Checking Linearizability: Theoretical Limits

Constantin Enea
IRIF, Univ Paris Diderot - Paris 7
Concurrent Objects
Multi-threaded programming

e.g. Java Development Kit SE
dozens of objects, including queues, maps, sets, lists, locks, atomic integers, …
Linearizability

- Each history $\delta$ induces a partial order on operations such that
  - $o_1 \sqsubseteq_\delta o_2$ iff $\text{ret } o_1$ occurs before $\text{call } o_2$ in $\delta$
- A history $\delta$ is Linearizable if there exists an equivalent *Sequential* history $\delta'$ (i.e. same operations), and
  - $o_1 \sqsubseteq_\delta o_2$ implies $o_1 \sqsubseteq_{\delta'} o_2$
- Ignoring uncompleted operations
- Strictly stronger than Sequential Consistency
Linearizability [Herlihy&Wing 1990]

Effects of each invocation appear to occur instantaneously.

Execution history

Linearization admitted by Queue ADT

\( \exists \text{ lin. } rb \subseteq \text{ lin } \land \text{ lin } \in \text{ Queue ADT} \)

returns-before (rb)
Efficient Concurrent Implementations

- Avoid the use of locks
- Maximise parallelisation of operations
- Check for interferences, and retry
- Use lower level synchronisation primitives (CAS)

- => Complex behaviours!
- => Need to ensure the atomic view to the user!
Example: Treiber Stack

class Node {
    Node tl;
    int val;
}

class NodePtr {
    Node val;
    int val;
}

void push(int e) {
    Node y, n;
    y = new();
    y->val = e;
    while(true) {
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {
    Node y, z;
    while(true) {
        y = TOP->val;
        if (y==0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}
Hand-over-Hand Set

- A set implementation based on sorted linked lists

```java
public class Entry {
    public Object value;
    public Entry next;
}

public class Set {
    Entry first;
    public boolean add(Object x) {...}
    public boolean remove(Object x) {...}
    public boolean contains(Object x)
    {...}
}
```

Sentinel node never deleted
Hand-over-Hand Set

adding an entry

removing an entry

Fine-grain locking. How many locks to acquire?
Hand-over-Hand Set

adding c (acquire lock for successor before releasing lock for predecessor):
Hand-over-Hand Set
adding c (acquire lock for successor before releasing lock for predecessor):
Hand-over-Hand Set

removing b (acquire lock for successor before releasing lock for predecessor):
Hand-over-Hand Set

- Can we acquire locks only when reaching the modification place?
- adding entry c: advance until reaching (b, d) and then lock
Hand-over-Hand Set
The shared state consists of an array.

4.1 Enqueue Methods With Non-Fixed Linearization Points

(AbsQ)

all queue implementations of which we are aware that are not forward-simulated by forward-simulates many more queue implementations. In fact, force refining implementations to eagerly pick among linearizations of their enqueues, it maintains a partial order of enqueues, rather than a linear sequence. Since even though they refine Wing [18], denoted linearization points, or linearization point actions.

op

erations, and circles depict call, return, and lin-

Fig. 2.

void enq(int x) {
    i = back++; items[i] = x;
}

int deq() {
    while (1) {
        range = back - 1;
        for (int i = 0; i <= range; i++) {
            x = swap(items[i], null);
            if (x != null) return x;
        }
    }
}

assert ret(k) := v

assert loc:

assert present(k) := v

assert pending(k) := v

assert present(k1) := v

assert pending(k1) := v

assert before(k1, k) := v

assert ret(deq, v, k) := v

assert inv(deq, k) := v

assert ret(enq, k) := v

assert lin(deq, v, k) := v

assert lin(enq, k) := v
Complexity of Testing Linearizability

Theorem [Gibbons.et.al.'97]
Checking linearizability for a fixed execution is NP-hard
Checking Linearizability: Complexity (finite-state implementations)

**Bounded Nb. of Threads:**
- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

**Unbounded Nb. of Threads:**
- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]


**Bouajjani et al., 2013:** Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

**Hamza 2015:** Jad Hamza: On the Complexity of Linearizability. NETYS 2015
Checking Linearizability: Complexity (finite-state implementations)

