Correctness in a Weakly Consistent Setting

Azalea Raad
Max Planck Institute for Software Systems (MPI-SWS)
History

Difficulty

 Sequential

 Sequential Consistency (SC)

 Weak Memory Concurrency (WMC)

time
Sequential Consistency (SC)

SC (a.k.a. interleaving concurrency):

**total** execution order (to) that respects program order (po)
Sequential Consistency (SC)

SC (a.k.a. interleaving concurrency):
- **total** execution order (to) that respects program order (po)

<table>
<thead>
<tr>
<th>Execution order (to)</th>
<th>( \mathbb{V}_b )</th>
<th>( \mathbb{V}_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

a before d ; c after b
Sequential Consistency (SC)

SC (a.k.a. interleaving concurrency):
*total* execution order (*to*) that respects program order (*po*)

<table>
<thead>
<tr>
<th>Execution order (<em>to</em>)</th>
<th>V_b</th>
<th>V_d</th>
</tr>
</thead>
<tbody>
<tr>
<td>a b c d</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>c d a b</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

- a before d; c after b
- a after d; c before b
Sequential Consistency (SC)

SC (a.k.a. interleaving concurrency):

**total** execution order (**to**) that respects program order (**po**)

### Execution order (**to**) | **V<sub>b</sub>** | **V<sub>d</sub>**
--- | --- | ---
abcde | 0 | 1
\[c\,d\,a\,b\] | 1 | 0
\[a\,c\,b\,d\,\] | 1 | 1
\[\,a\,c\,b\,d\,\] | 1 | 1
\[\,c\,a\,b\,d\,\] | 1 | 1
\[\,c\,a\,d\,b\,\] | 1 | 1

- a before d; c after b
- a after d; c before b
- a before d; c before b
Sequential Consistency (SC)

SC (a.k.a. interleaving concurrency):

*total* execution order (to) that respects program order (po)

<table>
<thead>
<tr>
<th>Vb</th>
<th>Vd</th>
<th>a before d ; c after b</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>a after d ; c before b</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>a before d ; c before b</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>not possible!</td>
</tr>
</tbody>
</table>
Weak Memory Concurrency (WMC)

No total execution order ($to$) $\Rightarrow$

*anomalies (litmus tests)*: behaviour absent under SC;
caused by:

- instruction *reordering*;
  - e.g. store buffering (SB)
Weak Memory Concurrency (WMC)

No total execution order (to) ⇒

**anomalies (litmus tests):** behaviour absent under SC; caused by:

- instruction **reordering**;
  - e.g. store buffering (SB)
- **different write propagation** across cache hierarchy;
  - e.g. Independent Reads of Independent Writes (IRIW)
WMC: Store Buffering

\[ a \ x := 1; \quad \parallel \quad c \ y := 1; \]
\[ b := y \quad \text{// } v_b \quad \parallel \quad d := x \quad \text{// } v_d \]

<table>
<thead>
<tr>
<th>( v_b )</th>
<th>( v_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

possible, due to reordering!

\[ 0 \ 0 \quad \text{store buffering(SB)} \]
Fences to Stop Reordering

\[ \begin{align*}
\text{a: } & \quad x := 1; \\
\text{b: } & \quad b := y \quad \text{// } v_b \\
\text{c: } & \quad y := 1; \\
\text{d: } & \quad d := x \quad \text{// } v_d
\end{align*} \]

<table>
<thead>
<tr>
<th>( V_b )</th>
<th>( V_d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

possible, due to reordering! 0 0
WMC: Independent Reads of Independent Writes

annotated behaviour *not* allowed *under SC* :

```
\begin{align*}
\text{a} & : x := 1 \\
\text{b} & : \text{b} := \text{x}; \quad // 1 \\
\text{c} & : \text{c} := \text{y}; \quad // 0 \\
\text{d} & : \text{y} := 1 \\
\text{e} & : \text{e} := \text{y}; \quad // 1 \\
\text{f} & : \text{f} := \text{x}; \quad // 0
\end{align*}
```
WMC: Independent Reads of Independent Writes

annotated behaviour *not* allowed *under SC*:

```
\[\begin{array}{c}
a \ x := 1 \\
b \ b := x; \quad // 1 \\
c \ c := y \quad // 0 \\
d \ y := 1 \\
e \ e := y; \quad // 1 \\
f \ f := x \quad // 0 \\
\end{array}\]
```
WMC: Independent Reads of Independent Writes

