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, …
Observational Refinement

\[ \iff \]

Linearizability/ Refinement
Observational Refinement

Reference implementation

```c
class AtomicStack {
    cell* top;
    Lock l;

    void push (int v) {
        l.lock();
        top->next = malloc(sizeof *x);
        top = top->next;
        top->data = v;
        l.unlock();
    }

    int pop () {
        ...
    }
}
```

Efficient implementation

```c
class TreiberStack {
    cell* top;

    void push (int v) {
        cell* t;
        cell* x = malloc(sizeof *x);
        x->data = v;
        do {
            t = top;
            x->next = top;
            top->data = v;
        } while (!CAS(&top,t,x));
    }

    int pop () {
        ...
    }
}
```

For every **Client**, **Client x Impl** included in **Client x Spec**
Formalizing Libraries/Programs

We fix an arbitrary set $\Omega$ of operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$$C = \{m(v)_o : m \in \mathbb{M}, v \in \mathbb{V}, o \in \Omega\},$$

$$R = \{\text{ret}(v)_o : v \in \mathbb{V}, o \in \Omega\}$$

of call actions and return actions; each call action $m(v)_o$ combines a method $m \in \mathbb{M}$ and value $v \in \mathbb{V}$ with an operation identifier $o \in \Omega$. Operation identifiers are used to pair call and return actions.

A sequence in $(C \cup R)^*$ is **well-formed** if every return is preceded by a matching call, each identifier is used at most once.

A sequence in $(C \cup R)^*$ is **sequential** if there exists a return between every successive two calls.
Formalizing Libraries/Programs

**Definition 3.1.** A library \(L\) is an LTS over alphabet \(C \cup R\) such that each execution \(e \in E(L)\) is well formed, and

- Call actions \(c \in C\) cannot be disabled:
  \[e \cdot e' \in E(L)\text{ implies } e \cdot c \cdot e' \in E(L)\text{ if } e \cdot c \cdot e'\text{ is well formed.}\]
- Call actions \(c \in C\) cannot disable other actions:
  \[e \cdot a \cdot c \cdot e' \in E(L)\text{ implies } e \cdot c \cdot a \cdot e' \in E(L).\]
- Return actions \(r \in R\) cannot enable other actions:
  \[e \cdot r \cdot a \cdot e' \in E(L)\text{ implies } e \cdot a \cdot r \cdot e' \in E(L).\]

**Definition 3.2.** A program \(P\) over actions \(\Sigma\) is an LTS over alphabet \((\Sigma \cup C \cup R)\) where each execution \(e \in E(P)\) is well formed, and

- Call actions \(c \in C\) cannot enable other actions:
  \[e \cdot c \cdot a \cdot e' \in E(P)\text{ implies } c \leftarrow a\text{ or } e \cdot a \cdot c \cdot e' \in E(P).\]
- Return actions \(r \in R\) cannot disable other actions:
  \[e \cdot a \cdot r \cdot e' \in E(P)\text{ implies } a \leftarrow r\text{ or } e \cdot r \cdot a \cdot e \in E(P).\]
- Return actions \(r \in R\) cannot be disabled:
  \[e \cdot e' \in E(P)\text{ implies } e \cdot r \cdot e' \in E(L)\text{ if } e \cdot r \cdot e'\text{ is well formed.}\]
Observational Refinement

**Definition 3.3.** The library $L_1$ refines $L_2$, written $L_1 \leq L_2$, iff

$$E(P \times L_1)|\Sigma \subseteq E(P \times L_2)|\Sigma$$

for all programs $P$ over actions $\Sigma$. 
Histories

For given sets \( \mathbb{M} \) and \( \mathbb{V} \) of methods and values, we fix a set \( \mathbb{L} = \mathbb{M} \times \mathbb{V} \times (\mathbb{V} \cup \{\bot\}) \) of operation labels, and denote the label \( \langle m, u, v \rangle \) by \( m(u) \Rightarrow v \). A history \( h = \langle O, <, f \rangle \) is a partial order \(<\) on a set \( O \subseteq \Theta \) of operation identifiers labeled by \( f : O \rightarrow \mathbb{L} \) for which \( f(o) = m(u) \Rightarrow \bot \) implies \( o \) is maximal in \(<\). The history \( H(e) \) of a well-formed execution \( e \in \Sigma^* \) labels each operation with a method-call summary, and orders non-overlapping operations:

- \( O = \{ \text{op}(e_i) : 0 \leq i < |e| \text{ and } e_i \in C \} \),
- \( \text{op}(e_i) < \text{op}(e_j) \) iff \( i < j \), \( e_i \in R \), and \( e_j \in C \).
- \( f(o) = \begin{cases} m(u) \Rightarrow v & \text{if } m(u)_o \in e \text{ and } \text{ret}(v)_o \in e \\ m(u) \Rightarrow \bot & \text{if } m(u)_o \in e \text{ and } \text{ret}(\_)_o \notin e \end{cases} \)

The histories admitted by a library \( L \) are \( H(L) = \{ H(e) : e \in E(L) \} \)
Histories

happens-before partial order

push(1) → pop ⇒ 1 → push(2) → push(3) → pop ⇒ EMPTY
Histories

**Definition 4.2.** Let \( h_1 = \langle O_1, <_1, f_1 \rangle \) and \( h_2 = \langle O_2, <_2, f_2 \rangle \). We say \( h_1 \) is weaker than \( h_2 \), written \( h_1 \preceq h_2 \), when there exists an injection \( g : O_2 \rightarrow O_1 \) such that

- \( o \in \text{range}(g) \) when \( f_1(o) = m(u) \Rightarrow v \) and \( v \neq \perp \),
- \( g(o_1) <_1 g(o_2) \) implies \( o_1 <_2 o_2 \) for each \( o_1, o_2 \in O_2 \),
- \( f_1(g(o)) \preceq f_2(o) \) for each \( o \in O_2 \).

where \( (m_1(u_1) \Rightarrow v_1) \preceq (m_2(u_2) \Rightarrow v_2) \) iff \( m_1 = m_2 \), \( u_1 = u_2 \), and \( v_1 \in \{v_2, \perp\} \). We say \( h_1 \) and \( h_2 \) are equivalent when \( h_1 \preceq h_2 \) and \( h_2 \preceq h_1 \).

Examples?

Equivalent histories need not be distinguished
Histories

If $h_1 \in H(L)$ and $h_2 \preceq h_1$ then $h_2 \in H(L)$.

$E(L) = \{ e \in (C \cup R)^* : H(e) \in H(L) \}$. 
History Inclusion

**THEOREM**

$L_1$ refines $L_2$  \iff  $\text{H}(L_1) \subseteq \text{H}(L_2)$  \iff  $\text{E}(L_1) \subseteq \text{E}(L_2)$

- ($\Rightarrow$) Given $h$ in $\text{Hist}(L_1)$, construct a program $P_h$ that imposes all the happen-before constraints of $h$.
- ($\Leftarrow$) Clients cannot distinguish executions with the same history. History inclusion implies Execution Inclusion
History Inclusion (=>)

We construct \( P_h = \langle Q, \Sigma, q_0, \delta \rangle \) over alphabet \( \Sigma = C \cup R \cup \{a\} \) whose states \( Q : O \to \mathbb{B}^2 \) track operations called/completed status. The initial state is \( q_0 = \{ o \mapsto (\bot, \bot) : o \in O \} \). Transitions are given by,

for each \( q \in Q, o \in O, m \in M, v \in V \)

\[
\begin{align*}
&\text{if } f(o) = m(v) \Rightarrow \bot \text{ and } q(o') \text{ for all } o' < o \text{ then} \\
&q[o \mapsto \bot, \bot] \xrightarrow{m(v)} q[o \mapsto T, \bot] \quad \text{preserving happens-before} \\
&\text{if } f(o) = m(\bot) \Rightarrow v \text{ then} \\
&q[o \mapsto T, \bot] \xrightarrow{\text{ret}(v)} \cdot \xrightarrow{a} q[o \mapsto T, T] \quad \text{counting ops completed in } h \\
&\text{if } f(o) = m(\bot) \Rightarrow \bot \text{ then} \\
&q[o \mapsto T, \bot] \xrightarrow{\text{ret}(v)} q[o \mapsto T, T] \quad \text{ops that are pending in } h \text{ (an execution may have more completed ops and less pending - no call for pending)}
\end{align*}
\]

\[ (??) \forall e \in E(P_h). \quad |(e|\Sigma)| = n \implies h \leq H(e) = a^n \quad \text{nb of completed ops in } h \]
History Inclusion (=>)