**Bounded Nb. of Threads:**
- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

**Unbounded Nb. of Threads:**
- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]
Concurrent Languages

- Concurrent language = ($\Sigma$, $D$)
  - $\Sigma$ an alphabet
  - $D \subseteq \Sigma \times \Sigma$

(Mazurkiewicz traces - $D$ is symmetric)

- $a$ and $b$ are called independent when $(a, b) \notin D$
- $\Rightarrow_D$ a relation that permutes independent symbols:
  - for all $(a, b) \notin D$, $\sigma \ ab \ \sigma' \Rightarrow_D \ \sigma' \ ba \ \sigma$ (and trans. closure)

- $cl_D(L) =$ all strings $\sigma'$ such that $\sigma' \Rightarrow_D \sigma$ for some $\sigma \in L$
- Ex: $\Sigma = \{a, b\}$, $L = (ab)^*$, $D = \emptyset$ and $D = \{(b, a)\}$
Specifications, Implementations

- Specification = a language over an alphabet containing symbols $p: m(a) \Rightarrow b$
- Example: bounded-value register, bounded size queue
- Implementation = a language over an alphabet containing symbols $p: \text{call } m(a)$ and $p: \text{ret } m(a) \Rightarrow b$ where returns “match” previous calls

- $\Sigma_p = \left( \Sigma_{\text{call}}(p) \cup \Sigma_{\text{ret}}(p) \right)$
- $\Sigma = \bigcup_p \Sigma_p$
Example: Treiber Stack

class Node {
    Node tl;
    int val;
}

class NodePtr {
    Node val;
}

TOP;

void push(int e) {
    Node y, n;
y = new();
y->val = e;
    while(true) {
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {
    Node y,z;
    while(true) {
        y = TOP->val;
        if (y==0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}

What is the specification?
Defining Linearizability

- \( \text{lin} = \bigcup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{\text{ret}} \times \Sigma_{\text{call}}) \)
- \( \text{Spec}^* = \) replacing \( p:m(a) \Rightarrow b \) with call/ret actions
- an execution \( \sigma \) is **linearizable** iff \( \sigma \in \text{cl}_{\text{lin}}(\text{Spec}^*) \)
- \( \text{Impl} \) is linearizable iff \( \text{Impl} \subseteq \text{cl}_{\text{lin}}(\text{Spec}^*) \)
- this inclusion check is undecidable in general (for regular languages)
Defining Linearizability

• Linearizability:
  • an execution $\sigma$ is linearizable iff there exists a sequence $\tau$ that contains $\sigma$ and linearization points (symbols $p:m(a)\Rightarrow b$) such that:
    • every projection over “actions” of the same process is “sequential”
    • the projection over linearization point actions is included in the specification
Defining Linearizability

- \( \text{lin} = \bigcup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{\text{ret}} \times \Sigma_{\text{call}}) \)
- \( \text{Spec}^* = \) replacing \( p:m(a) \Rightarrow b \) with call/ret actions
- an execution \( \sigma \) is \textit{linearizable} iff \( \sigma \in \text{cl}_{\text{lin}}(\text{Spec}^*) \)
- \( \text{Impl} \) is linearizable iff \( \text{Impl} \subseteq \text{cl}_{\text{lin}}(\text{Spec}^*) \)
  - this inclusion check is undecidable in general (for regular languages)
- \( \text{cl}_{\text{lin}}(\text{Spec}^*) = (\parallel_p \text{L}_{\text{lin-points}}(p) \parallel \text{Spec}) \downarrow (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}}) \)
Problem 2 (Letter Insertion). Input: A set of insertable letters $A = \{a_1, \ldots, a_l\}$. An NFA $N$ over an alphabet $\Gamma \cup A$.

Question: For all words $w \in \Gamma^*$, does there exist a decomposition $w = w_0 \cdots w_l$, and a permutation $p$ of $\{1, \ldots, l\}$, such that $w_0 a_{p[1]} w_1 \ldots a_{p[l]} w_l$ is accepted by $N$?

Reducing Letter Insertion to Linearizability:

1. there exists a word $w$ in $\Gamma^*$, such that there is no way to insert the letters from $A$ in order to obtain a word accepted by $N$
2. there exists an execution of $Lib$ with $k$ threads which is not linearizable w.r.t. $S_N$

$k = l + 2$
EXPSPACE-hardness

Define $k$, the number of threads, to be $l + 2$.
We will define a library $Lib$ composed of

- methods $M_1, \ldots, M_l$, one for each letter of $A$
- methods $M_\gamma$, one for each letter of $\Gamma$
- a method $M_{\text{Tick}}$.