<table>
<thead>
<tr>
<th>( x := 1 )</th>
<th>( b := x; ) ( // 1 )</th>
<th>( y := 1 )</th>
<th>( e := y; ) ( // 1 )</th>
<th>( f := x ) ( // 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( a )</td>
<td>( b )</td>
<td>( c := y ) ( // 0 )</td>
<td>( e )</td>
<td>( f )</td>
</tr>
</tbody>
</table>

annotated behaviour *not* allowed *under SC*:

```
  a  to  d
    X
  d  to  a
    X
```
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

```
resp
T1

a x := 1

T2

b b := x;

c c := y

d y := 1

e e := y;

f f := x

T3

y = 1

T4

y = 1

cache hierarchy

memory

림계비티리코피아토미아 커피치

non-multi-copy-atomic architecture
```
WMC: Independent Reads of Independent Writes

```
x:=1
```

```
T1
x=1
```

```
T2
b:=x;
c:=y
```

```
T3
y:=1
```

```
T4
e:=y;
f:=x
```

```
x=0
y=0
```

```
non-multi-copy-atomic architecture
```
WMC: Independent Reads of Independent Writes

T1: x := 1
T2: b := x; c := y
T3: y := 1
T4: e := y; f := x

cache hierarchy

x = 0 y = 0

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

T1

\[ x := 1 \]

T2

\[ b := x; // 1 \]
\[ c := y; // 0 \]

T3

\[ y := 1 \]

T4

\[ e := y; \]
\[ f := x \]

\( x := 1 \)
\( a \)
\( b \)
\( c \)
\( d \)
\( e \)
\( f \)

\( y := 1 \)

\( b := x; // 1 \)
\( c := y; // 0 \)
\( y := 1 \)
\( f := x \)

\( x = 1 \)
\( y = 1 \)
\( y = 0 \)

\( x = 0 \)
\( y = 0 \)

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

\[
\begin{align*}
&x := 1 \\
&b := x; // 1 \\
&c := y // 0 \\
&y := 1 \\
&e := y; // 1 \\
&f := x
\end{align*}
\]

\[
\begin{align*}
x &= 1 \\
y &= 1 \\
x &= 0 \\
y &= 0
\end{align*}
\]

Cache hierarchy

Memory

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

```
x := 1
b := x; // 1
c := y // 0
d := 1
e := y; // 1
f := x
```

T1

```
x = 1
```

T2

```
```

T3

```
y = 1
```

T4

```
```

cache hierarchy

memory

**non-multi-copy-atomic** architecture
WMC: Independent Reads of Independent Writes

\[ x := 1 \]
\[ c := y \] // 0
\[ y := 1 \]
\[ f := x \]

\[ a \]
\[ b := x ; / / 1 \]
\[ e := y ; / / 1 \]

\[ T1 \]
\[ T2 \]
\[ T3 \]
\[ T4 \]

cache hierarchy

memory

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

\[
x := 1
\]

\[
b := x; // 1
\]

\[
c := y // 0
\]

\[
y := 1
\]

\[
e := y; // 1
\]

\[
f := x // 0
\]

non-multi-copy-atomic architecture
WMC: Independent Reads of Independent Writes

non-multi-copy-atomic architecture
Weak Memory Concurrency (WMC)

No total execution order \((to) \Rightarrow\)

**anomalies** (litmus tests): behaviour absent under SC; caused by:

- instruction **reordering**;
  - e.g. store buffering (SB)
- **different write propagation** across cache hierarchy;
  - e.g. Independent Reads of Independent Writes (IRIW)

programming under WMC can be unintuitive and **error-prone**
Weak Memory Concurrency (WMC)

No total execution order ($to$) $\Rightarrow$

**anomalies** (*litmus tests*): behaviour absent under SC; caused by:

- instruction reordering;  
  — e.g. store buffering (SB)
- different write propagation across cache hierarchy;  
  — e.g. Independent Reads of Independent Writes (IRIW)

*Formal Methods* **to the rescue!**
WMC: State of the Art

• Formal Specification
  ‣ **Hardware** (architecture) level WMC specification
    — e.g. x86-TSO, ARMv7, ARMv8, POWER, …
  ‣ **Software** (language) level WMC specification
    — e.g. C/C++11, Java, …
WMC: State of the Art

- Formal Specification
  - **Hardware** (architecture) level WMC specification
    - e.g. x86-TSO, ARMv7, ARMv8, POWER, ...
  - **Software** (language) level WMC specification
    - e.g. C/C++11, Java, ...

