Checking Linearizability: Theoretical Limits

Constantin Enea
Ecole Polytechnique
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)

- \(\text{cl}_D(L) = \text{all strings} \ \sigma' \ \text{such that} \ \sigma' \Rightarrow_D \ \sigma \ \text{for some} \ \sigma \in L\)

- 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

• \( \Sigma_p = ( \Sigma_{\text{call}}(p) \cup \Sigma_{\text{ret}}(p) ) \)

• \( \Sigma = \bigcup \Sigma_p \)

• the projection of every sequence in the implementation over \( \Sigma_p \) must belong to a language \( L(p) \) where there is a return between every two calls
Example: Treiber Stack

```java
class Node {
    Node tl;
    int 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;
}

class NodePtr {
    Node val;
}

What is the specification?
Defining Linearizability

- \( \text{lin} = \cup_p (\Sigma_p \times \Sigma_p) \cup (\Sigma_{\text{ret}} \times \Sigma_{\text{call}}) \)
- \( \text{Spec}^* = \) 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

• lin = \( \cup_p ( \Sigma_p \times \Sigma_p ) \cup (\Sigma_{ret} \times \Sigma_{call}) \)

• Spec* = replacing \( p:m(a)\Rightarrow b \) with call/ret actions

• an execution \( \sigma \) is **linearizable** iff \( \sigma \in cl_{lin}(Spec^*) \)

• Impl is linearizable iff \( Impl \subseteq cl_{lin}(Spec^*) \)

• this inclusion check is undecidable in general (for regular languages)

• \( cl_{lin}(Spec^*) = ( \|_p L_{lin\_points}(p) \| Spec ) \downarrow ( \Sigma_{call} \cup \Sigma_{ret} ) \)
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 $w_0 a_{p[1]} w_1 \cdots a_{p[l]} w_l$ 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. $S_N$

$k = l+2$
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 $M_{Tick}$.

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

- $w$ contains 0 letter $M_{Tick}$, 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 $M_i$. 
EXPSPACE-hardness

Since $N$ is an NFA, $S_N$ is also an NFA. Moreover, its size is polynomial in the size of $N$. We now show the following equivalence:

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. $S_N$.

Let $w \in \Gamma^*$ such that there is no way to insert the letters from $A = \{a_1, \ldots, a_l\}$ to make it accepted by $N$. We construct an execution following Fig 7, which is indeed a valid execution.

**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 \( T[t_1] \cdot T[t_2] \cdots T[t_i] \) 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\([c_i]\) and D\([c_i]\) 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\([c_i]\) and D\([c_i]\) operations having performed T/I, resp., T/D rendezvous since the last Z\([c_i]\) operation.

The trickier part of our proof is indeed ensuring Properties 5 and 6. There we leverage Property 4: when all I\([c_i]\) and D\([c_i]\) operations between two Z\([c_i]\) operations overlap, every permutation of them, including those alternating between I\([c_i]\) and D\([c_i]\) 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\([c_i]\) and D\([c_i]\) 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\([c_i]\) and D\([c_i]\) operations between two Z\([c_i]\) 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\([c_i]\) or D\([c_i]\) operation \( \checkmark_1 \) overlaps with an Z\([c_i]\) operation \( \checkmark_2 \), \( \checkmark_1 \) can always be commuted over \( \checkmark_2 \) to ensure that the number of I\([c_i]\) and D\([c_i]\) operations does not match in some interval between Z\([c_i]\) 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

```plaintext
1 var q ∈ Q: T
2 var req[U]: T
3 var ack[U]: T
4 var dec[i ∈ N: i < d]: T
5 var zero[i ∈ N: i ∈ d]: B

7 // for each transition ⟨q, n, q'⟩
8 method M[q, n, q'] ()
   atomic
   wait(q);
   signal(req[n]);
   atomic
   wait(ack[n]);
   signal(q');
   return ()

17 // for each transition ⟨q, i, q'⟩
18 method M[q, i, q'] ()
   atomic
   wait(q);
   zero[i] := true;
   atomic
   if !zero[i] then
      signal(q');
   return ()

27 // for each final state qf
28 method M[qf] ()
   wait(qf);
   return

31 method M_inc[i] ()
   atomic
   if !zero[i] then
      wait(req[u_i]);
      signal(ack[u_i]);
      signal(dec[i])
      assume zero[i];
      return ()

