Verifying Linearizability

Jad Hamza

EPFL

19 December 2017
Why Linearizability? Ensuring Observational Refinement

If the implementation $L$ is linearizable (with respect to an atomic specification $S$), then for any user/client program $P$, we have:

$$P[L] \subseteq P[S]$$

i.e., $P$ produces less behaviors when using $L$ than when using $S$.

Application: If we prove a safety property on a program $P$ using an atomic queue $S$, we can replace the atomic queue by a (more efficient) concurrent linearizable implementation $L$, and the safety property will still hold.
If the implementation \( L \) is linearizable (with respect to an atomic specification \( S \)), then for any user/client program \( P \), we have:

\[
P[L] \subseteq P[S]
\]
Why Linearizability? Ensuring Observational Refinement

If the implementation $L$ is linearizable (with respect to an atomic specification $S$), then for any user/client program $P$, we have:

$$P[L] \subseteq P[S]$$

i.e., $P$ produces less behaviors when using $L$ than when using $S$
Why Linearizability? Ensuring Observational Refinement

If the implementation $L$ is linearizable (with respect to an atomic specification $S$), then for any user/client program $P$, we have:

$$P[L] \subseteq P[S]$$

i.e., $P$ produces less behaviors when using $L$ than when using $S$

**Application**: If we prove a safety property on a program $P$ using an atomic queue $S$, we can replace the atomic queue by a (more efficient) concurrent linearizable implementation $L$, and the safety property will still hold.
Events and Trace Example

Thread 1

- enq(1) → ret
- deq()

Thread 2

- deq() → ret(1)
- enq(3) → ret
- enq(2) → ret

- ret(3) → deq() → ret(2)
Events and Traces

Definition (Events)

A call event is a tuple with a thread identifier, a method name, and a parameter.
Events and Traces

**Definition (Events)**

A **call event** is a tuple with a **thread identifier**, a **method name**, and a **parameter**.

A **return event** is a pair with a **thread identifier** and a **return value**.
Events and Traces

Definition (Events)

A call event is a tuple with a thread identifier, a method name, and a parameter.

A return event is a pair with a thread identifier and a return value.

Definition (Trace)

A trace is a sequence of call and return events.
An **operation** is a tuple with a **thread identifier**, a **method name**, a **parameter** and a **return value**.
(corresponds to a pair of matching call and return events)
Linearization Points Example

The trace is linearizable to \( enq(1) \cdot enq(2) \cdot deq(1) \cdot deq(2) \).
And also linearizable to \( enq(1) \cdot deq(1) \cdot enq(2) \cdot deq(2) \).
And also linearizable to \( deq(1) \cdot enq(1) \cdot enq(2) \cdot deq(2) \). (not a valid Queue sequence)
The trace is linearizable to 
\[ enq(1) \cdot enq(2) \cdot deq(1) \cdot deq(2). \]
And also linearizable to 
\[ enq(1) \cdot deq(1) \cdot enq(2) \cdot deq(2). \]
And also linearizable to 
\[ deq(1) \cdot enq(1) \cdot enq(2) \cdot deq(2). \] (not a valid Queue sequence)
The trace is linearizable to $enq(1) \cdot enq(2) \cdot deq(1) \cdot deq(2)$. And also linearizable to $enq(1) \cdot deq(1) \cdot enq(2) \cdot deq(2)$. And also linearizable to $deq(1) \cdot enq(1) \cdot enq(2) \cdot deq(2)$. (not a valid Queue sequence)
Linearization Points Example

The trace is linearizable to $enq(1) \cdot enq(2) \cdot deq(1) \cdot deq(2)$. 
Linearization Points Example

The trace is linearizable to $enq(1) \cdot enq(2) \cdot deq(1) \cdot deq(2)$.
And also linearizable to $enq(1) \cdot deq(1) \cdot enq(2) \cdot deq(2)$. 
The trace is linearizable to $\text{enq}(1) \cdot \text{enq}(2) \cdot \text{deq}(1) \cdot \text{deq}(2)$.

And also linearizable to $\text{enq}(1) \cdot \text{deq}(1) \cdot \text{enq}(2) \cdot \text{deq}(2)$.

And also linearizable to $\text{deq}(1) \cdot \text{enq}(1) \cdot \text{enq}(2) \cdot \text{deq}(2)$. (not a valid Queue sequence)
Linearization Points

Definition (Linearizability)

A trace $t$ is \textbf{linearizable} with respect to a sequence of operations $w$, denoted $t \sqsubseteq w$ if, for each operation $o$, we can find a \textbf{point} (called linearization point) between the call and return event of $o$ such that:

the obtained sequence of operations is $w$. 
History

Definition (History)

A **history** \( h = (O, <) \) is a strict partial order (irreflexive and transitive) over a set of operations \( O \).
A history \( h = (O, <) \) is a strict partial order (irreflexive and transitive) over a set of operations \( O \).

