_i, 2 < i \leq n \), insert it into the new sequence.

**Step 2.2.1:** Using Algorithm 1 and ignoring the previous pop operations, we found the first push operation which precede \( Pop_i \). Let \( Push_y \) denote the push operation. If \( Push_y \) is not followed by pop operations, then insert \( Pop_i \) into the right of \( Push_y \). If \( Push_y \) is followed by pop operations, then insert \( Pop_i \) into the end of the pop operations.

After the above inserting operation, if \( Pop_i \) is not between any previous pop operation and its matching push operation, then there exists the following property:

**Property 2.**
(1) the new sequence satisfies the “FILO” semantic (we can get this by Property 1 and the first condition of the theorem), and (2) the new sequence preserves the \( <_H \) order. The reasons for (2) are as follows: According to Algorithm 1, \( Pop_i \) and any push operation in the sequence do not violate the happened-before order. According to the order of inserting pop operations, \( Pop_i \) and any pop operation ahead it do not violate the happened-before order. Since the new sequence satisfies the “FILO” semantics, for the pop operations behind \( Pop_i \), (if any), their responding push operations are also behind \( Pop_i \). Thus the push operations do not precede \( Pop_i \). Assume that the pop operations behind \( Pop_i \) precede \( Pop_i \), then we can get that \( Pop_i \) precedes their matching push operations (because the matching push operations precede the pop operations behind \( Pop_i \)). This contradicts the above fact. Thus \( Pop_i \) and any pop operation behind it do not violate the happened-before order.

**Step 2.2.2:** If \( Pop_i \) is between by some previous push operations and their matching push operations, (assume that \( Pop_z \) is the last one among the operations) then move \( Pop_i \) to the right of \( Pop_z \), as follows.

\[
\text{seq} : \ldots, Push_z, \ldots, Push_y, \ldots, Pop_i, \ldots, Pop_z, \ldots
\]

\[
\text{seq'} : \ldots, Push_z, \ldots, Push(y), \ldots, Pop_z, Pop_i, \ldots
\]

\( \text{seq} \) and \( \text{seq'} \) are the sequences before and after the move transforming, respectively. Before the move transforming, there exists the following property:

**Property 3:**
In the \( \text{seq} \), \( Pop_i \) does not precede the push operations between \( Pop_1 \) and \( Pop_i \). The reasons for this are as follows: Before inserting \( Pop_i \), the sequence satisfies the “FILO” semantics. For the push operations between \( Pop_1 \) and \( Pop_i \), their matching pop operations are in the \( \text{seq} \). If there exists a push operation \( push \) in the push operations such that \( Pop_i \) \( \prec_H \) \( push \), then \( Pop_i \) \( \prec_H \) \( Match(push) \). This contradicts the linearization order of the pop operations.

**Property 4:**
After the move transforming, the new sequence \( \text{seq'} \) preserves the happened-before order and satisfies the “FILO” semantics. The reasons for this are as follows: Before inserting \( Pop_i \), the sequence satisfies the “FILO” semantics. According to Property 1 and the first condition of the theorem, the new sequence satisfies the “FILO” semantics. By Property 3 and Algorithm 1, in the \( \text{seq'} \), \( Pop_i \) and any push operation do not violate the happened-before order. Similar to the proof in the above case, we can get that in the \( \text{seq'} \), \( Pop_i \) and any pop operation do not violate the happened-before order.

2. \( H \) is also linearizable when \( H \) contains the pop operations returning empty.

We construct the linearization of \( H \) by the following process:

Generally, if \( Match(Pop_i) = \epsilon \), let \( A \) denote the linearization of \( Pop_1, \ldots, Pop_{i-1} \) and their matching push operations (constructing by the above method), let \( B \) denote the linearization of the other operations of \( H \). Let \( H' = A^\wedge(Pop_i)^\wedge B \).

Obviously, any two push operations do not violate the happened-before order in \( H' \) (i.e. in \( H \), for any two push operations \( op_1 \) and \( op_2 \), if \( op_1 \prec_H \) \( op_2 \) then in \( H' \) \( op_1 \prec_H \) \( op_2 \)).

In the following, we show that in \( H' \) (1) any two push operations do not violate the happened-before order, and (2) any pop operation and any push operation do not violate the happened-before order.