- Formal Verification
  - Correctness of *language to architecture compilation*, e.g. C11 to x86
  - Correctness of *compiler transformations*
  - **Verified compilers**, e.g. CompCertTSO
  - Case studies
    - Linux RCU, ARC, …
WMC: State of the Art

- **Formal Specification**
  - *Hardware* (architecture) level WMC specification
    - e.g. x86-TSO, ARMv7, ARMv8, POWER, …
  - *Software* (language) level WMC specification
    - e.g. C/C++11, Java, …

- **Formal Verification**
  - Correctness of *language to architecture compilation*, e.g. C11 to x86
  - Correctness of *compiler transformations*
  - *Verified compilers*, e.g. CompCertTSO
  - Case studies
    - Linux RCU, ARC, …

\[low-level!\]
WMC: State of the Art

- Formal Specification
  - Hardware (architecture) level WMC specification — e.g. x86-TSO, ARMv7, ARMv8, POWER, ...
  - Software (language) level WMC specification — e.g. C/C++11, Java, ...

- Formal Verification
  - Correctness of language to architecture compilation, e.g. C11 to x86
  - Correctness of compiler transformations
  - Verified compilers, e.g. CompCertTSO
  - Case studies — Linux RCU, ARC, ...

What about high-level concurrent library specification & verification under WMC?
Part I.

Concurrent Library *Specification* under WMC
Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
  - large body of case studies
Concurrent Library Specification

- Sequential Consistency (SC) -- well-explored
  - semantic-based: linearisability
  - program logics: Hoare logic, separation logic, etc.
    - large body of case studies

- Weak Memory Concurrency (WMC) -- under-explored
  - linearisability variants
  - program logic adaptations
    - small body of case studies
Concurrent Library Specification

• Sequential Consistency (SC) -- well-explored
  ‣ semantic-based: linearisability
  ‣ program logics: Hoare logic, separation logic, etc.
  ‣ **large** body of case studies

• Weak Memory Concurrency (WMC) -- under-explored
  ‣ linearisability variants
  ‣ program logic adaptations
  ‣ **small** body of case studies
  
  tied to a **particular** WMC memory model (MM)!
  E.g. C11, TSO, ...
Concurrent Library Specification

• Sequential Consistency (SC) -- well-explored
  ‣ semantic-based: linearisability
  ‣ program logics: Hoare logic, separation logic, etc.
  ‣ large body of case studies

• Weak Memory Concurrency (WMC) -- under-explored
  ‣ linearisability variants
  ‣ program logic adaptations
  ‣ small body of case studies

---

**wanted**

**general**

**MM-agnostic**

**declarative**

**specification & verification framework**

E.g. C11, TSO, ...

wanted general MM-agnostic declarative specification & verification framework
Declarative Framework Desiderata

- **Agnostic** to memory model
  - support *both SC* and *WMC* specs

- **General**
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs

- **Compositional**
  - verify *client programs*

```plaintext
q:=new-queue();
s:=new-stack();

enq(q,1); push(s,2)   a:=pop(s);
                        if(a==2) b:=deq(q) // returns 1
```
Declarative Framework Desiderata

- **Agnostic** to memory model
  - support *both* SC and WMC specs

- **General**
  - port *existing SC (linearisability) specs*
  - port *existing WMC specs* (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs

- **Compositional**
  - verify *client programs*
  - verify *library implementations* ⇒ *towers of abstraction*

```
q := new-queue();
s := new-stack();

enq(q, 1); a := pop(s);
push(s, 2) if a == 2
  b := deq(q) // returns 1
```
• Define a (partial) happens-before relation $hb$ on events
  
  $\forall (e_1, e_2) \in hb \iff e_1.\text{end} < \text{time} e_2.\text{begin}$
  
  -- e.g. $(a, b) \in hb$           $(a, c) \notin hb$
• Define a (partial) happens-before relation $hb$ on events
  > $(e_1, e_2) \in hb \iff e_1.\text{end} <_\text{time} e_2.\text{begin}$
    -- e.g. $(\text{a}, \text{b}) \in hb$ $(\text{a}, \text{c}) \notin hb$

• Linearisable $\iff \exists \text{to. to totally}$ orders events
  > $hb \subseteq \text{to}$
  > $\text{to}$ is a legal sequence (library-specific)
    -- e.g. $\text{to}$ is a FIFO sequence
Linearisability