For a trace \( t \), we define the history \( \text{hist}(t) \) to be \( (O, <) \) where:

- \( O \) is the set of operations that appear in \( t \)
- for \( o_1, o_2 \in O, o_1 < o_2 \) iff the return event of \( o_1 \) is before the call event of \( o_2 \) in \( t \).
Example of Trace/History
Example of Trace/History

Thread 1
- enq(1)
- ret
- deq()
- ret
- enq(2)
- ret
- deq()
- ret
- enq(3)
- ret

Thread 2
- deq(2)
- enq(1)
- deq(1)
- enq(3)
- deq(3)
- enq(2)
Another Definition for Linearizability

**Definition (Linearizability of a history)**
We say that a **history** $h = (O, \prec)$ is **linearizable** with respect to a sequence $w$, denoted $h \sqsubseteq w$ if we can obtain $w$ by reordering the operations of $h$, while respecting the order $\prec$. 
Another Definition for Linearizability

Definition (Linearizability of a history)

We say that a **history** \( h = (O, <) \) is **linearizable** with respect to a sequence \( w \), denoted \( h \sqsubseteq w \) if we can obtain \( w \) by reordering the operations of \( h \), while respecting the order \(<\).

\(<\) must be a **subset** of the total order given by \( w \): \(< \subseteq <_w\)
Another Definition for Linearizability

Definition (Linearizability of a history)

We say that a history $h = (O, <)$ is linearizable with respect to a sequence $w$, denoted $h \sqsubseteq w$ if we can obtain $w$ by reordering the operations of $h$, while respecting the order $<$.

$<$ must be a subset of the total order given by $w$: $< \subseteq <_w$

Definition (Linearizability of a trace)

A trace $t$ is linearizable with respect to $w$, denoted $t \sqsubseteq w$ if $\text{hist}(t)$ is linearizable with respect to $w$. 
Example

Linearizable to:

tenq(1) · deq(1) · enq(3) · deq(3) · enq(2) · deq(2).

Example

Linearizable to: $enq(1) \cdot deq(1) \cdot enq(3) \cdot deq(3) \cdot enq(2) \cdot deq(2)$. 
Linearizability

Definition (Specification)
A specification $S$ is a set of sequences.
Linearizability

**Definition (Specification)**
A **specification** \( S \) is a set of sequences.

**Definition (Linearizability with respect to a specification)**
A **history** \( h \) is linearizable with respect to a **specification** \( S \), denoted \( h \sqsubseteq S \), if there exists \( w \in S \) such that \( h \sqsubseteq w \).
Linearizability

Definition (Specification)
A specification $S$ is a set of sequences.

Definition (Linearizability with respect to a specification)
A history $h$ is linearizable with respect to a specification $S$, denoted $h \sqsubseteq S$, if there exists $w \in S$ such that $h \sqsubseteq w$.

Definition (Linearizability of a library)
A library $L$ is linearizable with respect to $S$, denoted $L \sqsubseteq S$ if every history/trace produced by $L$ is linearizable with respect to $S$. 
Linearizability checking problems

Problem (Testing)

Given a **history** $h$, and a **specification** $S$, check whether $h \sqsubseteq S$
Linearizability checking problems

Problem (Testing)

Given a **history** \( h \), and a **specification** \( S \), check whether \( h \sqsubseteq S \)

Problem (Verification)

Given a **library** \( L \), and a **specification** \( S \), check whether \( L \sqsubseteq S \).

(\textit{check} \( h \sqsubseteq S \) for every \( h \) in \( L \))
Motivation for Testing: Bug-Finding

- Enumerate many traces of a library
- Check for each one, individually, whether it is linearizable
Motivation for Testing: Bug-Finding

- Enumerate *many traces* of a library
- Check for each one, individually, whether it is linearizable

If we find a non-linearizable trace, we found a **bug**.
Motivation for Testing: Bug-Finding

• Enumerate many traces of a library
• Check for each one, individually, whether it is linearizable

If we find a non-linearizable trace, we found a bug.