40 method M_dec[i] ()
   atomic
   if !zero[i] then
      wait(dec[i]);
      atomic
      wait(req[-u_i]);
      signal(ack[-u_i]);
      assume zero[i];
      return ()

49 method M_zero[i] ()
   atomic
   if zero[i] then
      zero[i] := false;
   return ()
```

Fig. 8. The library // for each transition ⟨q, n, q'⟩
Fig. 9. An increment transition // for each transition
Simulating runs of

Initially, all the binary semaphores except

should execute the operation

for all

T

for each transition

method calls as in Figure 9. An increment transition

successive calls to

method

that once a method is called it also executes its first atomic section, followed

by executions of

of unit vectors, and

if

then, when the semaphore

denotes the type of a binary semaphore).

wait

atomic

atomic

atomic

atomic

atomic

atomic

atomic

atomic
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]
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 $C = \langle Q, \Sigma, \ell_0, \leftarrow \rangle$ with initial state $\ell_0 \in Q$ and transitions $\leftarrow \subseteq Q \times \Sigma \times Q$ 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** $C^* = \langle Q, \Sigma, \ell_0, \leftarrow \rangle$ of a library $L$ nondeterministically calls $L$’s methods in any order: $Q = \{\ell_0\}$ and $\leftarrow = Q \times \Sigma \times Q$. 

**Libraries**
Example

class Node {
    Node tl;
    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 \( m_0, m \in Q_L \cup \{ \perp \} \); \( m_0 = m = \perp \) when thread \( t \) is not executing a library

\[
\begin{align*}
\text{INTERNAL} & \quad u_1(t) = \langle \ell, m_0, m_1 \rangle \\
& \quad \langle m_1, v_1 \rangle \xrightarrow{a} \langle m_2, v_2 \rangle \\
& \quad u_2 = u_1(t \mapsto \langle \ell, m_0, m_2 \rangle) \\
& \quad \langle v_1, u_1 \rangle \xrightarrow{\langle a, t \rangle} \langle v_2, u_2 \rangle \\
\text{CALL} & \quad u_1(t) = \langle \ell_1, \perp, \perp \rangle \\
& \quad m_0 \in I_M \\
& \quad \ell_1 \xrightarrow{M(m_0, m_f)} C \ell_2 \\
& \quad u_2 = u_1(t \mapsto \langle \ell_1, m_0, m_0 \rangle) \\
& \quad \langle v, u_1 \rangle \xrightarrow{\text{call}(M, m_0, t)} \langle v, u_2 \rangle \\
\text{RETURN} & \quad u_1(t) = \langle \ell_1, m_0, m_f \rangle \\
& \quad m_f \in F_M \\
& \quad \ell_1 \xrightarrow{M(m_0, m_f)} C \ell_2 \\
& \quad u_2 = u_1(t \mapsto \langle \ell_2, \perp, \perp \rangle) \\
& \quad \langle v, u_1 \rangle \xrightarrow{\text{ret}(M, m_f, t)} \langle v, u_2 \rangle \\
\end{align*}
\]

\textbf{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 \textit{canonical} VASS,\(^2\) 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 completion of a strict, serial permutation $\pi$ is a permutation $\pi'$ of actions of each operation, must be preserved in deriving that serial sequence from a permutation of actions in the original concurrent execution. Both notions are corresponding to some serial sequence, a "conflict relation," relating the individual actions of each operation, must be preserved in deriving that serial sequence from a permutation of actions in the original concurrent execution.
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 $S$:

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

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

**Fig. 2.** The sequential specification of two-element stacks containing the (abstract) value $a$, given as the language of a finite automaton, whose operation alphabet indicates both the argument and return values. **Fig. 3.** The pending closure of the stack specification from Figure 2.

**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 $(\pi \mid S) \in \overline{S}$. 
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 $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$. 

\[
\begin{array}{c}
q_e & \xrightarrow{\text{push}[a, \text{true}]} & q_a & \xrightarrow{\text{push}[a, \text{true}]} & q_{a,a} \\
\xrightarrow{\text{pop}[\cdot, \text{false}]} & & \xrightarrow{\text{pop}[\cdot, \text{true}]} & & \xrightarrow{\text{pop}[\cdot, \text{true}]} \\
\end{array}
\]
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 $\langle m_1, v_1 \rangle \xrightarrow{a}^M \langle m_2, v_2 \rangle$ for some $v_1, v_2 \in V$. A function $LP : L \rightarrow \wp(\Sigma_L)$ is called a linearization-point mapping when for each $M \in L$:

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

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

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;
}
```