- Define a (partial) happens-before relation $hb$ on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
    - e.g. $(a, b) \in hb$, $(a, c) \notin hb$

- **Linearisable** $\iff \exists \text{to. to totally}$ orders events
  - $hb \subseteq \text{to}$
  - $\text{to}$ is a **legal** sequence (library-specific)
    - e.g. $\text{to}$ is a **FIFO** sequence

```plaintext
thread 1

thread 2

time

donkey

• Define a (partial) happens-before relation $hb$ on events
  - $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
    - e.g. $(a, b) \in hb$, $(a, c) \notin hb$

• **Linearisable** $\iff \exists \text{to. to totally}$ orders events
  - $hb \subseteq \text{to}$
  - $\text{to}$ is a **legal** sequence (library-specific)
    - e.g. $\text{to}$ is a **FIFO** sequence

```
• Define a (partial) happens-before relation $hb$ on events
  
  $\bullet$ $(e_1, e_2) \in hb \iff e_1.\text{end} <_{\text{time}} e_2.\text{begin}$
  
  -- e.g. $(a, b) \in hb$ $(a, c) \notin hb$

• **Linearisable** $\iff \exists to. to$ **totally** orders events

  $\bullet$ $hb \subseteq to$

  $\bullet$ $to$ is a **legal** sequence (library-specific)

  -- e.g. $to$ is a **FIFO** sequence

linearisable

a b c a c b

✔ ✔ ❌
Why Not Linearisability?

- assumes \(<_{\text{time}}\) order -- not present under WMC
- requires \(\text{total}\) order on all events -- not always possible under WMC

```
x:=1;
b:=y // 0
```

```
y:=1;
d:=x // 0
```

(SB) not linearisable
Why Not Linearisability?

- Assumes \(\prec\)time order -- not present under WMC
- Requires total order on all events -- not always possible under WMC

? per-location linearisability?

\[
\begin{align*}
\text{a} & : x := 1; \\
\text{b} & : y \quad \text{// 0} \\
\text{c} & : y := 1; \\
\text{d} & : d := x \quad \text{// 0}
\end{align*}
\]

(SB)

\(x\) to \(y\) to \(d\) to \(a\)

\(b\) to \(c\)

not linearisable
Why Not Linearisability?

✘ assumes $\prec_{\text{time}}$ order -- not present under WMC
✘ requires total order on all events -- not always possible under WMC
❓ per-location linearisability?
✘ cannot model weak specs, e.g. C11, TSO

\[
\begin{align*}
\text{a} & : x := 1; \\
\text{b} & : y := 0 \\
\text{c} & : y := 1; \\
\text{d} & : d := x \quad // 0
\end{align*}
\]

Not linearisable
Why Not Linearisability?

- Assumes $\leq_{time}$ order -- not present under WMC
- Requires total order on all events -- not always possible under WMC

Per-location linearisability?
- Cannot model weak specs, e.g. C11, TSO

```
x:=1;  b:=y  // 0
y:=1;  d:=x  // 0
```

(not linearisable)

```
x:=1;  b:=y  // 0
y:=1;  d:=x  // 0
```

(SB)

(not linearisable)
Our Solution

✔ no particular memory model
✔ no \(<_{\text{time}}\) order
✔ no total order on events
✔ per-library specification
  ➡ set of library executions

\[ \{ G \mid G \text{ satisfies certain axioms} \} \]
  library execution
**Example: Queue Library**

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)  // may read 1 or empty (∅)

a := pop(s);
if (a == 2)
    b := deq(q)
```

\[ G_{queue} = < E, po, so, hb > \]
Example: Queue Library

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)

a := pop(s);
if (a == 2)
    b := deq(q) // may read 1 or empty (⊥)
```

\[
G_{\text{queue}} = \langle E, \text{po}, \text{so}, \text{hb} \rangle
\]

Diagram:

- **n** new-queue(q)
- **e** enq(q, 1)
- **d** deq(q, 1)
Example: **Queue** Library

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)

a := pop(s);
if (a == 2)
  b := deq(q)  // may read 1 or empty (⊥)
```

**G**

\[ G_{queue} = < \{E\}, po, so, hb > \]
Example: **Queue** Library

```plaintext
q:=new-queue();
s:=new-stack();

enq(q, 1);
push(s, 2)

a:=pop(s);
if(a==2)
  b:=deq(q) // may read 1 or empty (⊥)
```

**G_{queue} = < \(E\), \(po\), \(so\), \(hb\) >**
Library Executions

Example: **Queue** Library