Let \( Push_x \) and \( Push_z \) be a push operation and a pop operation in \( A \), respectively. Let \( Push_y \) be a push operation and a pop operation in \( B \), respectively.

Since \( Push_x \prec_H Match(Push_x) \), \( Pop_i \not\prec_H Match(Push_x) \) and \( Pop_i \not\prec_H Match(Push_x) \), we can get \( Pop_y \not\prec_H Push_x \).

By the first part of the second condition, we can get \( Push_y \not\prec_H Pop_i \). Consider the following cases. (1) If \( Pop_i \prec_H Push_y \), then obviously, \( Push_y \not\prec_H Push_x \) and \( Push_y \not\prec_H Pop_i \). (2) If \( Push_y \prec_H Pop_i \), then by
the second part of the second condition, we can get $\text{Push}_y \not\prec_H \text{Push}_x$ and $\text{Push}_y \not\prec_H \text{Pop}_x$.

$(\Longleftrightarrow)$ Since $H$ is linearizable, there exists a safety mapping from $\text{POP}(H)$ to $\text{PUSH}(H)$ and $\epsilon$. We assume that $H'$ is a linearization of $H$. Let $\text{Pop}_1, \text{Pop}_2, \ldots, \text{Pop}_n$ be the maximal subsequence of $H'$ consisting of pop operations. Obviously, it is a linearization of the pop operations of $H$. It is not difficult to show that they satisfy the two conditions of Theorem 5.1.

Note that by Step 2.2.2 in the above proof, the initial linear order of pop operations in the Theorem 5.1 is probably different from the final linear order of the pops operations which is extracted from the linearization of the execution constructed in the above proof. In section X, we will illustrate this point using an example. Next, we show that the initial linear order of pop operations can be easily constructed.

6. HOW TO CONSTRUCTING THE LINEARIZATIONS OF THE POP OPERATIONS

Our method requires an initial linearization of pop operations which satisfies the two conditions of Theorem 5.1. A challenge for applying our method is how to construct such linearization of pop operations. Fortunately, for all concurrent stacks we have verified, the initial linearization can be constructed in terms of the atomic write actions of the pop operations which logically or physically remove the values in the stacks, i.e., the removing actions are viewed as “linearization points”, in the initial linear orders, the pop operations is in the order of the removing actions in concurrent executions.

The physical removing action in a pop method physically removes a value in the stack and the pop method returns the value finally. The logical removing action in a pop method only fixes a value in the stack, the pop method returns the value finally. After the logical removing action, other pop operations cannot logically or physically remove the value. For instance, the statement $\text{CAS}(\&c \rightarrow \text{pop}, \bot_{\text{pop}}, \top_{\text{pop}})$ in the FA stack[], is a logical removing action, the atomic method remove of the TS stack, is a physical removing action. Obviously, the initial linearization can be easily constructed in terms of the fixed “linearization points”.

Then, an important question is, for any stack, whether the (logical or physical) removing action of its pop method can be chosen as a linearization point to construct the initial linearization? Obviously, when the linearization constructed in terms of the removing actions of the pop operations is ineffective for establishing the first condition of Theorem 5, they cannot be chosen as such linearization points.

For simplicity, we consider the executions containing only two pop operations where their removing actions cannot be chosen as such linearization points. Two basic example executions are shown in Figure 4 and Figure 5.

![Figure 4](image-url)

In this figure, the black circles of the pop operations stand for the logical or physical removing actions. In Figure 4, $\text{pop}(y)$ begins to execution before the removing action of $\text{pop}(x)$. The only linearization of the execution is $\text{push}(x), \text{push}(y), \text{pop}(y), \text{pop}(x)$. The linearization of the two pop operations constructed in terms of the two removing actions is $\text{pop}(x), \text{pop}(y)$. Under the linearization of the two pop operations, the first condition of Theorem 5 is not established. Thus, the two removing actions cannot be chosen as linearization points. If there is no the pop operation of Thread 3, then the pop operation of Thread 3 must remove the value pushed by $\text{push}(y)$, to make the execution linearizable. Thus the $\text{pop}(y)$’s actions before the $\text{pop}(y)$’s removing action affect the execution of $\text{pop}(x)$, and prevent it observing $\text{push}(y)$’s effect (or the value pushed by $\text{push}(y)$). Such pop algorithms are uncommon. Generally, except for the logical or physical removing actions, the pop’s actions do not prevent the values in the stack from being removed by other pop operations. For all concurrent stacks we have met, their pop methods have such fixed linearization points. For most of concurrent stacks we have verified, the actions before the removing action either read the shared state or access (read or write) the local state, and do not affect the executions of other operations.