(??) \( \forall e \in E(P_h). |(e|\Sigma)| = n \implies h \preceq H(e) \)

For every execution \( e_1 \in E(P_h \times L_1) \) with \( e_1|\Sigma = n \),

there must exist an execution \( e_2 \in E(P_h \times L_2) \) such that \( e_2|\Sigma = e_1|\Sigma \)

(by observational refinement)

Therefore, \( h \preceq H(e_2) \).

Since \( e_2| (C \cup R) \in E(L_2) \), we have that \( H(e_2) \in H(L_2) \)

By closure under weakening, \( h \in H(L_2) \)
History Inclusion (\(\leq\))

**THEOREM**

\[ L_1 \text{ refines } L_2 \iff H(L_1) \subseteq H(L_2) \iff E(L_1) \subseteq E(L_2) \]

Let \( e \in E(P \times L_1) \)

\( e \mid (C \cup R) \in E(L_1) \) implies \( H(e) \in H(L_1) \) implies \( H(e) \in H(L_2) \)

Therefore, \( e \mid (C \cup R) \in E(L_2) \) which by definition of the product \( P \times L_2 \), implies \( e \in E(P \times L_2) \)
Linearizability [Herlihy&Wing 1990]

Effects of each invocation appear to occur instantaneously

\[\exists \text{ lin. } rb \subseteq \text{ lin } \wedge \text{ lin } \in \text{ Queue ADT}\]
About Linearizability

History inclusion \( H(L_1) \subseteq H(L_2) \) equiv. to linearizability when \( L_2 \) is atomic

**Definition 3.1.** A library \( L \) is an LTS over alphabet \( C \cup R \) such that each execution \( e \in E(L) \) is well formed, and

- **Call actions** \( c \in C \) cannot be disabled:
  \[
e \cdot e' \in E(L) \text{ implies } e \cdot c \cdot e' \in E(L) \text{ if } e \cdot c \cdot e' \text{ is well formed.}
  \]
  
- **Call actions** \( c \in C \) cannot disable other actions:
  \[
e \cdot a \cdot c \cdot e' \in E(L) \text{ implies } e \cdot c \cdot a \cdot e' \in E(L).
  \]
  
- **Return actions** \( r \in R \) cannot enable other actions:
  \[
e \cdot r \cdot a \cdot e' \in E(L) \text{ implies } e \cdot a \cdot r \cdot e' \in E(L).
  \]

We write \( e_1 \sim e_2 \) when \( e_2 \) can be derived from \( e_1 \) by applying zero or more of the above rules. The closure of a set \( E \) of executions under \( \sim \) is denoted \( \overline{E} \).

A library \( L \) is called atomic if it is defined by the closure of some set \( E \) of sequential executions, i.e., \( E(L) = \overline{E} \).
About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when $L_2$ is \textbf{atomic}

Linearizability is defined by an execution order: $e_1 \sqsubseteq e_2$ iff there exists a well-formed execution $e'_1$ obtained from $e_1$ by appending return actions, and deleting call actions, such that:

$e_2$ is a permutation of $e'_1$ that preserves the order between return and call actions, i.e., a given return action occurs before a given call action in $e'_1$ iff the same holds in $e_2$.