The specification $S_N$ is defined as the set of words $w$ over the alphabet
\[
\{M_1, \ldots, M_l\} \cup \{M_{\text{Tick}}\} \cup \{M_\gamma | \gamma \in \Gamma\}
\]
such that one the following condition holds:

- $w$ contains 0 letter $M_{\text{Tick}}$, or more than 1, or
- for a letter $M_i$, $i \in \{1, \ldots, l\}$, $w$ contains 0 such letter, or more than 1, or
- when projecting over the letters $M_\gamma$, $\gamma \in \Gamma$ and $M_i$, $i \in \{1, \ldots, l\}$, $w$ is in $N_M$, where $N_M$ is $N$ where each letter $\gamma$ is replaced by the letter $M_\gamma$, and where each letter $a_i$ is replaced by the letter $M_i$. 

---

**Fig. 4.** Description of $M_\gamma$, $\gamma \in \Gamma$

**Fig. 5.** Description of $M_1, \ldots, M_l$

**Fig. 6.** Description of $M_{\text{Tick}}$
EXPSPACE-hardness

Fig. 7. Non-linearizable execution corresponding to a word $\gamma_1 \ldots \gamma_m$ in which we cannot insert the letters from $A = \{a_1, \ldots, a_l\}$ to make it accepted by $N$. The points represent steps in the automata.
Checking Linearizability: Complexity
(finite-state implementations)

Bounded Nb. of Threads:
• EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:
• Undecidable [Bouajjani et al., 2013]
• Decidable with “fixed linearization points” [Bouajjani et al. 2013]
Undecidability

• Reduction from reachability in counter machines

• Given a counter machine A, we construct a library $L_A$ and a specification $S_A$ such that $L_A$ is not linearizable w.r.t. $S_A$ iff A reaches the target state

• $L_A =$ transition methods $T[t]$, increments $I[c_i]$, decrements $D[c_i]$ and zero-tests $Z[c_i]

• $L_A$ allows only valid sequences of transitions

• $S_A$ allows executions which don’t reach the target state, or which erroneously pass some zero-test

  – it doesn’t contain $M[q_f]$,  
  – it ends in $M[q_f]$ and it contains a prefix of the form

    \[
    (M_{\text{inc}}[i] M_{\text{dec}}[i])^*(M_{\text{inc}}[i]^+ + M_{\text{dec}}[i]^+)M_{\text{zero}}[i]
    \]

  – it ends in $M_f$ and it contains a subword of the form

    \[
    M_{\text{zero}}[i] (M_{\text{inc}}[i] M_{\text{dec}}[i])^*(M_{\text{inc}}[i]^+ + M_{\text{dec}}[i]^+)M_{\text{zero}}[i].
    \]
Undecidability

1. A sequence $t_1 t_2 \ldots t_i$ of $A$-transitions is modeled by a pairwise-overlapping sequence of $T[t_1] \cdot T[t_2] \cdot \ldots \cdot T[t_i]$ operations.
2. Each $T[t]$-operation has a corresponding $I[c_i]$, $D[c_i]$, or $Z[c_i]$ operation, depending on whether $t$ is, resp., an increment, decrement, or zero-test transition with counter $c_i$.
3. Each $I[c_i]$ operation has a corresponding $D[c_i]$ operation.
4. For each counter $c_i$, all $I[c_i]$ and $D[c_i]$ between $Z[c_i]$ operations overlap.
5. For each counter $c_i$, no $I[c_i]$ nor $D[c_i]$ operations overlap with a $Z[c_i]$ operation.
6. The number of $I[c_i]$ operations between two $Z[c_i]$ operations matches the number of $D[c_i]$ operations.
Undecidability

- a T/T signal between T[*] operations
- for each counter c, a T/I, T/D, T/Z between T[*] operations and, resp., I[c_i], D[c_i] and Z[c_i] operations
- an I/D signal between I[c_i] and D[c_i] operations
- a T/C signal between T[t] operations and I[c_i], D[c_i] operations, for zero-testing transitions t
Undecidability

Fig. 6. The \( L \)-A simulation of an \( A \)-execution with two increments followed by two decrements and a zero-test of counter \( c_1 \). Operations are drawn as horizontal lines containing rendezvous actions drawn as circles. Matching rendezvous actions are connected by dotted lines labeled by rendezvous type. Time advances to the right.