In Figure 5, $\text{pop}(y)$ begins to execution after the removing action of $\text{pop}(x)$. $\text{pop}(x)$ removes the value $x$ before $\text{pop}(y)$ begins to execution. Thus, if there is no $\text{pop}(y)$, then $\text{pop}(x)$ also remove $x$. In this case, the execution of the three operations is not linearizable. Such pop algorithms are nonexistent.

7. VERIFYING THE TIME-STAMPED STACK

We illustrate our proof technique on the Time-Stamped Stack (the TS stack, for short)[15]. The linearization verification of the stack is challenging, because both of its push and pop methods have no fixed linearization points(see [15]). Figure 6 shows the pseudo code for the TS stack.
For two consecutive calls to the algorithm, the latter returns a bigger timestamp than the former (2).

This stack maintains an array pools of singly-linked lists, one for each thread. Each node in the list contains a data value (val), a timestamp (timestamp), the next pointer (next). The push operation of a thread only inserts an element into its associated list. The top pointer of the list is annotated with an ABA-counter to avoid the ABA-problem.

The operations on the list are as follows:

- **insert(v)** inserts a node with a value v and a timestamp泰, to the head of the list and returns a reference to the new node.
- **getYoungest** returns a reference to the node with the youngest timestamp (i.e., the head node), or null if the list is empty.
- **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.

The **push** method first inserts an element into its associated list (line E1), then generates a timestamp (line E2) and sets the timestamp field of the new node to the new timestamp (line E3).

The **pop** method first generates a timestamp startTime, attempts to remove an element by calling the method tryRem.

The tryRem method traverses every list, searching for the node with the youngest timestamp to remove (line T2 - T14). If the node has been found, the method tries to remove it (line T20).

Elimination: During the traversing, if the tryRem method finds a node whose timestamp is bigger than the timestamp startTime of the pop operation (line T8), then the tryRem method tries to remove it (line T9). In this case, the node must have been pushed during the current pop operation. Thus, the push operation must have been interleaved with the current pop operation, and can be eliminated.

Emptiness checking: During the traversing, if the tryRem method finds a list that is empty, then its top pointer is recorded in the array emArr. After the traversing, if no candidate node for removal is found then the tryRem method check whether their top pointers have changed (lines T15). If not, the list must have been empty at line T15.

For a complete history H of the TS stack, let Match map POP(H) to PUSH(H) and ϵ. A push operation always inserts a node with a value into a list; A pop method either removes a node from lists and returns the value of the node or return empty; A node is removed at most once; Thus, Match is a safe mapping.

### Figure 5. pop(y) begins to execution after the removing action of pop(x)

![Figure 5. pop(y) begins to execution after the removing action of pop(x)](image)

### Figure 6. the TS stack

![Figure 6. the TS stack](image)
THEOREM 7.1. For any complete history of the TS stack, let \( H \) be a subsequence obtained by deleting elimination pairs of the history. \( H \) is linearizable with respect to the standard “LIFO” specification.

Proof. For a pop operation returning a non-empty value, we choose \( T_{20} \) (a successful removing node action) as its linearization point; for a pop operation returning empty, we choose \( T_{15} \) as its linearization point; We use these linearization points to construct a linearization \( \text{Pop}_1, \text{Pop}_2, \ldots, \text{Pop}_n \) of the pop operations. We show that the TS stack satisfies the conditions of Theorem 5.1.

1. If \( \text{Match}(\text{Pop}_i) = \text{Push}_i \neq \epsilon \), let \( \text{PUSH}' = \{ \text{push} \mid \text{push} \in \text{PUSH}(H) \} \setminus \{ \text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1}) \} \wedge \text{push} \prec_H \text{Pop}_i \}, \) then \( \forall \text{push} \in \text{PUSH}' \). \( \text{Push}_i \prec_H \text{Push}_x \).