Limitation of testing: cannot verify that all the traces of a library are linearizable (there are infinitely many traces)
Bruteforce Algorithm

Given \( h = (O, <) \) and \( S \), check whether \( h \sqsubseteq S \):

- If there exists a permutation \( w \) of \( O \) such that \( w \) respects \( < \) and \( w \in S \), return true
- Otherwise, return false

Worst case: \(|O|! \) permutations to explore
Bruteforce Algorithm

Given $h = (O, \prec)$ and $S$, check whether $h \sqsubseteq S$:

- If there exists a permutation $w$ of $O$ such that $w$ respects $\prec$ and $w \in S$, return true
- Otherwise, return false
Bruteforce Algorithm

Given $h = (O, <)$ and $S$, check whether $h \subseteq S$:

- If there exists a permutation $w$ of $O$ such that $w$ respects $<$ and $w \in S$, return \textbf{true}
- Otherwise, return \textbf{false}

Worst case: $|O|!$ permutations to explore
Example

Check each of the $6! = 720$ permutations.
Example

Check each of the $6! = 720$ permutations.
Example (minor improvement)
Example (minor improvement)

Start from the **minimal nodes**, and only explore linearizations that respect $<$ and the **specification**.
Another Bruteforce Algorithm

Let $S$ be a specification. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.
Another Bruteforce Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)
Another Bruteforce Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$.
(coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
```

Another Brute-force Algorithm

Let $S$ be a specification. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix · o // add o to the prefix
    isMinimal(h, o) &&
    newPrefix ∈ S &&
    isLinearizable(newPrefix, h - o)
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore
Another Brute-force Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix \cdot o // add o to the prefix
    isMinimal(h, o) &&
    newPrefix \in S &&
    isLinearizable(newPrefix, h - o)
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore.
Another Bruteforce Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, \prec)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  if h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix \ o // add o to the prefix
    isMinimal(h, o) &&
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore
Another Bruteforce Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```java
def isLinearizable(prefix: Seq[Operations], h: History):
    Boolean = {
        h.isEmpty || // if h is empty, we are done!
        h.operations.exists { o =>
            val newPrefix = prefix \ o // add o to the prefix
            isMinimal(h, o) &&
            newPrefix \ S &&
        }
    }
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|\text{O}|!$ permutations to explore.
Another Bruteforce Algorithm

Let $S$ be a specification. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix · o // add o to the prefix
    isMinimal(h, o) &&
    newPrefix ∈ S &&
    isLinearizable(newPrefix, h - o)
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore
Another Bruteforce Algorithm

Let $S$ be a **specification**. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, \prec)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```java
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  if (h.isEmpty) { // if h is empty, we are done!
    h.operations.exists { o =>
      val newPrefix = prefix \ o // add o to the prefix
      isMinimal(h, o) &&
      newPrefix \in S &&
      isLinearizable(newPrefix, h - o)
    }
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore
Another Brute-force Algorithm

Let $S$ be a specification. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operation], h: History): Boolean = {
  h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix · o // add o to the prefix
    isMinimal(h, o) &&
    newPrefix ∈ S &&
    isLinearizable(newPrefix, h - o)
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable(Seq()), h}$ holds.
Another Bruteforce Algorithm

Let $S$ be a specification. Let $\text{prefix} \in S$ be a sequence of operations and $h = (0, <)$ be a history.

Check if there exists $w$ such that $h \sqsubseteq w$ and $\text{prefix} \cdot w \in S$. (coincides with $h \sqsubseteq S$ when $\text{prefix}$ is empty)