\( T/I, T/D, \) and \( T/Z \) rendezvousing ensures Property 2, \( I/D \) rendezvousing ensures Property 3, and \( T/C \) rendezvousing ensures Property 4. Note that even in the case where not all pending \( I[c_1] \) and \( D[c_1] \) operations perform \( T/C \) rendezvous with a concurrent \( T[t] \) operation, where \( t \) is a zero-test transition, at the very least, they overlap with all other pending \( I[c_1] \) and \( D[c_1] \) operations having performed \( T/I, \) resp., \( T/D, \) rendezvous since the last \( Z[c_1] \) operation.

The trickier part of our proof is indeed ensuring Properties 5 and 6. There we leverage Property 4: when all \( I[c_1] \) and \( D[c_1] \) operations between two \( Z[c_1] \) operations overlap, every permutation of them, including those alternating between \( I[c_1] \) and \( D[c_1] \) operations, is strict, i.e., is permitted by the definition of linearizability. Our specification \( S_A \) takes advantage of this in order to match the unbounded number of \( I[c_1] \) and \( D[c_1] \) operations using only bounded memory.

Lemma 5. The specification \( S_A \) accepting all sequences which either do not end with a transition to the target state, or in which the number of alternating \( I[c_1] \) and \( D[c_1] \) operations between two \( Z[c_1] \) operations are unequal, is regular.

Lemma 5 gives a way to ensure Properties 5 and 6, since any trace which is \( S_A \)-linearizable either does not encode an execution to \( A \)'s target state, or respects Property 5 while violating Property 6—i.e., the number of increments and decrements between zero-tests does not match—or violates Property 5: in the latter case, where some \( I[c_1] \) or \( D[c_1] \) operation \( \square_1 \) overlaps with a \( Z[c_1] \) operation \( \square_2 \), \( \square_1 \) can always be commuted over \( \square_2 \) to ensure that the number of \( I[c_1] \) and \( D[c_1] \) operations does not match in some interval between \( Z[c_1] \) operations. Thus any trace which is not \( S_A \)-linearizable must respect both Properties 5 and 6. It follows that any trace of \( L_A \) which is not \( S_A \)-linearizable guarantees Properties 1–6, and ultimately corresponds to a valid execution of \( A \), and visa versa, thus reducing counter machine state-reachability to \( S_A \)-linearizability.

Theorem 3. The linearizability problem for unbounded concurrent systems with regular specifications is undecidable.
Undecidability

1 var \( q \in Q \): \( T \)
2 var \( \text{req}[U] \): \( T \)
3 var \( \text{ack}[U] \): \( T \)
4 var \( \text{dec}[i \in \mathbb{N} : i < d] \): \( T \)
5 var \( \text{zero}[i \in \mathbb{N} : i < d] \): \( \mathbb{B} \)

6 // for each transition \( \langle q, n, q' \rangle \)
7 method \( M[q, n, q']() \)
8     atomic
9         wait(\( q \));
10        signal(req[n]);
11    atomic
12        wait(ack[n]);
13        signal(q');
14    return ()
15
16 // for each transition \( \langle q, i, q' \rangle \)
17 method \( M[q, i, q']() \)
18     atomic
19         wait(\( q \));
20        zero[i] := true;
21    atomic
22        if !zero[i] then
23            signal(q');
24    return ()
25
26 // for each final state \( q_f \)
27 method \( M[q_f]() \)
28     wait(q_f);
29     return
30
31 method \( M_{\text{inc}}[i]() \)
32     atomic
33     if !zero[i] then
34         wait(req[u_i]);
35         signal(ack[u_i]);
36         signal(dec[i])
37         assume zero[i];
38         return ()
39
40 method \( M_{\text{dec}}[i]() \)
41     atomic
42     if !zero[i] then
43         wait(dec[i]);
44     atomic
45         wait(req[-u_i]);
46         signal(ack[-u_i]);
47         assume zero[i];
48         return ()
49
50 method \( M_{\text{zero}}[i]() \)
51     atomic
52     if zero[i] then
53         zero[i] := false;
54     return ()
Checking Linearizability: Complexity (finite-state implementations)