Proof. Assume that there exists a push operation \( \text{Push}_x \prec_H \text{Pop}_i \). \( \text{Push}_x \) have inserted a node into its associated list before \( \text{Pop}_i \) traverses these lists. During the traversing the lists, if \( \text{Pop}_i \) does not access the node inserted by \( \text{Push}_x \), then \( \text{Pop}_i \) accesses the head node with a bigger timestamp of the \( \text{Push}_x \)'s associated list. Thus the timestamp of the head node is also bigger than the one of \( \text{Push}_i \). In this case, we can get \( \text{Push}_i \prec_{ts} \text{Push}_x \). Because \( \text{Push}_x \prec_H \text{Pop}_i \), \( \text{Push}_x \) will not remove the node inserted by \( \text{Push}_x \) (Because \( \text{Pop}_i \) will remove the node with the youngest timestamp). This contradicts the fact. If \( \text{Pop}_i \) accesses the node inserted by \( \text{Push}_x \), then by \( \text{Push}_i \prec_{ts} \text{Push}_x \), \( \text{Pop}_i \) will not remove the node inserted by \( \text{Push}_i \). This also contradicts the fact.

2. If \( \text{Match}(\text{Pop}_i) = \epsilon \), then (a) \( \forall \text{push} \in \text{PUSH}(H) \wedge \text{push} \prec_H \text{Pop}_i \implies \text{push} \in \{ \text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1}) \} \); (b) let \( \text{PPN} = \{ \text{push} \mid \text{push} \simeq_H \text{Pop}_i \wedge \text{push} \in (\text{PUSH}(H) \setminus \{ \text{Match}(\text{Pop}_1), \ldots, \text{Match}(\text{Pop}_{i-1}) \}) \}, \) then \( \forall \text{push}, \text{x} \cdot x \leq i - 1 \wedge \text{push} \in \text{PPN} \implies \text{push} \not\prec_{H} \text{Match}(\text{Pop}_x) \).

Proof. If \( \text{Match}(\text{Pop}_i) = \epsilon \), all lists are empty at the time point when \( T_{15} \) is executed. Thus the nodes inserted by the push operations which precede \( \text{Pop}_i \) are removed by the pop operations in front of \( \text{Pop}_i \). Thus the first clause of the condition is valid. When the statement \( T_{15} \) of \( \text{Pop}_i \) is executed, the pop operations in the set \( \text{PPN} \) do not complete the inserting node action, however, the pop operations in front of \( \text{Pop}_i \) and their matching push operations complete the actions of deleting and inserting nodes, respectively. Thus the second clause is valid.

An example execution of the TS stack (adapted from \([X]\)) is depicted in fig. 7. We will construct a linearization of the execution history by using the method in the proving process of Theorem 5.1. Let \( \text{push}(v, t)/\text{pop}(v, t) \) denote the push/pop operation which pushes/pops the value \( v \) and generates the timestamp \( t \). The black circles of the pop operations in fig.x stand for the actions of removing nodes (at \( T_{20} \)).

The above execution is feasible. For example, consider the execution of Thread 5. \( T_5 \) first accesses \( T_{20}'s \) associated list and chooses the node inserted by \( T_1 \) as a candidate. \( T_5 \) continues to access \( T_{20}'s \) associated list before \( T_{20}'s \) inserts a node into the list. Finally, \( T_5 \) accesses \( T_{20}'s \) associated list after \( T_5 \) removes a node from the list. Thus, the pop operation of \( T_5 \) removes the node inserted by \( T_1 \).

First, we construct the following linearization of push operations using the method shown in Step 1 in the proof of Theorem 5.1.

\[ \text{push}(x, 1), \text{push}(y, 3), \text{Push}(z, 4) \]

Second, we construct the following linearization of pop operations in terms of the actions of removing nodes.

\[ \text{pop}(z, 5), \text{pop}(x, 2), \text{pop}(y, 6) \]

Finally, we insert the pop operations , one after another, into the linearization sequence of push operations. By the algorithm 1 and \( \text{push}(z, 4) \prec \text{pop}(z, 5) \), we first insert \( \text{pop}(z, 5) \) to the end of the push sequence and get the following sequence.

\[ \text{push}(x, 1), \text{push}(y, 3), \text{push}(z, 4), \text{pop}(z, 5) \]