Treiber Stack
Exercices (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

An action sequence $\sigma$ is called effectuated when every completed operation of $\sigma$ is either effectuated or read-only, and an effectuated completion $\sigma'$ of $\sigma$ is effect preserving when each effectuated operation of $\sigma$ also appears in $\sigma'$. Given a linearization-point mapping $\text{LP}$, and a read-points mapping $\text{RP}$ of an action sequence $\sigma$, we say a permutation $\pi$ of $\sigma$ is point preserving when every two operations of $\pi$ are ordered by their linearization/read points in $\sigma$.

**Definition 4.** A trace $\tau$ is $(S, \text{LP})$-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 $(S, \text{LP})$-linearizable for some mapping $\text{LP}$. 
Checking Static Linerizability

- $A_S = \text{a deterministic automaton recognizing the Specification}$
- we define a monitor to be composed with $L[C]$ that simulates the Specification
  - methods have a new local variable $\text{RO}$ which is initially $\emptyset$ (records return values of read-only operations)
  - if $mf \in \text{RO}$ in an invocation of $M$, then $M[m_0,mf]$ is read-only and a state of $A_S$ in which $M[m_0,mf]$ is enabled has been observed
- $L[C]$ executes a linearization point $\Rightarrow$ the state of the Specification is advanced to the $M[m_0,mf]$ successor ($m_0$ 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 $M[m_0,\ast]$ operation $\Rightarrow$ $\text{RO}$ is enriched with every $mf$ such that $M[m_0,mf]$ is read-only and enabled in the current specification state
- $L[C]$ executes the return of an $M[m_0,mf]$ read-only operation $\Rightarrow$ if $mf \notin \text{RO}$ 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
Reducing Linearizability to Reachability
Checking Lin. using “bad patterns”

- Reduce linearizability checking to reachability (EXPSPACE-complete):
  - Define (sequential) data-structure S using inductive rules
  - S is data independent and closed under projection
  - Characterize sequential executions of S using bad patterns
  - Characterize concurrent executions, linearizable w.r.t. S using bad patterns (one per rule)
  - Define a regular automaton $A_i$ for each bad pattern
  - Reduce “L is linearizable w.r.t. S” to: for all i, $L \cap A_i = \emptyset$
Histories = Posets of events

Thread 1
push(1)  pop ⇒ 2
pop ⇒ 1  push(2)  push(3)  pop ⇒ 3

Thread 2

happens-before partial order
Concurrent Queues

Linearizability $\equiv$ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

“Value v dequeued without being enqueued”

```
  deq: v
```

“Value v dequeued before being enqueued”

```
  deq: v    enq: v
```

“Value v dequeued twice”

```
  deq: v    deq: v
```

“Values dequeued in the wrong order”

```
  enq: v₁   enq: v₂   deq: v₂   deq: v₁
```
Concurrent Queues

Linearizability $\equiv$ Exclusion of bad patterns (assuming each value is enqueued at most once - sound under data independence)

“Value v dequeued without being enqueued”

```
  deq: v
```

“Value v dequeued before being enqueued”

```
  deq: v  enq: v
```

“Value v dequeued twice”

```
  deq: v  deq: v
```

“Values dequeued in the wrong order”

```
  enq: v₁  enq: v₂  deq: v₂  deq: v₁
```

“Dequeue wrongfully returns empty”

```
  deq: empty
```

[ICALP’15]
Concurrent Queues

**Linearizability** ≡ Exclusion of **bad patterns** (assuming each value is enqueued at most once - sound under data independence)

- "Value v dequeued without being enqueued"
  
  deq: v

- "Value v dequeued before being enqueued"
  
  deq: v  enq: v

- "Value v dequeued twice"
  
  deq: v  deq: v

- "Values dequeued in the wrong order"

  enq: v₁  enq: v₂  deq: v₂  deq: v₁

- "Dequeue wrongfully returns empty"

  deq: empty

[ICALP’15]
Concurrent Queues

Linearizability $\equiv$ Exclusion of bad patterns (assuming each value is enqueued at most once - sound under data independence)