An execution $e_1$ is \textit{linearizable} w.r.t. a library $L_2$ iff there exists a sequential execution $e_2 \in E(L_2)$, with only completed operations, such that $e_1 \sqsubseteq e_2$. A library $L_1$ is \textit{linearizable} w.r.t. $L_2$, written $L_1 \sqsubseteq L_2$, iff each execution $e_1 \in E(L_1)$ is linearizable w.r.t. $L_2$. 
About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when $L_2$ is atomic

Linearizability compares execs of $L_1$ with pending ops. with execs of $L_2$ with only completed ops => problematic when $L_2$ contains non-terminating methods

**Example 5.1.** Let $L$ be the library whose kernel contains the single execution $e = m(u)_1 m'(u)_2 \text{ret}(v)_1$, in which the call to $m'$ is pending. Although $L$ refines itself, since refinement is reflexive, $L$ is not linearizable w.r.t. itself, since $e$ could only be linearizable w.r.t. $L$ if $E(L)$ were to contain one of the following executions:

- $m(u)_1 \text{ret}(v)_1$,
- $m(u)_1 m'(u)_2 \text{ret}(v)_1 \text{ret}(-)_2$
- $m(u)_1 \text{ret}(v)_1 m'(u)_2 \text{ret}(-)_2$,
- $m'(u)_2 \text{ret}(-)_2 m(u)_1 \text{ret}(v)_1$.

Yet $E(L) = \{e\}$ clearly contains none of them.
About Linearizability

History inclusion $H(L_1) \subseteq H(L_2)$ equiv. to linearizability when $L_2$ is atomic

**Lemma 5.1.** $e_1 \sqsubseteq e_2$ iff $H(e_1) \leq H(e_2)$.

**Theorem 2.** $L_1 \subseteq L_2$ iff $H(L_1) \subseteq H(L_2)$, if $L_2$ is atomic.

Proof. $(=>)$ Let $h \in H(L_1)$. Then, every execution $e_1$ with $H(e_1) = h$ is linearizable w.r.t. some execution $e_2 \in L_2$

By the lemma above, $H(e_1) \leq H(e_2)$. By closure under weakening, if $H(e_2) \in H(L_2)$ then any weakening, $h$ in particular, belongs to $H(L_2)$.

$(<=)$ Let $e_1 \in E(L_1)$. By hypothesis, $H(e_1) \in H(L_2)$, which implies $e_1 \in E(L_2)$.

Since $L_2$ is atomic, there exists a sequential $e_2 \in E(L_2)$ with only completed ops such that $H(e_1) \in H(L_2)$ such that $e_1$ is lin. w.r.t. $e_2$. 
Linearizability Proofs based on Forward Simulations
Linearizability vs Refinement

- Modelling concurrent objects with Labeled Transition Systems (LTSs)
- Linearizability is a property of sequences of call/return actions
- Given an ADT A, define a reference implementation `Spec(A)` which admits all histories linearizable w.r.t. A
  - standard reference implementations (atomic method bodies): call, return, and linearization point actions
  
  ![Diagram](image)

  - Linearizability = inclusion of traces with call/return actions (these are the only common actions) between `Impl` and `Spec(A)`
    - the actions included in traces are called observable
Proving Refinement

Inductive reasoning for proving refinement: forward/backward simulations

Simulations: relations between states of the impl. and spec., relating initial states and

Implementation:

Specification:

Forward

Backward
Proving Refinement

- Given two LTSs A and B such that A refines B [Abadi et al.’91, Lynch et al.’95]

<table>
<thead>
<tr>
<th>Frw Sim (FS)</th>
<th>Bckw Sim (BS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>exists if</td>
<td>B deterministic</td>
</tr>
<tr>
<td>exists if we add</td>
<td>A forest</td>
</tr>
<tr>
<td>Prophecy vars to A</td>
<td>History vars to A</td>
</tr>
</tbody>
</table>

- Forward simulations are easier to derive and establish (standard invariant checking)
Proving Linearizability

- **Impl** is linearizable w.r.t. **A** iff **Impl** refines **Spec(A)**
  - refinement = inclusion of traces with call/return actions (observable actions)

- **Spec(A)** is not deterministic when projected on observable actions =>
  backward simulations are unavoidable in general

- Classes of implementations for which forward simulations are sufficient -
  associate linearization points with statements of the implementation
  - the linearization point actions become **observable**
  - **Spec(A)** is deterministic assuming that **A** is **deterministic**
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;
}
```
Herlihy & Wing Queue