By the algorithm 1 and \( \text{push}(x, 1) \prec \text{pop}(x, 2) \), we insert \( \text{pop}(x, 2) \) into the right of \( \text{push}(x, 1) \) and get the following sequence.

\[ \text{push}(x, 1), \text{pop}(x, 2), \text{push}(y, 3), \text{push}(z, 4), \text{pop}(z, 5) \]

Because \( \text{push}(z, 4) \prec \text{pop}(y, 6) \) and \( \text{push}(z, 4) \) is followed by \( \text{pop}(z, 5) \), we insert \( \text{pop}(y, 6) \) into the right of \( \text{pop}(z, 5) \) and get the final sequence.

\[ \text{push}(x, 1), \text{pop}(x, 2), \text{push}(y, 3), \text{push}(z, 4), \text{pop}(z, 5), \text{pop}(y, 6) \]

Obviously, the final sequence does not violate the happened-before order, satisfies the “FILO” semantics and is a linearization of the above execution. Note that the final linearization of pop operations \( \text{pop}(x, 2), \text{pop}(z, 5), \text{pop}(y, 6) \) is difference from the original linearization of pop operations \( \text{pop}(z, 5), \text{pop}(x, 2), \text{pop}(y, 6) \).
8. VERIFYING OTHER STACKS

The Treiber stack[16] is based on a singly-linked list with a top pointer, is a lock-free concurrent stack. The push and pop methods of the stack try to update the top pointer using cas instructions to finish their operations. We choose the cas instruction which successfully removes the head node as the linearization point of the pop method and construct the initial linearization of pop operations in terms of them. Obviously, the head node is inserted by the latest push operation among all nodes of the current list. Thus, each pop operation always removes the node pushed by the latest push operation in the initial order of pop operations.

The HSY stack[12] is also based on a singly-linked list. similar to the Treiber stack, the HSY stack first tries to update the top pointer by cas instructions to finish the push or pop operations. If the cas instructions fail to update the top pointer, then the HSY stack use elimination mechanism to finish the operations. By Theorem 4.1, we only need to consider the executions of the common push and pop operations (not elimination pairs). The linearizability verification of the common operations is similar to the one of the Treiber stack.

The FA-Stack[4] is a fast array-based concurrent stack. A past path of a push operation attempts to store an element at the global array. If the past path fails in storing, the push operation switches to a slow-path. In the slow-path, the push operation publishes a push request to enlist help of pop operations. Thus the pop operations try to help it to store the element. Similar to push operations, a past path of a pop operation attempts to find an element to be popped. If the past path fails in popping or returning empty, the pop operation switches to a slow-path. In the slow-path, the pop operation publishes a pop request to enlist help and then other pop operations help it to pop an element or return an empty. For a pop operation popping an element from a fast path, we choose the successful CAS action \( \text{CAS}(&c \rightarrow \text{pop}, \bot_{\text{pop}}, \top_{\text{pop}}) \), which logically removes an element from the cell \( c \), as a linearization point of the pop operation. For a pop operation popping an element from a slow path, we choose the successful CAS action \( \text{CAS}(&c \rightarrow \text{pop}, \bot_{\text{pop}}, r) \), which reserves the cell \( c \) for the pop request \( r \), as a linearization point of the pop operation. Once the cell is reserved for a pop operation, other pop operations do not remove the element from the cell. Thus, the reserving action is also a logical removing action. We show that the FA-Stack is linearizable by using the following two invariants.

Invariant 1. Assume that \( c[i] \) is the current cell visited by a pop operation (or pop helper), then the pop operation continues to visit the predecessor cell of \( c[i] \) only after ensuring that \( c[i] \) is unusable, popped by other pop operations, or reserved for other pop requests.

Invariant 2. Assume that two push operations \( push_1, push_2 \) store their elements at the two cells \( c[i], c[j] \), respectively. If \( j > i \), then \( push_j \neq push_i \).

Assume that a pop operation logically removes an element from the cell \( c[i] \). By Invariant 1, when the pop operation logically removes the element from the cell \( c[i] \), then for all \( j > i \), the element from the cell \( c[j] \) have been removed logically. By Invariant 2, when the pop operation logically removes the element from the cell \( c[i] \), the element from \( c[j] \) is inserted by the latest push operation in the current stack.