“Value v dequeued without being enqueued”

\[
\begin{align*}
\text{enq: } v & \\
\text{deq: } v & \\
\end{align*}
\]

“Value v dequeued before being enqueued”

\[
\begin{align*}
\text{enq: } v & \\
\text{deq: } v & \\
\text{enq: } v & \\
\end{align*}
\]

“Value v dequeued twice”

\[
\begin{align*}
\text{enq: } v & \\
\text{deq: } v & \\
\text{deq: } v & \\
\end{align*}
\]

“Values dequeued in the wrong order”

\[
\begin{align*}
\text{enq: } v_1 & \\
\text{enq: } v_2 & \\
\text{deq: } v_2 & \\
\text{deq: } v_1 & \\
\end{align*}
\]

“Dequeue wrongfully returns empty”

\[
\begin{align*}
\text{enq: } v_1 & \\
\text{enq: } v_2 & \\
\text{deq: } v_2 & \\
\text{deq: } v_1 & \\
\text{enq: } v_n & \\
\text{deq: } v_{n-1} & \\
\text{deq: } v_n & \\
\text{deq: } \text{empty} & \\
\end{align*}
\]

[ICALP’15]
Concurrent Stacks

Linearizability $\equiv$ Exclusion of bad patterns (assuming each value is enqueued at most once, which is sound under data independence)

“Value v popped without being pushed”
“Value v popped before being pushed”
“Value v popped twice”
“Pop wrongfully returns empty”

“Pop doesn’t return the top of the stack”
Concurrent Stacks

Linearizability $\equiv$ Exclusion of bad patterns (assuming each value is enqueued at most once, which is sound under data independence)

“Value v popped without being pushed”
“Value v popped before being pushed”
“Value v popped twice”
“Pop wrongfully returns empty”

“Pop doesn’t return the top of the stack”
Concurrent Stacks

Linearizability ≡ Exclusion of bad patterns (assuming each value is enqueued at most once, which is sound under data independence)

“Value v popped without being pushed”
“Value v popped before being pushed”
“Value v popped twice”
“Pop wrongfully returns empty”
Checking Lin. using “bad patterns”

- Reduce linearizability checking to reachability (EXPSPACE-complete):
  - Define (sequential) data-structure $S$ using inductive rules
  - $S$ is data independent and closed under projection
  - Characterize sequential executions of $S$ using bad patterns
  - Characterize concurrent executions, linearizable w.r.t. $S$ using bad patterns (one per rule)
  - Define a regular automaton $A_i$ for each bad pattern
  - Reduce “$L$ is linearizable w.r.t. $S$” to: $\forall i, L \cap A_i = \emptyset$
Inductive definition of the Register

\[ R_{wr} : u \in R \iff \text{Write}_x \cdot (\text{Read}_x)^* \cdot u \in R \]

- including the empty sequence
Inductive definition of the Queue

Two rules to build the sequences belonging to the Queue such as

\[ Enq_4 Enq_3 Deq_4 Deq_3 EMP Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]

\[ R_{Enq} : \quad u \in Q \land u \in Enq^* \Rightarrow u \cdot Enq_x \in Q \]
\[ R_{EnqDeq} : \quad u \cdot v \in Q \land u \in Enq^* \Rightarrow Enq_x \cdot u \cdot Deq_x \cdot v \in Q \]
\[ R_{EMP} : \quad u \cdot v \in Q \land \text{no unmatched } Enq \text{ in } u \Rightarrow u \cdot EMP \cdot v \in Q \]

Derivation:

\[ \epsilon \in Q \]
\[ \rightarrow Enq_1 Deq_1 \in Q \]
\[ \rightarrow Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]
\[ \rightarrow Enq_3 Deq_3 Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]
\[ \rightarrow Enq_4 Enq_3 Deq_4 Deq_3 Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]
\[ \rightarrow Enq_4 Enq_3 Deq_4 Deq_3 EMP Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]
Inductive definition of the Stack

\[ R_{PushPop} : \ u \cdot v \in S \land \text{no unmatched } Push \text{ in } u, v \Rightarrow \text{Push}_x \cdot u \cdot \text{Pop}_x \cdot v \in S \]

\[ R_{Push} : \ u \cdot v \in S \land \text{no unmatched } Push \text{ in } u \Rightarrow u \cdot \text{Push}_x \cdot v \in S \]

\[ R_{EMP} : \ u \cdot v \in S \land \text{no unmatched } Push \text{ in } u \Rightarrow u \cdot \text{EMP} \cdot v \in S \]

