Checking Linearizability: Theoretical Limits

Constantin Enea

Ecole Polytechnique
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

\[ \exists \text{lin. } \text{rb} \subseteq \text{lin} \land \text{lin} \in \text{Queue ADT} \]
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;
}

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;
    }
}

class NodePtr {  
    Node val;
}

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

adding c (acquire lock for successor before releasing lock for predecessor):
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;
        }
    }
}
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
- \( \Sigma_p = ( \Sigma_{\text{call}}(p) \cup \Sigma_{\text{ret}}(p) ) \)
- \( \Sigma = \bigcup_p \Sigma_p \)
Example: Treiber Stack

class Node {
    Node tl;
    int val;
}

class NodePtr {
    Node val;
    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;
}

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 \( p:\text{m}(a)\Rightarrow b \) 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 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)

- \( \text{cl}_{\text{lin}}(\text{Spec}^*) = ( \|_p L_{\text{lin}\_\text{points}}(p) \| \text{Spec} ) \downarrow (\Sigma_{\text{call}} \cup \Sigma_{\text{ret}}) \)

Language of call - lin.point - ret triples for process \( p \)
**EXPSPACE-hardness**

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. $S_N$

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

![Diagram of $M_\gamma$, $\gamma \in \Gamma$](image4)

![Diagram of $M_1, \ldots, M_l$](image5)

![Diagram of $M_{\text{Tick}}$](image6)

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 $M_i$. 
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
- $L_A = \text{transition methods } T[t], \text{increments } I[c_i], \text{decrements } D[c_i] \text{ and zero-tests } Z[c_i]$
- $L_A$ 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
  $$(M_{inc}[i] M_{dec}[i])^*(M_{inc}[i]^+ + M_{dec}[i]^+) M_{zero}[i]$$
- it ends in $M_f$ and it contains a subword of the form
  $$M_{zero}[i](M_{inc}[i] M_{dec}[i])^*(M_{inc}[i]^+ + M_{dec}[i]^+) M_{zero}[i].$$
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

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’]()
9     atomic
10      wait(q);
11      signal(req[n]);
12     atomic
13      wait(ack[n]);
14      signal(q’);
15      return ()

17 // for each transition ⟨q, i, q’⟩
18 method M[q, i, q’]()
19     atomic
20      wait(q);
21      zero[i] := true;
22     atomic
23      if !zero[i] then
24          signal(q’);
25      return ()

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

31 method M_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 ()

40 method M_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 ()

50 method M_zero[i]()
51     atomic
52     if zero[i] then
53         zero[i] := false;
54     return ()
Undecidability

The linearizability problem for unbounded concurrent systems with regular specifications is undecidable.
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 \( \rightarrow = Q \times \Sigma \times Q \).
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;
    }
}

class NodePtr {
    Node val;
    } TOP;

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

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

\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 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 $\pi$ of a trace $\tau$ is a permutation of the actions of $\tau$.
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 $S$:

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

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

**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.
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 $S$:

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

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 $(\pi \mid S) \in \overline{S}$. 

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 $w_1 \cdot M[m_0, m_f] \cdot w_3 \in S$. 

Diagram:

\begin{center}
\begin{tikzpicture}[->,>=stealth',shorten >=1pt,auto,node distance=2.5cm,semithick]
  \node (q0) {$q_e$};
  \node (q1) [right of=q0] {$q_a$};
  \node (q2) [right of=q1] {$q_{a,a}$};

  \path
    (q0) edge [loop above] node {pop[\cdot, false]} (q0)
    (q0) edge node {push[a, true]} (q1)
    (q1) edge [bend right] node {pop[\cdot, true]} (q0)
    (q1) edge node {push[a, true]} (q2)
    (q2) edge [bend right] node {pop[\cdot, true]} (q1);
\end{tikzpicture}
\end{center}
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;
    } 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;
}
```

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 $LP$, and a read-points mapping $RP$ 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 $\langle S, LP \rangle$-linearizable when $\tau$ is effectuated, and there exists a read-points mapping $RP$ 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 $\langle S, LP \rangle$-linearizable for some mapping $LP$. 
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 \( M[m_0,mf] \) 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 \( M[m_0,*] \) operation \( \Rightarrow \) 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 \( mf \not\in RO \) 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