```scala
def isLinearizable(prefix: Seq[Operations], h: History): Boolean = {
  h.isEmpty || // if h is empty, we are done!
  h.operations.exists { o =>
    val newPrefix = prefix \ o // add o to the prefix
    isMinimal(h, o) &&
    newPrefix \ S &&
    isLinearizable(newPrefix, h - o)
  }
}
```

For a history $h$, we have $h \sqsubseteq S$ iff $\text{isLinearizable}(\text{Seq}(), h)$ holds.

Worst case: still $|O|!$ permutations to explore
Polynomial-Time Algorithm for Testing?

Theorem (Gibbons & Korach 97)

Given $h$ and $S$, checking $h \sqsubseteq S$ is NP-complete.
Polynomial-Time Algorithm for Testing?

**Theorem (Gibbons & Korach 97)**

Given \( h \) and \( S \), checking \( h \sqsubseteq S \) is NP-complete.

\( \Rightarrow \) No *polynomial-time* algorithm, unless \( P = NP \)
Polynomial-Time Algorithm for Testing?

Theorem (Gibbons & Korach 97)

Given $h$ and $S$, checking $h \sqsubseteq S$ is NP-complete.

$(\Rightarrow)$ No polynomial-time algorithm, unless $P = NP$

However, there are polynomial-time algorithms if we look at particular specifications $S$. 
Testing Problem for Queues

Problem (Linearizability for Queues)

Given a **history** $h$, check whether $h \sqsubseteq \text{Queue}$. 
Bad Pattern 1 and Bad Pattern 1’

A dequeue operation with no corresponding enqueue.

- (BP1) a $\text{deq}(1)$ such that $\text{enq}(1)$ does not exist at all
- (BP1’) two or more $\text{deq}(1)$ (this is bad because we assume enqueues are unique)
Bad Pattern 2

Two enqueue’s $enq(1) < enq(2)$ such that $deq(2) < deq(1)$.
(if $deq(1)$ isn’t in the history, we pose that $deq(2) < deq(1)$ holds)
Bad Pattern 3 (Example A)

A \textit{deq}(empty) operation \textcolor{red}{covered} by pairs of enqueue/dequeue.
Bad Pattern 3 (Example B)

A `deq(empty)` operation covered by pairs of enqueue/dequeue.
Bad Pattern 3 (Example C)

A \textit{deq(\textit{empty})} operation \textbf{covered} by pairs of enqueue/dequeue.
Defining Bad Pattern 3 Formally

Given a history \( h = (O, <) \), and some \( deq(empty) \) operation in \( O \), we construct a graph \( G \) such that:

- the vertices of \( G \) are the values that are enqueued in \( h \) and a vertex for the \( deq(empty) \) operation
- there is an edge from \( v_1 \) to \( v_2 \) iff \( enq(v_1) < deq(v_2) \)
- there is an edge from \( deq(empty) \) to \( v \) iff \( deq(empty) < deq(v) \)
- there is an edge from \( v \) to \( deq(empty) \) iff \( enq(v) < deq(empty) \)

**Definition**

The operation \( deq(empty) \) is **covered** iff there is a **cycle** going through \( deq(empty) \) in the graph.
Bad Patterns

- (BP1) a \textit{deq}(v) such that there exists no \textit{enq}(v)
- (BP1') two \textit{deq}(v) operations (or more)
- (BP2) two enqueue operations dequeued in the \textit{wrong order}
- (BP3) a \textit{deq}(\textit{empty}) operation which is \textit{covered}
Bad Patterns

- (BP1) a $deq(v)$ such that there exists no $enq(v)$
- (BP1') two $deq(v)$ operations (or more)
- (BP2) two enqueue operations dequeued in the wrong order
- (BP3) a $deq(\text{empty})$ operation which is covered

Theorem (Bad Patterns)

Let $h$ be a history (with unique enqueues).

Then $h \sqsubseteq \text{Queue}$ if and only if $h$ doesn’t contain one of these bad patterns
Polynomial-time algorithm

We can check in polynomial-time if $h$ has a bad pattern.

**Theorem**

Let $h$ be a history (with unique enqueues).
We can check $h \subseteq \text{Queue}$ in *polynomial-time*.

**Proof.**

Check for the absence of bad patterns. Each one can be checked in polynomial-time.

- (BP1) a $\text{deq}(v)$ such that there exists no $\text{enq}(v)$
- (BP1’) two $\text{deq}(v)$ operations (or more)
- (BP2) two enqueue operations dequeued in the *wrong order*
- (BP3) a $\text{deq}(\text{empty})$ operation which is *covered*
Polynomial-time algorithm

We can check in polynomial-time if $h$ has a bad pattern.

**Theorem**

Let $h$ be a history (with unique enqueues).
We can check $h \sqsubseteq \text{Queue}$ in **polynomial-time**.

**Proof.**

Check for the absence of bad patterns. Each one can be checked in polynomial-time.

- (BP1) a $\text{deq}(v)$ such that there exists no $\text{enq}(v)$
- (BP1') two $\text{deq}(v)$ operations (or more)
- (BP2) two enqueue operations dequeued in the **wrong order**
- (BP3) a $\text{deq}($empty$)$ operation which is **covered**
Limitations of Testing

Checking that \( h \subseteq S \) one by one, we can never be sure that \( L \subseteq S \). A library produces an **infinite** amount of traces histoires.
Herlihy & Wing Queue

var table = Map[Int, Value]() // represents the queue
var n: Int = 0 // index of the next enqueue
Herlihy & Wing Queue

```scala
var table = Map[Int,Value]() // represents the queue
var n: Int = 0 // index of the next enqueue

def enqueue(v: Value): Unit = {
  synchronized { i = n; n = n + 1 } // atomic operation
  table(i) = v
}
```

Herlihy & Wing Queue