```plaintext
q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2)
a := pop(s);
if (a == 2)
  b := deq(q)  // may read 1 or empty (∅)
```

---

**G**\(_{\text{queue}}\) = < \(\mathcal{E}, \text{po}, \text{so}, \text{hb}\) >

- **Events**: \(\mathcal{E}\)
- **Program-Order**: \(\text{po}\)
- **Synchronisation-Order**: \(\text{so}\)
- **Heap**: \(\text{hb}\)

---

- **new-queue(q)**
- **enq(q, 1)**
- **deq(q, 1)**
- **new-queue(q)**
- **enq(q, 1)**

---

17
**Library Executions**

Example: *Queue* Library

\[
q := \text{new-queue}(); \\
s := \text{new-stack}(); \\
\text{enq}(q, 1); \\
\text{push}(s, 2) \\
\text{a} := \text{pop}(s); \\
\text{if}(\text{a} == 2) \\
\text{b} := \text{deq}(q) // \text{may read 1 or empty (⊥)}
\]

\[
G_{\text{queue}} = < \text{E}, \text{po}, \text{so}, \text{hb} >
\]

- **Events**
- **Program-Order**
- **Happens-Before Order**
- **Synchronisation-Order**

**Diagram**

- **new-queue(q)**
  - **e** enq(q, 1)  \(\xrightarrow{\text{po}}\)  **d** deq(q, 1)  \(\xrightarrow{\text{so}}\)  **n** new-queue(q)
  - **n** new-queue(q)  \(\xrightarrow{\text{po}}\)  **e** enq(q, 1)  \(\xrightarrow{\text{po}}\)  **d** deq(q, 1)  \(\xrightarrow{\text{po}}\)  **n** new-queue(q)  \(\xrightarrow{\text{po}}\)  **d** deq(q, ⊥)
Example: *Queue* Library

\[
q := \text{new-queue}(); \\
s := \text{new-stack}(); \\
enq(q, 1); \\
push(s, 2); \\
a := \text{pop}(s); \\
\text{if}(a == 2) \\
b := \text{deq}(q) // may\ read\ 1\ or\ empty\ (\bot)
\]

\[
G_{\text{queue}} = < E, po, so, hb > \\
\text{events} \hspace{2cm} \text{program-order} \hspace{2cm} \text{happens-before\ order} \\
\text{happens-before\ order}\ \ (\text{coming\ soon!}) \hspace{2cm} \text{synchronisation-order}
\]

```
\text{new-queue}(q) \hspace{2cm} \text{new-queue}(q) \\
\text{enq}(q, 1) \hspace{2cm} \text{enq}(q, 1) \\
\text{deq}(q, 1) \hspace{2cm} \text{deq}(q, \bot)
```
Library Executions

Example: *Queue* Library

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)
```

```plaintext
a := pop(s);
if(a == 2)
b := deq(q) // may read 1 or empty (∅)
```

How to eliminate this "incorrect" behaviour?

```
new-queue(q)
```

```
new-queue(q)
```

```
enq(q, 1)
```

```
deco(q, 1)
```

```
enq(q, 1)
```

```
deco(q, ⊥)
```

Program Executions

\[ [P] = \left\{ G_P = < E, po, so > \mid \ldots \right\} \]

- semantics of \( P \)
- execution of \( P \)
- events
- program-order
- synchronisation-order
  (library-specific)
Program Executions

\[ \boxed{P} = \left\{ G_P = < E, \text{po}, \text{so} > \mid \cdots \right\} \]

semantics of \( P \)

execution of \( P \)

q:=new-queue();
s:=new-stack();

enq(q,1);
push(s,2)

a:=pop(s);
if(a==2)
b:=deq(q) // should return 1
Program Executions

\[ [P] = \{ G_P = \langle E, \text{po}, \text{so} \rangle \mid \ldots \} \]

- semantics of \( P \)
- execution of \( P \)

```
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2);
a := pop(s);
if (a == 2)
    b := deq(q) // should return 1
```
Program Executions

\[ [P] = \{ G_P = \langle E, po, so \rangle \mid \cdots \} \]

semantics of \( P \)

execution of \( P \)