Derivation for \( Push_1 \text{Push}_2 \text{Pop}_2 \text{Pop}_1 \text{EMP} \text{Push}_3 \text{Pop}_3 \in S \)

\[ \epsilon \in S \]

\[ \Rightarrow \text{Push}_3 \text{Pop}_3 \in S \]

\[ \Rightarrow \text{Push}_2 \text{Pop}_2 \text{Push}_3 \text{Pop}_3 \in S \]

\[ \Rightarrow \text{Push}_1 \text{Push}_2 \text{Pop}_2 \text{Pop}_1 \text{Push}_3 \text{Pop}_3 \in S \]

\[ \Rightarrow \text{Push}_1 \text{Push}_2 \text{Pop}_2 \text{Pop}_1 \text{EMP} \text{Push}_3 \text{Pop}_3 \in S \]
Data Independence

- Input methods = methods taking an argument
- A sequential execution $u$ is called \textit{differentiated} if for all input methods $m$ and every $x$, $u$ contains at most one invocation $m(x)$
- $S_\neq$ is the set of differentiated executions in $S$

A \textit{renaming} $r$ is a function from $\mathcal{D}$ to $\mathcal{D}$. Given a sequential execution (resp., execution or history) $u$, we denote by $r(u)$ the sequential execution (resp., execution or history) obtained from $u$ by replacing every data value $x$ by $r(x)$.

\textbf{Definition 6.} \textit{The set of sequential executions (resp., executions or histories) $S$ is data independent if:}

\begin{itemize}
  \item for all $u \in S$, there exists $u' \in S_\neq$, and a renaming $r$ such that $u = r(u')$,
  \item for all $u \in S$ and for all renaming $r$, $r(u) \in S$.
\end{itemize}

\textbf{Theorem:} A data-independent implementation $I$ is linearizable w.r.t. a data-independent specification $S$ iff $I_\neq$ is linearizable w.r.t. $S_\neq$
Closure under projection

**Projection**: Subsequence consistent with the values

If

\[ Enq_4 Enq_3 Deq_4 Deq_3 Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]

Then

\[ Enq_4 Deq_4 Enq_2 Enq_1 Deq_2 Deq_1 \in Q \]

**Lemma**

Any data structure defined in our framework is closed under projection

**Proof.**

The predicates used (\( u \in Enq^* \) and “no unmatched Enq in \( u \)”) are closed under projection
Characterization of sequential executions

We assume that the rules defining a data-structure are well-formed, that is:

• for all $u \in [S]$, there exists a unique rule, denoted by $\text{last}(u)$, that can be used as the last step to derive $u$, i.e., for every sequence of rules $R_{i_1}, \ldots, R_{i_n}$ leading to $u$, $R_{i_n} = \text{last}(u)$. For $u \notin [S]$, $\text{last}(u)$ is also defined but can be arbitrary, as there is no derivation for $u$.

• if $\text{last}(u) = R_i$, then for every permutation $u' \in [S]$ of a projection of $u$, $\text{last}(u') = R_j$ with $j \leq i$. If $u'$ is a permutation of $u$, then $\text{last}(u') = R_i$.

**Example 6.** For Queue, we define $\text{last}$ for a sequential execution $u$ as follows:

• if $u$ contains a $\text{DeqEmpty}$ operation, $\text{last}(u) = R_{\text{DeqEmpty}}$,

• else if $u$ contains a $\text{Deq}$ operation, $\text{last}(u) = R_{\text{EnqDeq}}$,

• else if $u$ contains only $\text{Enq}$'s, $\text{last}(u) = R_{\text{Enq}}$,

• else (if $u$ is empty), $\text{last}(u) = R_0$.