```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;
        }
    }
}
```
Non-fixed Linearization Points

**Enqueue**

1. **i(e,x)**: index \( i \) of enqueue with id \( e \) that will insert item \( x \)

   - `NULL` → `NULL` → `NULL` → `x`

**Dequeue**

1. **d: range, i = ...**

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: CAS(...)` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`

   - `NULL` → `NULL` → `NULL` → `NULL` → `d: i ++`
Non-fixed Linearization Points

\[ i(\epsilon_1, x) \quad e_1: \text{inv}(x) \quad e_1: \text{back}++ \]
\[ e_2: \text{inv}(y) \quad i(\epsilon_1, x) \quad \text{null} \quad i(\epsilon_1, y) \quad \text{null} \quad \text{null} \quad \text{null} \quad \text{null} \quad \text{null} \]

\[ e_2: \text{back}++ \quad i(\epsilon_1, x) \quad \text{null} \quad i(\epsilon_1, y) \quad \text{null} \quad \text{null} \quad \text{null} \quad \text{null} \quad \text{null} \]
\[ e_2: \text{items}[i] = y \]

\[ e_1: \text{items}[i] = x \]
\[ e_2: \text{ret} \]

\[ d_1: \text{deq}(x) \]
\[ d_2: \text{deq}(y) \]
Non-fixed Linearization Points

\[ e_1: inv(x) \quad e_2: i = \text{back}++ \]
\[ e_2: inv(y) \quad e_2: i = \text{back}++ \]
\[ e_2: \text{items}[i] = y \]
\[ e_1: \text{items}[i] = x \]
\[ e_1: \text{ret} \]
\[ d_2: \text{deq}(y) \]
\[ d_2: \text{deq}(x) \]

\[ e_1: inv(x) \quad e_1: i = \text{back}++ \]
\[ e_2: i = \text{back}++ \]
\[ e_2: \text{items}[i] = y \]
\[ e_1: \text{items}[i] = x \]
\[ e_1: \text{ret} \]
\[ d_2: \text{deq}(y) \]
\[ d_2: \text{deq}(x) \]

\[ e_1: inv(x) \quad e_1: i = \text{back}++ \]
\[ e_2: i = \text{back}++ \]
\[ e_2: \text{items}[i] = y \]
\[ e_1: \text{items}[i] = x \]
\[ e_1: \text{ret} \]
\[ d_2: \text{deq}(y) \]
\[ d_2: \text{deq}(x) \]
Non-fixed Linearization Points

Non-fixed linearization points => proofs based on forward simulations are impossible in general

Possible for certain ADTs, queues and stacks [BEEM-CAV’17]
  • assuming fixed linearization points only for dequeue/pop
  • reference implementations whose states are partial orders of enq/push

Non-fixed Linearization Points
Non-fixed Linearization Points

Non-fixed linearization points => proofs based on forward simulations are impossible in general

Possible for certain ADTs, queues and stacks [BEEM-CAV’17]

• assuming fixed linearization points only for dequeue/pop
• reference implementations whose states are partial orders of enq/push
Non-fixed Linearization Points

**Non-fixed** linearization points => proofs based on **forward simulations** are impossible in general

Possible for certain ADTs, **queues and stacks** [BEEM-CAV’17]

- assuming **fixed** linearization points only for **dequeue/pop**
- reference implementations whose states are **partial orders** of enq/push

![Diagram showing linearization points and corresponding operations for dequeue and enqueue operations.](image)
Forward Sim. for H&W Queue

FS \( f \) between HWQ and AbsQ. Given a HWQ state \( s \) and an AbsQ state \( t \), \((s, t) \in f\) iff:

- Pending enqueues in \( s \) are pending and maximal in \( t \).
- Order in \( t \) is consistent with the positions reserved in items of \( s \).
- For two enqueues \( e_1, e_2 \) and dequeue \( d \), if \( e_1 \) reserves a position before \( e_2 \), \( d \) is visiting an index in between and \( d \) can remove \( e_2 \) in \( s \), then \( e_1 \) cannot be ordered before \( e_2 \) in \( t \).