\( hb = (po \cup so)^+ \)

\begin{align*}
q &= \text{new-queue}(); \\
& \text{enq}(q, 1); \\
& \text{push}(s, 2); \\
\end{align*}

\begin{align*}
so &= \text{new-stack}(); \\
& a := \text{pop}(s); \\
& \text{if}(a == 2) \\
& b := \text{deq}(q) // \text{should return } 1
\end{align*}

\begin{align*}
a &= \text{push}(s, 2) \\
d &= \text{deq}(q, 1)
\end{align*}
Program Executions

\[ [P] = \{ G_P = < E, po, so > \mid \ldots \} \]

semantics of \( P \)
execution of \( P \)

\[ hb = (po \cup so)^+ \]

\[ q := \text{new-queue}(); \]
\[ s := \text{new-stack}(); \]
\[ \text{enq}(q, 1); \]
\[ \text{push}(s, 2) \]
\[ \text{a} := \text{pop}(s); \]
\[ \text{if}(a == 2) \]
\[ \text{b} := \text{deq}(q) \] // should return 1

Diagram:

- \( q \): new-queue(q)
- \( s \): new-stack(s)
- \( e \): enq(q, 1)
- \( r \): pop(s, 2)
- \( a \): push(s, 2)
- \( d \): deq(q, 1)
- \( f \): deq(q, ⊥)

Edges:
- \( po \)
- \( so \)
- \( hb \)
Program Executions

$$[[P]] = \{ G_P = < E, po, so > \mid \cdots \}$$

allow libraries to constrain each other via $hb$!

How? $hb$ defined on program executions
From **Program** to **Library** Executions

\[ G_P = \langle E, po, so \rangle \]

**hb** = \((po \cup so)^+\)

```
q new-queue(q)
  \[ po \]

s new-stack(s)
  \[ po \]

```

```
enq(q, 1)
  \[ po \]

```

```
push(s, 2)
  \[ po \]

```

```
pop(s, 2)
  \[ po \]

```

```
deq(q, 1)
  \[ po \]

```

```
new-queue(q)

push(s, 2)
```

```
new-stack(s)
```

```
enq(q, 1)
```

```
pop(s, 2)
```

```
deq(q, 1)
```
From **Program** to **Library** Executions

$$G_P = \langle E, \text{po}, \text{so} \rangle$$

$$E_{queue} \cup E_{stack}$$

$$hb = (\text{po} \cup \text{so})^+$$

Diagram:

- **q** new-queue(q)
- **s** new-stack(s)
- **e** enq(q, 1)
- **a** push(s, 2)
- **r** pop(s, 2)
- **d** deq(q, 1)

Arrows indicate the execution order and operations.
From *Program* to *Library* Executions

\[ G_P = \langle E, \text{po}, \text{so} \rangle \]

\[ G_{\text{queue}} \oplus G_{\text{stack}} \]

\[ E_{\text{queue}} \cup E_{\text{stack}} \]

\[ hb = (\text{po} \cup \text{so})^+ \]
From **Program** to **Library** Executions

\[ G_P = < E, po, so > \]

\[ G_{queue} \oplus G_{stack} \]

\[ E_{queue} \cup E_{stack} \]

\[ hb = (po \cup so)^+ \]

\[
\begin{align*}
G_{queue} &= < E_{queue}, > \\
G_{stack} &= < E_{stack}, >
\end{align*}
\]
From **Program** to **Library** Executions

\[ G_P = < E, po, so > \]

\[ G_{queue} \oplus G_{stack} \]

\[ E_{queue} \cup E_{stack} \]

\[ hb = (po \cup so)^{+} \]

\[ G_{queue} = < E_{queue}, po_{queue}, > \]
From *Program* to *Library* Executions

\[ G_P = < E, po, so > \]

\[ G_{\text{queue}} \oplus G_{\text{stack}} \]

\[ E_{\text{queue}} \cup E_{\text{stack}} \]

\[ hb = ( po \cup so )^+ \]

\[ G_{\text{queue}} = < E_{\text{queue}}, po_{\text{queue}}, so_{\text{queue}}, > \]
From **Program** to **Library** Executions

\[ G_P = < E, po, so > \]

\[ G_{queue} \oplus G_{stack} \]

\[ E_{queue} \cup E_{stack} \]

\[ hb = (po \cup so)^+ \]

\[ G_{queue} = < E_{queue}, po_{queue}, so_{queue}, hb> \]

\[ hb_{queue} \]
Example Revisited

```
q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2)
a := pop(s);
if (a == 2)
b := deq(q) // should return 1
```

```
enq(q, 1);
push(s, 2)
a := pop(s);
if (a == 2)
b := deq(q) // should return 1
```
Example Revisited

```
q:=new-queue();
s:=new-stack();

enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```