**Bounded Nb. of Threads:**
- EXSPACE-complete [Alur et al., 1996, Hamza 2015]

**Unbounded Nb. of Threads:**
- Undecidable [Bouajjani et al., 2013]
- Decidable with “fixed linearization points” [Bouajjani et al. 2013]
Libraries

A *method* is a finite automaton $M = \langle Q, \Sigma, I, F, \rightarrow \rangle$ with labeled transitions
$\langle m_1, v_1 \rangle \xrightarrow{a} \langle m_2, v_2 \rangle$ between method-local states $m_1, m_2 \in Q$ paired with
finite-domain shared-state valuations $v_1, v_2 \in V$. The initial and final states
$I, F \subseteq Q$ represent the method-local states passed to, and returned from, $M$.

A *client* of a library $L$ is a finite automaton $C = \langle Q, \Sigma, \ell_0, \rightarrow \rangle$ with initial
state $\ell_0 \in Q$ and transitions $\rightarrow \subseteq Q \times \Sigma \times Q$ labeled by the alphabet $\Sigma = \{ M(m_0, m_f) : M \in L, m_0, m_f \in Q_M \}$ of library method calls

*most general client* $C^* = \langle Q, \Sigma, \ell_0, \rightarrow \rangle$ of a library $L$ nondeterministically calls
$L$’s methods in any order: $Q = \{ \ell_0 \}$ and $\rightarrow = Q \times \Sigma \times Q$. 
Example

class Node {
    Node tl;
    int val;
}
class NodePtr {
    Node val;
    int val;
}

void push(int e) {
    Node y, n;
    y = new();
    y->val = e;
    while(true) {
        n = TOP->val;
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {
    Node y, z;
    while(true) {
        y = TOP->val;
        if (y==0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}
Libraries

A configuration $c = \langle v, u \rangle$ of $L[C]$ is a shared memory valuation $v \in V$, along with a map $u$ mapping each thread $t \in \mathbb{N}$ to a tuple $u(t) = \langle \ell, m_0, m \rangle$, composed of a client-local state $\ell \in Q_C$, along with initial and current method states $m_0, m \in Q_L \cup \{\bot\}$; $m_0 = m = \bot$ when thread $t$ is not executing a library

\[
\begin{align*}
\text{INTERNAL} & \quad u_1(t) = \langle \ell, m_0, m_1 \rangle \\
& \quad \langle m_1, v_1 \rangle \xrightarrow{a} \langle m_2, v_2 \rangle \\
& \quad u_2 = u_1(t \mapsto \langle \ell, m_0, m_2 \rangle) \\
& \quad \langle v, u_1 \rangle \xrightarrow{\langle a, t \rangle} \langle v_2, u_2 \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{CALL} & \quad u_1(t) = \langle \ell_1, \bot, \bot \rangle \\
& \quad m_0 \in I_M \\
& \quad \ell_1 \xrightarrow{M(m_0,m_f)} \ell_2 \\
& \quad u_2 = u_1(t \mapsto \langle \ell_1, m_0, m_0 \rangle) \\
& \quad \langle v, u_1 \rangle \xrightarrow{\text{call}(M,m_0,t)} \langle v, u_2 \rangle \\
\end{align*}
\]

\[
\begin{align*}
\text{RETURN} & \quad u_1(t) = \langle \ell_1, m_0, m_f \rangle \\
& \quad m_f \in F_M \\
& \quad \ell_1 \xrightarrow{M(m_0,m_f)} \ell_2 \\
& \quad u_2 = u_1(t \mapsto \langle \ell_2, \bot, \bot \rangle) \\
& \quad \langle v, u_1 \rangle \xrightarrow{\text{ret}(M,m_f,t)} \langle v, u_2 \rangle \\
\end{align*}
\]

**Fig. 1.** The transition relation $\rightarrow_{L[C]}$ for the library-client composition $L[C]$. 
VASS model

We associate to each concurrent system $L[C]$ a *canonical VASS*, denoted $\mathcal{A}_{L[C]}$, whose states are the set of shared-memory valuations, and whose vector components count the number of threads in each thread-local state; a transition of $\mathcal{A}_{L[C]}$ from $\langle v_1, n_1 \rangle$ to $\langle v_2, n_2 \rangle$ updates the shared-memory valuation from $v_1$ to $v_2$ and the local state of some thread $t$ from $u_1(t)$ to $u_2(t)$ by decrementing the $u_1(t)$-component of $n_1$, and incrementing the $u_2(t)$-component, to derive $n_2$. 
Specifications