Afek et al. propose a simple array-based stack [17]. The stack is represented as an infinite array, and a marker, rang, pointing to the end of the used part of the array. To push an element, a push operation first obtains a cell index by incrementing range and then stores the element at the cell. A pop operation first reads the range, and then searches from range to the first cell to see if it contains a non-NULL element. If it finds such an element, then it removes and returns the element. Otherwise, it returns empty. For the pop operation which does not return empty, we choose the successful swap action \( \text{swap}(item[i], \text{null}) \), which returns a non-null value, as linearization point of the pop operation. We show that the concurrent stack is linearizable by using the invariants similar to the above two invariants.

9. RELATED WORK AND CONCLUSION

There has been a great deal of work on linearizability verification[18-25]. 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 [25]. However, as Khyzha et al. argue [20], it remains the case that all but the simplest algorithms are difficult to verify.

To the best of our knowledge, there exist only two earlier published full proofs of the TS stack: (1) the original proof by Dodds et al.[15], and (2) a forward simulation proof by Bouajjani et al.[18].

In the original proof of the TS stack, Dodds et al. have proposed a set of conditions sufficient to ensure linearizability of concurrent stacks. In addition to the happened-before order, their conditions require an auxiliary insert-remove order which relates pushes to pops and vice versa, and two helper orders \( \text{ins} \) and \( \text{rem} \) over push operations and pop operations, respectively. However, it is difficult to show that there exists an insert-remove order that satisfies the definition of order correctness. For applying the method directly on the TS stack, they have to construct an intermediate structure called the TS buffer. This made the linearization proof of the TS stack be complex and not be intuitive. Although our method requires a linear order of the pop operations, the linear order can be easily constructed, as discussed above. The conditions in our theorem intuitonally express that the “LIFO” semantics of
a concurrent stack and lead to simple and natural correctness proofs.

Bouajjani et al. propose a forward simulation based technique for verifying linearizability. They have successfully applied the method to verify the TS stack and the HW queue. In fact, for the TS stack, there does not exist a forward simulation to the standard sequential stack and they have to construct a deterministic atomic reference implementation (as an intermediate specification) for the TS stack, and the linearizability proof is reduced to showing that the TS stack is forward-simulated by the intermediate specification. In comparison, our proof technique is simpler and more intuitive.

Conclusion We present a simple proof technique for verifying linearizability of concurrent stacks. Our technique reduce the problem of proving linearizability of concurrent stacks to establishing a set of conditions. The conditions can be easily verified just by reasoning about happened-before orders of operations, verifiers do not need to know other proof techniques, and can easily and quickly learn to use the proof technique. We have successfully applied the method to several challenging concurrent stacks: the TS stack, the HSY stack and the FA stack. Our proof technique is suitable for automation, as it requires just checking the key invariant: when a common pop operation removes a value in the stack, the value is the latest value in the current stack. In the future, we would like to build a fully automated tool for the proof technique.

10. FUNDING

This work was supported by the National Nature Science Foundation of China [61020106009, 61272075].