Since the conditions we use to define $\text{last}$ are closed under permutations, we get that for any permutation $u_2$ of $u$, $\text{last}(u) = \text{last}(u_2)$, and $\text{last}$ can be extended to histories. Therefore, the rules $R_0, R_{\text{EnqDeq}}, R_{\text{DeqEmpty}}$ are well-formed.
Characterization of sequential executions

- MS(R) = the set of sequences “matching” a rule R

Lemma 3. Let $S = R_1, \ldots, R_n$ be a data-structure and $u$ be a differentiated sequential execution. Then,

$$u \in S \iff \text{proj}(u) \subseteq \bigcup_{i \in \{1, \ldots, n\}} \text{MS}(R_i)$$

**Lemma (Characterization of Queue Sequential Executions)**

- $w \in \mathcal{Q}$ iff every projection $w'$ of $w$ is either of the form
  - $\text{Enq}_x \cdot u \cdot \text{Deq}_x \cdot v$ (with $u \in \text{Enq}^*$) or
  - $u \cdot \text{EMP} \cdot v$ (with no unmatched Enq in $u$)
Characterization of concurrent executions

**Definition 7.** A data-structure $S = R_1, \ldots, R_n$ is said to be step-by-step linearizable if for any differentiated execution $e$, any $i \in \{1, \ldots, n\}$ and $x \in \mathbb{D}$, if $e$ is linearizable with respect to $\text{MS}(R_i)$ with witness $x$, we have:

$$e \setminus x \subseteq [R_1, \ldots, R_i] \implies e \subseteq [R_1, \ldots, R_i]$$

- the history linearizable $\text{MS}(R_{\text{EnqDeq}})$ with witness $d_1$
  - $\text{Enq}(d_1)$ is minimal among all operations and $\text{Deq}(d_1)$ minimal among all dequeue
- Excluding the operations on $d_1$, the history is linearizable w.r.t. $[R_{\text{Enq}}, R_{\text{EnqDeq}}]$, i.e., $\text{Enq}(d_2)$ $\text{Enq}(d_3)$ $\text{Deq}(d_2)$ $\text{Deq}(d_3)$
- The notion of step-by-step linearizable ensures that the history is linearizable w.r.t. Queue
Step-by-Step Lin. of Register

Lemma 9. Register is step-by-step linearizable.

Proof. Let $h$ be a differentiated history, and $u$ a sequential execution such that $h \subseteq u$ and such that $u$ matches the rule $R_{WR}$ with witness $x$. Let $a$ and $b_1, \ldots, b_s$ be respectively the Write and Read’s operations of $h$ corresponding to the witness.

Let $h' = h \setminus x$ and assume $h' \subseteq [R_0, R_{WR}]$. Let $u' \in [R_0, R_{WR}]$ such that $h' \subseteq u'$. Let $u_2 = a \cdot b_1 \cdot b_2 \cdots b_s \cdot u'$. By using rule $R_{WR}$ on $u'$, we have $u_2 \in [R_0, R_{WR}]$. Moreover, we prove that $h \subseteq u_2$ by contradiction. Assume that the total order imposed by $u_2$ doesn’t respect the happens-before relation of $h$. All three cases are not possible:

- the violation is between two $u'$ operations, contradicting $h' \subseteq u'$,
- the violation is between $a$ and another operation, i.e. there is an operation $o$ which happens before $a$ in $h$, contradicting $h \subseteq u$,
- the violation is between some $b_i$ and a $u'$ operation, i.e. there is an operation $o$ which happens before $b_i$ in $h$, contradicting $h \subseteq u$.

Thus, we have $h \subseteq u_2$ and $h \subseteq [R_0, R_{WR}]$, which ends the proof. \qed
Characterization of concurrent executions

Lemma 4. Let \( S \) be a data-structure with rules \( R_1, \ldots, R_n \). Let \( e \) be a differentiated execution. If \( S \) is step-by-step linearizable, we have (for any \( j \)):

\[
e \subseteq [R_1, \ldots, R_j] \iff \text{proj}(e) \subseteq \bigcup_{i \leq j} \text{MS}(R_i)
\]

Proof (\( \iff \)) By induction on the size of \( e \). We know \( e \in \text{proj}(e) \) so it can be linearized with respect to a sequential execution \( u \) matching some rule \( R_k \) \((k \leq j)\) with some witness \( x \). Let \( e' = e \setminus x \).

Since \( S \) is well-formed, we know that no projection of \( e \) can be linearized to a matching set \( \text{MS}(R_i) \) with \( i > k \), and in particular no projection of \( e' \). Thus, we deduce that \( \text{proj}(e') \subseteq \bigcup_{i \leq k} \text{MS}(R_i) \), and conclude by induction that \( e' \subseteq [R_1, \ldots, R_k] \).

