Reducing Linearizability to Classic Verification Problems
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

push(1) → pop ⇒ 2

pop ⇒ 1 → push(2) → push(3) → pop ⇒ 3
Concurrent Queues [ICALP’15]

Linearizability $\equiv$ Exclusion of bad patterns (assuming each value is enqueued at most once - sound under data independence)
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
Concurrent Queues \textsuperscript{[ICALP’15]}

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

“Value \textit{v dequeued without being enqueued}”

```
\begin{align*}
\text{deq: } v \\
\end{align*}
```

“Value \textit{v dequeued before being enqueued}”

```
\begin{align*}
\text{deq: } v & \quad \text{enq: } v \\
\end{align*}
```
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
```

[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”

```
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: v_1  ---  enq: v_2  ---  deq: v_2  ---  deq: v_1

```
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”

“Value v dequeued before being enqueued”

“Value v dequeued twice”

“Values dequeued in the wrong order”

“Dequeue wrongfully returns 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"
- "Value v dequeued before being enqueued"
- "Value v dequeued twice"
- "Values dequeued in the wrong order"
- "Dequeue wrongfully returns empty"
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** [ICALP’15]

**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: v_1` → `enq: v_2` → `deq: v_2` → `deq: v_1`

- "Dequeue wrongfully returns empty"
  - `enq: v_1` → `enq: v_2` → `deq: empty`

- "Values dequeued in the wrong order"
  - `enq: v_n` → `deq: v_{n-1}` → `deq: v_n`
Concurrent Stacks

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

“We Value v popped without being pushed”
“We Value v popped before being pushed”
“We Value v popped twice”
“We 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”

“Pop doesn’t return the top of the stack”
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 $i$, $L \cap A_i = \emptyset$
Inductive definition of the Register

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

- including the empty sequence
Inductive definition of the Queue

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

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

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

Derivation:

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

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

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

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

Derivation for \( Push_1 \, Push_2 \, Pop_2 \, Pop_1 \, EMP \, Push_3 \, Pop_3 \in S \)

\[ \epsilon \in S \]
\[ \Rightarrow \, Push_3 \, Pop_3 \in S \]
\[ \Rightarrow \, Push_2 \, Pop_2 \, Push_3 \, Pop_3 \in S \]
\[ \Rightarrow \, Push_1 \, Push_2 \, Pop_2 \, Pop_1 \, Push_3 \, Pop_3 \in S \]
\[ \Rightarrow \, Push_1 \, Push_2 \, Pop_2 \, Pop_1 \, EMP \, Push_3 \, Pop_3 \in S \]
Data Independence

- Input methods = methods taking an argument
- A sequential execution $u$ is called 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 $S$
Data Independence

- Input methods = methods taking an argument
- A sequential execution \( u \) is called *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 *renaming* \( r \) is a function from \( \mathbb{D} \) to \( \mathbb{D} \). 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 \( r(x) \).

**Definition 6.** The set of sequential executions (resp., executions or histories) \( S \) is data independent if:

- for all \( u \in S \), there exists \( u' \in S_\neq \), and a renaming \( r \) such that \( u = r(u') \),
- for all \( u \in S \) and for all renaming \( r \), \( r(u) \in S \).
Data Independence

- Input methods = methods taking an argument
- A sequential execution $u$ is called 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 renaming $r$ is a function from $\mathbb{D}$ to $\mathbb{D}$. 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 $r(x)$.

**Definition 6.** The set of sequential executions (resp., executions or histories) $S$ is data independent if:

- for all $u \in S$, there exists $u' \in S_\neq$, and a renaming $r$ such that $u = r(u')$,
- for all $u \in S$ and for all renaming $r$, $r(u) \in S$.

**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 $\text{last}(u') = R_i$. 
Characterization of sequential executions

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

1. For all \( u \in \left[ S \right] \), 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 \( u \notin \left[ S \right] \), \( \text{last}(u) \) is also defined but can be arbitrary, as there is no derivation for \( u \).

2. If \( \text{last}(u) = R_i \), then for every permutation \( u' \in \left[ S \right] \) 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)

$w \in Q$ 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:

\[
e \setminus x \subseteq [R_1, \ldots, R_i] \implies e \subseteq [R_1, \ldots, R_i]
\]
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 notion of step-by-step linearizability ensures that the history is linearizable w.r.t. a queue.

- 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 deques
- 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
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 \( u_2 = a \cdot b_1 \cdot b_2 \ldots b_s \cdot u' \). 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. \( \square \)
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 $j$):

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

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 $MS(R_i)$ with $i > k$, and in particular no projection of $e'$. Thus, we deduce that $\text{proj}(e') \subseteq \bigcup_{i \leq k} MS(R_i)$, 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 $k \leq j$. \(\square\)
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 (\(\Leftarrow\)) 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 $k \leq j$. \(\square\)

Lemma

$E$ is linearizable to $Q$ iff every projection $E'$ of $E$ is linearizable to the form $\text{Enq}_x \cdot u \cdot \text{Deq}_x \cdot v$ (with $u \in \text{Enq}^*$) or to the form $u \cdot \text{EMP} \cdot v$ (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:

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

\[ e \nsubseteq S \iff \exists e' \in \text{proj}(e). e' \nsubseteq MS(R) \text{ (where } R = \text{last}(e')) \]

*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 $R_{\text{Enq}Deq}$):

\[
\begin{array}{ccc}
\text{Enq}_1 & \text{Deq}_2 \\
\hline
\text{Enq}_1 \prec \text{Enq}_2 \\
\text{Deq}_2 \prec \text{Deq}_1
\end{array}
\]

or $\text{Deq}_1$ before $\text{Enq}_1$
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:

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

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

\textit{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:

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

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

R \text{EMP}
Exercice

• 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"

deq: empty

enq: v₁  deq: v₁

enq: v₂  deq: v₂  deq: v₁

enq: vₙ  deq: vₙ⁻¹...

enq: vₙ  deq: vₙ

deq: v₁
Linearizability vs Refinement

[enq(v)]

[a,b,...] => [a,b,...,v]

[deq()]

[v,a,b,...] => [a,b,...]
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 \( \text{Spec}(A) \) which admits all histories linearizable w.r.t. A
  - standard reference implementations (atomic method bodies): call, return, and linearization point actions
    - Linearizability = inclusion of traces with call/return actions (these are the only common actions) between \( \text{Impl} \) and \( \text{Spec}(A) \)
      - 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
Proving Refinement

Inductive reasoning for proving refinement: forward/backward simulations

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

Forward

Implementation: $i_s_1 \xrightarrow{a} i_s_2$

Specification: $a_s_1$
Proving Refinement

Inductive reasoning for proving refinement: forward/backward simulations

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

Forward

Implementation: \( i_s_1 \rightarrow a \rightarrow i_s_2 \)

Specification: \( a s_1 \rightarrow a s_2 \)
Proving Refinement

Inductive reasoning for proving refinement: forward/backward simulations

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

Forward

Implementation:

Specification:
Proving Refinement

**Inductive reasoning for proving refinement:** forward/backward simulations

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

- **Forward**
  - Implementation: $i_{s1} \xrightarrow{a} i_{s2}$
  - Specification: $a_{s1} \xrightarrow{a} \exists a_{s2}$

- **Backward**
  - Implementation: $i_{s1} \xrightarrow{a} i_{s2}$
  - Specification: $i_{s2}$
Proving Refinement

Inductive reasoning for proving refinement: forward/backward simulations

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

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

Properties:

- [Lynch et al., 1995] Given two LTSs $A$ and $B$ such that $A$ refines $B$, Frw Sim (FS) exists if $B$ deterministic and Bckw Sim (BS) exists if $A$ forest.

Constantin Enea (KU)
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>Prophecy vars to A</td>
</tr>
<tr>
<td></td>
<td>History vars to A</td>
</tr>
</tbody>
</table>

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

[Diagram showing the sequence of calls and linearization points]

- Call `enq(v)`
- Linearize `enq(v)`
- Call `enq(v')`
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;
}
```