A specification $S$ of a library $L$ is a language over the specification alphabet

$$\Sigma_S \overset{\text{def}}{=} \{ M[m_0, m_f] : M \in L, m_0, m_f \in Q_M \}.$$ 

Definition 2 (Linearizability [20]). A trace $\tau$ is $S$-linearizable when there exists a completion\(^4\) $\pi$ of a strict, serial permutation of $\tau$ such that $(\pi \mid S) \in S$. 

---

\(^4\): A serial permutation of a trace is a sequence of actions that respects the order in which actions are executed.
Specifications

The pending closure of a specification \( S \), denoted \( \overline{S} \) is the set of \( S \)-images of serial sequences which have completions whose \( S \)-images are in \( S \):

\[
\overline{S} \overset{\text{def}}{=} \{(\sigma \mid S) \in \Sigma_S^* : \exists \sigma' \in \Sigma_S^*. (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.
\]
Specifications

The pending closure of a specification $S$, denoted $\overline{S}$ is the set of $S$-images of serial sequences which have completions whose $S$-images are in $S$:

$$\overline{S} \overset{\text{def}}{=} \{ (\sigma \mid S) \in \overline{\Sigma}^* : \exists \sigma' \in \Sigma^*. (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma \}.$$
Specifications

The pending closure of a specification $S$, denoted $\overline{S}$ is the set of $S$-images of serial sequences which have completions whose $S$-images are in $S$:

$$\overline{S} \stackrel{\text{def}}{=} \{(\sigma \mid S) \in \Sigma_S^* : \exists \sigma' \in \Sigma_S^*. (\sigma' \mid S) \in S \text{ and } \sigma' \text{ is a completion of } \sigma\}.$$  

![Fig. 2.](image1)

![Fig. 3.](image2)

**Lemma 1.** The pending closure $\overline{S}$ of a regular specification $S$ is regular.

**Lemma 2.** A trace $\tau$ is $S$-linearizable if and only if there exists a strict, serial permutation $\pi$ of $\tau$ such that $(\pi \mid S) \in \overline{S}$. 

---

*Note: The images referenced as `image1` and `image2` are not provided in the text and would need to be included for a complete understanding.*
Read-only operations

Given a method $M$ of a library $L$ and $m_0, m_f \in Q_M$, an $M[m_0, m_f]$-operation $\theta$ is read-only for a specification $S$ if and only if for all $w_1, w_2, w_3 \in \Sigma^*_S$,

1. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ then $w_1 \cdot M[m_0, m_f]^k \cdot w_2 \in S$ for all $k \geq 0$, and
2. If $w_1 \cdot M[m_0, m_f] \cdot w_2 \in S$ and $w_1 \cdot w_3 \in S$ then $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$. 

\begin{tikzpicture}
  \node[state, initial] (q_e) {$q_e$};
  \node[state] (q_a) [right of=q_e] {$q_a$};
  \node[state] (q_a_a) [right of=q_a] {$q_{a,a}$};
  \draw[->] (q_e) edge node {push[$a$, true]} (q_a);
  \draw[->] (q_a) edge node {push[$a$, true]} (q_a_a);
  \draw[->] (q_e) edge node {pop[\cdot, false]} (q_a);
  \draw[loop above] (q_e) edge node {pop[\cdot, true]} (q_e);
  \draw[loop above] (q_a) edge node {pop[\cdot, true]} (q_a);
  \draw[loop above] (q_a_a) edge node {pop[\cdot, true]} (q_a_a);
\end{tikzpicture}
Linearization points

The control graph $G_M = \langle Q_M, E \rangle$ is the quotient of a method $M$’s transition system by shared-state valuations $V$: $\langle m_1, a, m_2 \rangle \in E$ iff $\langle m_1, v_1 \rangle \leftarrow^a_M \langle m_2, v_2 \rangle$ for some $v_1, v_2 \in V$. A function $\text{LP}: L \to \wp(\Sigma_L)$ is called a linearization-point mapping when for each $M \in L$:

1. each symbol $a \in \text{LP}(M)$ labels at most one transition of $M$,
2. any directed path in $G_M$ contains at most one symbol of $\text{LP}(M)$, and
3. all directed paths in $G_M$ containing $a \in \text{LP}(M)$ reach the same $m_a \in F_M$.