We finally use the fact that \( S \) is step-by-step linearizable to deduce that \( e \subseteq [R_1, \ldots, R_k] \) and \( e \subseteq [R_1, \ldots, R_j] \) because \( k \leq j \).

Lemma

\( E \) is linearizable to \( Q \) iff every projection \( E' \) of \( E \) is linearizable to the form \( \text{Enq}_x \cdot u \cdot \text{Deq}_x \cdot v \) \((\text{with } u \in \text{Enq}^*)\) or to the form \( u \cdot \text{EMP} \cdot v \) \((\text{with no unmatched Enq in } u)\)
Characterization of concurrent executions

**Lemma 5.** Let $S$ be a data-structure with rules $R_1, \ldots, R_n$. Let $e$ be a differentiated execution. If $S$ is step-by-step linearizable, we have:

$$e \subseteq S \iff \forall e' \in \text{proj}(e). e' \subseteq \text{MS}(R) \text{ where } R = \text{last}(e')$$

$$e \not\subseteq S \iff \exists e' \in \text{proj}(e). e' \not\subseteq \text{MS}(R) \text{ (where } R = \text{last}(e'))$$

*E is non-linearizable wrt Queue iff it has a projection $E'$ of the form bad pattern 1, or bad pattern 2.*

**Bad Pattern 1 (rule $R_{EnqDeq}$):**

- $Enq_1$
- $Deq_2$

$Enq_1 < Enq_2$
$Deq_2 < Deq_1$

or $Deq_1$ before $Enq_1$
Lemma 5. Let $S$ be a data-structure with rules $R_1, \ldots, R_n$. Let $e$ be a differentiated execution. If $S$ is step-by-step linearizable, we have:

$$e \in S \iff \forall e' \in \text{proj}(e). e' \in \text{MS}(R) \text{ where } R = \text{last}(e')$$

$$e \notin S \iff \exists e' \in \text{proj}(e). e' \notin \text{MS}(R) \text{ (where } R = \text{last}(e'))$$

$E$ is non-linearizable wrt Queue iff it has a projection $E'$ of the form bad pattern 1, or bad pattern 2.
Characterization of concurrent executions

- define for each R, a finite state automaton A which recognizes (a subset of) the executions e which have a projection not linearizable w.r.t. MS(R)

**Definition 8.** A rule R is said to be co-regular if we can build an automaton A such that, for any data-independent implementation I, we have:

\[ I \cap A \neq \emptyset \iff \exists e \in I_{\pm}, e' \in \text{proj}(e). \text{last}(e') = R \land e' \notin MS(R) \]
Characterization of concurrent executions

- define for each R, a finite state automaton A which recognizes (a subset of) the executions e which have a projection not linearizable w.r.t. MS(R)

**Definition 8.** A rule R is said to be co-regular if we can build an automaton A such that, for any data-independent implementation I, we have:

\[ I \cap A \neq \emptyset \iff \exists e \in I_\#, e' \in \text{proj}(e). \text{last}(e') = R \land e' \notin \text{MS}(R) \]

we assume that all actions call Enq(1) occur at the beginning
Exercices (2)

We consider a sequential specification defined by the language $S = (a())^n (b())^m$ where all the invocations of $a()$ occur before invocations of $b()$.

1. Describe a reduction of checking linearizability w.r.t. the specification $S$ to a reachability problem. More precisely, describe a labeled transition system (monitor) that accepts exactly all the histories of a given implementation (sequences of call and return actions) that are not linearizable w.r.t. $S$. The synchronized product between a transition system representing an implementation and this monitor (where the synchronization actions are call and returns) reaches an accepting state of the monitor iff the implementation is not linearizable.
Exercises (3)

• What is the complexity of checking linearizability of a differentiated history of a concurrent queue?
Exercises (3)

• What is the complexity of checking linearizability of a differentiated history of a concurrent queue?

“Value v dequeued without being enqueued”

```
deq: v
```

“Value v dequeued before being enqueued”

```
deq: v  enq: v
```

“Value v dequeued twice”

```
deq: v  deq: v
```

“Values dequeued in the wrong order”

```
enq: v₁  enq: v₂  deq: v₂  deq: v₁
```

“Dequeue wrongfully returns empty”

```
enq: v₁  deq: v₁
  ↓
enq: v₂  deq: v₂
  ↓
  ...  deq: v_{n-1}
  ↓
  deq: vₙ
```