```scala
var table = Map[Int, Value]() // represents the queue
var n: Int = 0 // index of the next enqueue

def enqueue(v: Value): Unit = {
    synchronized { i = n; n = n + 1 } // atomic operation
    table(i) = v
}

def dequeue(): Value = {
    while (true) {
        val m = n
        for (k <- 0 to m - 1) {
            // get the element at index k, and write null instead
            val v = SWAP(table(k), null)
            // if not null, return the element
            if (v != null)
                return v
        }
    }
}
```
H&W Queue is Linearizable

Theorem

The H&W Queue $L_{H&W}$ is linearizable, i.e. $L_{H&W} \sqsubseteq Queue$. 

Proof.

We prove that $h \sqsubseteq Queue$ for every $h \in L_{H&W}$. It suffices to prove that $h$ has no bad pattern. We assume that $h$ has a bad pattern and derive a contradiction.

• BP1: Not possible because dequeue always returns values from the map, and the map always contains values that were previously enqueued.

• BP1': Not possible when assuming unique enqueues, and due to the atomicity of SWAP.

• BP2: Not possible as the first enqueue operation will be stored at a smaller index in the table.

• BP3: Not possible because dequeue never returns empty.
H&W Queue is Linearizable

Theorem

The H&W Queue $L_{H&W}$ is linearizable, i.e. $L_{H&W} \subseteq \text{Queue}$.

Proof.

We prove that $h \subseteq \text{Queue}$ for every $h \in L_{H&W}$. It suffices to prove that $h$ has no bad pattern. We assume that $h$ has a bad pattern and derive a contradiction (stetch)
H&W Queue is Linearizable

Theorem

The H&W Queue $L_{H&W}$ is linearizable, i.e. $L_{H&W} \subseteq \text{Queue}$.

Proof.

We prove that $h \sqsubseteq \text{Queue}$ for every $h \in L_{H&W}$. It suffices to prove that $h$ has no bad pattern. We assume that $h$ has a bad pattern and derive a contradiction (stetch)

- BP1: Not possible because dequeue always returns values from the map, and the map always contains values that were previously enqueued.
H&W Queue is Linearizable

Theorem

The H&W Queue \( L_{H&W} \) is linearizable, i.e. \( L_{H&W} \sqsubseteq Queue \).

Proof.

We prove that \( h \sqsubseteq Queue \) for every \( h \in L_{H&W} \). It suffices to prove that \( h \) has no bad pattern. We assume that \( h \) has a bad pattern and derive a contradiction (stetch)

- BP1: Not possible because dequeue always returns values from the map, and the map always contains values that were previously enqueued.
- BP1': Not possible when assuming unique enqueues, and due to the atomicity of SWAP.
H&W Queue is Linearizable

Theorem

The H&W Queue $L_{H&W}$ is linearizable, i.e. $L_{H&W} \subseteq Queue$.

Proof.

We prove that $h \subseteq Queue$ for every $h \in L_{H&W}$. It suffices to prove that $h$ has no bad pattern. We assume that $h$ has a bad pattern and derive a contradiction (stetch)

- BP1: Not possible because dequeue always returns values from the map, and the map always contains values that were previously enqueued.
- BP1’: Not possible when assuming unique enqueues, and due to the atomicity of SWAP.
- BP2: Not possible as the first enqueue operation will be stored at a smaller index in the table
The H&W Queue is Linearizable

Theorem

The H&W Queue $L_{H&W}$ is linearizable, i.e. $L_{H&W} \subseteq Queue$.

Proof.

We prove that $h \subseteq Queue$ for every $h \in L_{H&W}$. It suffices to prove that $h$ has no bad pattern. We assume that $h$ has a bad pattern and derive a contradiction (stetch)

- **BP1**: Not possible because dequeue always returns values from the map, and the map always contains values that were previously enqueued.
- **BP1’**: Not possible when assuming unique enqueues, and due to the atomicity of SWAP.
- **BP2**: Not possible as the *first* enqueue operation will be stored at a smaller index in the table
- **BP3**: Not possible because dequeue never returns empty
Summary

- Testing for **finding bugs**
- Verification for finding bugs or proving correctness
- Checking linearizability for one trace is **NP-complete**
- But, **polynomial-time** if we restrict the specification to Queue/Stack and histories with unique enqueues/pushes
- It is enough to check for **bad patterns**
- Careful: Stack bad patterns are **not symmetric** wrt Queue

References:
(1) Aspect-Oriented Linearizability Proofs. Chakraborty et al.
(2) On Reducing Linearizability to State Reachability. Bouajjani et al.