An action $\langle a, i \rangle$ of an $M$-operation is called a linearization point when $a \in \text{LP}(M)$, and operations containing linearization points are said to be effectuated; $\text{LP}(\theta)$ denotes the unique linearization point of an effectuated operation $\theta$. A read-points mapping $\text{RP}: \Theta \to \mathbb{N}$ for an action sequence $\sigma$ with operations $\Theta$ maps each read-only operation $\theta$ to the index $\text{RP}(\theta)$ of an internal $\theta$-action in $\sigma$. 
Fixed Linearization Points

- **Fixed** linearization points: the linearization point is fixed to a particular statement in the code

```java
class Node {
    Node tl;
    int val;
}
class NodePtr {
    Node val;
} TOP

void push(int e) {
    Node y, n;
    y = new();
    y->val = e;
    while (true) {
        y->tl = n;
        if (cas(TOP->val, n, y))
            break;
    }
}

int pop() {
    Node y, z;
    while (true) {
        y = TOP->val;
        if (y == 0) return EMPTY;
        z = y->tl;
        if (cas(TOP->val, y, z))
            break;
    }
    return y->val;
}
```
Exercises (1)

- Does the Herlihy & Wing queue admit **fixed** linearization points?

```c
void enq(int x) {
    i = back++; items[i] = x;
}
int deq() {
    while (1) {
        range = back - 1;
        for (int i = 0; i <= range; i++) {
            x = swap(items[i], null);
            if (x != null) return x;
        }
    }
}
```
Static linearizability

Definition 4. A trace $\tau$ is $\langle S, \text{LP} \rangle$-linearizable when $\tau$ is effectuated, and there exists a read-points mapping $\text{RP}$ of $\tau$, along with an effect-preserving completion $\pi$ of a strict, point-preserving, and serial permutation of $\tau$ such that $(\pi \mid S) \in S$.

Definition 5 (Static Linearizability). The system $L[C]$ is $S$-static linearizable when $L[C]$ is $\langle S, \text{LP} \rangle$-linearizable for some mapping $\text{LP}$.
Checking Static Linerizability

- $A_S$ = a deterministic automaton recognizing the Specification
- we define a monitor to be composed with $L[C]$ that simulates the Specification
  - methods have a new local variable RO which is initially $\emptyset$ (records return values of read-only operations)
  - if $mf \in RO$ in an invocation of $M$, then $M[m0,mf]$ is read-only and a state of $A_S$ in which $M[m0,mf]$ is enabled has been observed
- $L[C]$ executes a linearization point $\Rightarrow$ the state of the Specification is advanced to the $M[m0,mf]$ successor ($m0$ is the initial state of the current operation and $mf$ is the unique final state reachable from this lin. point)
- $L[C]$ executes an internal action from an $M[m0,*]$ operation $\Rightarrow$ RO is enriched with every $mf$ such that $M[m0,mf]$ is read-only and enabled in the current specification state
- $L[C]$ executes the return of an $M[m0,mf]$ read-only operation $\Rightarrow$ if $mf \notin RO$ then the monitor goes to an error state
EXPSPACE-hardness

- Reduce control state reachability in VASS (which is EXPSPACE-complete) to static linearizability
  - Use the library from the undecidability proof without the zero-test method (the specification excludes only executions not reaching the target state)
Checking Linearizability: Complexity (finite-state implementations)

Bounded Nb. of Threads:

• EXSPACE-complete [Alur et al., 1996, Hamza 2015]

Unbounded Nb. of Threads:

• Undecidable [Bouajjani et al., 2013]

• Decidable with “fixed linearization points” [Bouajjani et al. 2013]


Bouajjani et al., 2013: Ahmed Bouajjani, Michael Emmi, Constantin Enea, Jad Hamza: Verifying Concurrent Programs against Sequential Specifications. ESOP 2013

Hamza 2015: Jad Hamza: On the Complexity of Linearizability. NETYS 2015