Treiber Stack
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

Dequeue

Herlihy & Wing Queue

$i(e,x)$: index $i$ of enqueue with id $e$ that will insert item $x$
Non-fixed Linearization Points
Non-fixed Linearization Points

\[ e_1: \text{inv}(x) \quad e_1: \text{back++} \]

\[ e_2: \text{inv}(y) \quad e_2: \text{back++} \]

\[ e_2: \text{items}[i] = y \]

\[ d_2: \text{deq}(y) \]

\[ d_1: \text{deq}(x) \]

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

i = back++

\[ i(e_1, x) \]

\[ i(e_2, x) \]

\[ i(e_2, y) \]
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
Non-fixed Linearization Points

... enq(v₁):compl

happens-before of enqueues

... enq(v₂):compl

... enq(v₃):pend

ret enq(v)

... enq(v):pend

... ret enq(v)

... enq(v):compl

... enq(v₂):compl

... enq(v₃):pend

... enq(v₁):compl
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

happens-before of enqueues

\[
\begin{align*}
\text{enq}(v_1):\text{compl} & \quad \text{enq}(v_3):\text{pend} \\
\text{enq}(v_2):\text{compl} & \quad \text{enq}(v):\text{pend} \\
\text{...} & \\
\end{align*}
\]

\[\text{ret enq}(v)\]

\[
\begin{align*}
\text{enq}(v_1):\text{compl} & \quad \text{enq}(v_3):\text{pend} \\
\text{enq}(v_2):\text{compl} & \quad \text{enq}(v):\text{compl} \\
\text{...} & \\
\end{align*}
\]
Non-fixed Linearization Points

\[ \text{enq}(v_0) : - \]
\[ \text{enq}(v_1) : \text{compl} \]
\[ \text{enq}(v_2) : \text{compl} \]
\[ \text{enq}(v_3) : \text{pend} \]

\[ \text{lin deq}(v_0) \]

\[ \text{enq}(v_0) : - \]
\[ \text{enq}(v_1) : \text{compl} \]
\[ \text{enq}(v_2) : \text{compl} \]
\[ \text{enq}(v_3) : \text{pend} \]

minimal element
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 enq(v0):-, enq(v1):compl, enq(v2):compl, enq(v3):pend, lin deq(v0), minimal element]
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 \( t \).