REFERENCES

[1] Herlihy, M., and Wing, J. (1990) Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems, 12, 463-492.
[2] Herlihy, M., and Shavit, N. (2008) The Art of Multiprocessor Programming. Morgan Kaufmann, Massachusetts.
[3] I. Filipović, P. W. O’Hearn, N. Rinetzky, and H. Yang. Abstraction for concurrent objects. Theor. Comput. Sci., 411(51-52):4379-4398, 2010.
[4] Peng, Yaqiong, and Zhiyu Hao. Fa-stack: A fast array-based stack with wait-free progress guarantee. IEEE Transactions on Parallel and Distributed Systems 29.4 (2017): 843-857.
[5] Haas, A. (2015) Fast concurrent data structures through timestamping. PhD thesis, University of Salzburg. http://www.cs.uni-salzburg.at/~ahaas/papers/thesis.pdf.
[6] Yang, C., Mellor-Crummey, J. (2016, February). A wait-free queue as fast as fetch-and-add. In Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (pp. 1-13).
[7] Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer III, W. N., and Shavit, N., [2005] A lazy concurrent list-based set algorithm. 9th International Conference on Principles of Distributed Systems, LNCS 3974, Pisa, Italy, 12-14 December, pp. 3-16. Springer-Verlag, Berlin.
[8] Hoffman, M., Shalev, O., and Shavit, N. (2007) The baskets queue. 11th International Conference on Principles of Distributed Systems, LNCS 4878, Guadeloupe, French West Indies, 17-20 December, pp. 401-414. Springer-Verlag, Berlin.
[9] Henzinger, T. A., Sezgin, A., and Vafeiadis, V. (2013) Aspect-oriented linearizability proofs. 24th International Conference On Concurrency Theory (CONCUR 2013), LNCS 8052, Buenos Aires, Argentina, 27-30 August, pp. 242-256. Springer-Verlag, Berlin.
[10] Wen, T., Song, L., You, Z. (2019). Proving Linearizability Using Reduction. The Computer Journal, 62(9), 1342-1364.
[11] Liang, H., and Feng, X. (2013) Modular verification of linearizability with non-fixed linearization points. Conference on Programming Language Design and Implementation (PLDI 2013), Seattle, WA, USA, 16-19 June, pp. 459-470. ACM, New York.
[12] N. Shavit and D. Touitou. Elimination trees and the construction of pools and stacks. Theory of Computing Systems, 30:645-670, 1997.
[13] Moir, M., Nussbaum, D., Shalev, O., Shavit, N. (2005, July). Using elimination to implement scalable and lock-free fifo queues. In Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures (pp. 253-262).
[14] Hendler, D., Shavit, N., and Yerushalmi, L. (2004) A scalable lock-free stack algorithm. 16th annual ACM symposium on Parallelism in algorithms and architectures (SPAA 2004), Barcelona, Spain, 27-30 June, pp. 206-215. ACM, New York.
[15] M. Dodds, A. Haas, and C. M. Kirsch. A scalable, correct time-stamped stack. In POPL, 2015.
[16] Treiber, R. K. (1986) System programming: coping with parallelism. Technical Report RJ 5118, IBM Almaden Research Center. http://opac.inria.fr/record=b1015261.
[17] Y. Afek, E. Gafni, and A. Morrison, Common2 extended to stacks and unbounded concurrency, in Proc. ACM Symp. Principles Distrib. Comput., 2006, pp. 218-227
[18] Khyzha, A., Dodds, M., Gotsman, A., and Parkinson, M. (2017) Proving linearizability using partial orders. 26th European Symposium on Programming, LNCS 10201, Uppsala, Sweden, 22-29 April, pp. 639-667. Springer-Verlag, Berlin.
[19] Singh, V., Neamtiu, I., and Gupta, R. (2016) Proving concurrent data structures linearizable. 27th International Symposium on Software Reliability Engineering (ISSRE 2016), Ottawa, ON, Canada, 23-27 October, pp. 230-240. IEEE Computer Society, Los Alamitos, California.
[20] Schellhorn, G., Wehrheim, H., and Derrick, J. (2012). How to prove algorithms linearisable. International Conference on Computer Aided Verification (CAV 2012), LNCS 7358, Berkeley, CA, USA, 7-13 July, pp. 243-259. Springer-Verlag, Berlin.
[21] Bouajjani, A., Emmi, M., Enea, C., and Mutluergil, S. O. (2017) Proving linearizability using forward simulations. *International Conference on Computer Aided Verification (CAV 2017)*, LNCS 10427, Heidelberg, Germany, 24-28 July, pp. 542-563. Springer-Verlag, Cham.

[22] Abdulla, P. A., Jonsson, B., and Trinh, C. Q. (2018) Fragment abstraction for concurrent shape analysis. *European Symposium on Programming (ESOP 2018)*, LNCS 10801, Thessaloniki, Greece, 14-20 April, pp. 442-471. Springer-Verlag, Cham.

[23] Vindum, S. F., Frumin, D., Birkedal, L. (2021). Mechanized Verification of a Fine-Grained Concurrent Queue from Facebook’s Folly Library.

[24] Feldman, Y. M., Khyzha, A., Enea, C., Morrison, A., Nanevski, A., Rinetzky, N., Shoham, S. (2020). Proving highly-concurrent traversals correct. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA), 1-29.

[25] Dongol, Brijesh, and John Derrick. Verifying linearizability: A comparative survey, *ACM Computing Surveys (CSUR)* 48.2 (2015): 1-43.