```
enq(q, 1);
push(s, 2)  // should return 1
```
Example Revisited

q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2) || a := pop(s);
if (a == 2)
b := deq(q) // should return 1

How does \textit{hb} exclude the \textbf{RHS} execution?

\textcolor{green}{\Longrightarrow} \textbf{library axioms!}

\begin{itemize}
\item e \text{ enq}(q, 1)
\item r \text{ pop}(s, 2)
\item d \text{ deq}(q, 1)
\end{itemize}
Queue Axioms

\[ G = < E, po, so, hb > \] is a consistent queue execution iff:

1. \( E \) contains queue events

\[
\begin{align*}
\text{q} & \quad \text{new-queue}(q) \\
\text{e} & \quad \text{enq}(q, v) \\
\text{d} & \quad \text{deq}(q, w) \\
\text{f} & \quad \text{deq}(q, \bot) \\
\end{align*}
\]

\( v, w \in \text{Val} \)
Queue Axioms

$G = < E, po, so, hb >$ is a consistent queue execution iff:

1. $E$ contains queue events

   - \text{new-queue(q)}
   - \text{enq(q, v)}
   - \text{deq(q, w)}
   - \text{deq(q, ⊥)}

   $v, w \in \text{Val}$

2. so is \textbf{1-to-1}

   so relates \textit{matching} \text{enq/deq} events;

   \begin{align*}
   \text{if} & \quad \text{enq(q, v)} \quad \text{so} \quad \text{deq(q, w)} \\
   \text{then} & \quad v = w
   \end{align*}
Queue Axioms

\[ G = < E, po, so, hb > \] is a consistent \textit{queue} execution iff:

1. \( E \) contains queue events

\[
\begin{align*}
\text{q} & \quad \text{new-queue(q)} \\
\text{e} & \quad \text{enq}(q, v) \\
\text{d} & \quad \text{deq}(q, w) \\
\text{f} & \quad \text{deq}(q, \bot) \\
\end{align*}
\]

\( v, w \in \text{Val} \)

2. \( so \) is \textit{1-to-1}

\( so \) relates \textit{matching} \( \text{enq/deq} \) events;

\[
\begin{align*}
\text{if} \quad & \quad \text{enq}(q, v) & \quad \text{so} & \quad \text{deq}(q, w) \\
\text{then} \quad & \quad v = w \\
\text{for all} \quad & \quad v \\
\end{align*}
\]
Queue Axioms

\( G = < E, po, so, hb > \) is a consistent \textit{queue} execution iff:

1. \( E \) contains queue events

\[
\begin{align*}
\text{new}-\text{queue}(q) & \quad \text{enq}(q, v) & \quad \text{deq}(q, w) & \quad \text{deq}(q, \perp)
\end{align*}
\]

\( v, w \in \text{Val} \)

2. \( so \) is \textbf{1-to-1}

\( so \) relates \textit{matching} \textit{enq}/\textit{deq} events;

\[
\begin{align*}
\text{if} & \quad \text{enq}(q, v) \quad \text{so} \quad \text{deq}(q, w) & \quad \text{then} & \quad v = w \\
\text{enq}(q, v) \quad \text{so} \quad \text{deq}(q, \perp) & \quad \text{for all} & \quad v
\end{align*}
\]

3. \( \exists \) \textit{to}. \textit{to totally} orders \( E \),

\( hb \subseteq \text{to} \quad \text{and} \quad \text{to} \) is a FIFO sequence
Example Revisited

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)

// should return 1
a := pop(s);
if (a == 2)
b := deq(q)
```

\( hb = (po \cup so)^+ \)
Example Revisited

```
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)  // should return 1
```

\[
hb = (po \cup so)^+\]
Example Revisited

```plaintext
q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2)  // a := pop(s);
if(a == 2)
  b := deq(q)  // should return 1
```

$hb = (po \cup so)^+$

$hb \subseteq to$
Example Revisited

```
q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2)
a := pop(s);
if (a == 2)
    b := deq(q)  // should return 1
```

```
hb = (po ∪ so)⁺
```

```
hb ⊆ to
```
Example Revisited

```plaintext
q := new-queue();
s := new-stack();
enq(q, 1);
push(s, 2);  // should return 1

a := pop(s);
if (a == 2)
    b := deq(q)
```

Diagram:

- **q** (new-queue(q))
- **e** (enq(q, 1))
- **a** (push(s, 2))
- **r** (pop(s, 2))
- **f** (deq(q, ⊥))

Arrows:
- `hb`, `po` from **q** to **e**
- `hb`, `po` from **e** to **a**
- `hb`, `po` from **a** to **f**
- `hb`, `po` from **f** to **q**
- `po` from **e** to **r**
- `po` from **r** to **f**
- `so` from **f** to **e**

Equations:

- \( hb = (po \cup so)^+ \)
- \( hb \subseteq to \)
Program Executions

\[
[P] = \left\{ G_P = <E, po, so> \mid \cdots \right\}
\]

\[
q := \text{new-queue}();
\text{s := new-stack}();
enq(q, 1); \parallel a := \text{pop}(s);
push(s, 2) \parallel \text{if}(a == 2)
\quad \text{b := deq}(q)
\]
Program Executions

\[ \{ \] = \{ G_P = \langle E, po, so \rangle \}

\begin{align*}
q &:= \text{new-queue}(); \\
s &:= \text{new-stack}(); \\
\text{enq}(q, 1); &| a := \text{pop}(s); \\
\text{push}(s, 2) &| \text{if}(a == 2) \\
\end{align*}

b := \text{deq}(q)
Program Executions

\[ P \] = \{ G_P = \langle E, po, so \rangle \mid \cdots \}

queue & stack calls

\[
\begin{align*}
q &:= \text{new-queue}(); \\
s &:= \text{new-stack}(); \\
\text{enq}(q,1); &\quad a := \text{pop}(s); \\
\text{push}(s,2) &\quad \text{if}(a==2) \\
&\quad \text{b} := \text{deq}(q)
\end{align*}
\]
Program Executions

\[ [P] = \left\{ G_P = \langle E, po, so \rangle \right\} \]

queue & stack calls

\begin{align*}
q & := \text{new-queue}(); \\
s & := \text{new-stack}(); \\
enq(q, 1); & \quad a := \text{pop}(s); \\
push(s, 2) & \quad \text{if}(a == 2) \\
& \quad b := \text{deq}(q)
\end{align*}

G_{queue} \text{ sats. queue axioms} \\
G_{stack} \text{ sats. stack axioms}
Program Executions

$$\mathbf{[P]} = \{ G_P = \langle E, p_0, s_0 \rangle \mid G_{\text{queue}} \text{ sats. queue axioms} \quad G_{\text{stack}} \text{ sats. stack axioms} \}$$

Queue & Stack calls

- q:=new-queue();
- s:=new-stack();
- enq(q,1);
- push(s,2)
- a:=pop(s);
- if(a==2)
  - b:=deq(q)

$$P \text{ calls } L_1 \ldots L_n \text{ and } G \in \mathbf{[P]} \iff G_{L_1} \text{ sats. } L_1 \text{ axioms}$$

... 

$$G_{L_n} \text{ sats. } L_n \text{ axioms}$$
Queue Axioms

\[ G = \langle E, po, so, hb \rangle \] is a consistent **queue** execution iff:

1. \( E \) contains queue events
2. \( so \) is **1-to-1**; \( so \) relates **matching** enq/deq events
3. \( \exists \, to. \, to \text{ totally} \) orders \( E \),
   \[ hb \subseteq to \quad \text{and} \quad to \text{ is a FIFO sequence} \]

\[ \times \text{ too strong} \]
\[ \times \text{ difficult to find } to \text{ witness} \]
Strong Queue Axioms
C11 Herlihy-Wing Queue Implementation

**new-queue()** \(\triangleq\)
let \(q = alloc(+\infty)\) in \(q\)

**enq(q, v)** \(\triangleq\)
let \(i = fetch-add(q, 1, \text{rel})\) in
store\((q + i + 1, v, \text{rel})\);

**deq(q)** \(\triangleq\)
loop
let \(range = load(q, \text{acq})\) in
for \(i = 1\) to \(range\) do
  let \(x = atomic-xchg(q + i, 0, \text{acq})\) in
  if \(x \neq 0\) then break

\(\times\) does not satisfy **strong** queue axioms
**Strong** Queue Axioms

C11 Herlihy-Wing Queue Implementation

\[\begin{align*}
\text{new-queue}() & \triangleq \\
& \quad \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) & \triangleq \\
& \quad \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
& \quad \text{store}(q + i + 1, v, \text{rel});
\end{align*}\]

\[\begin{align*}
\text{deq}(q) & \triangleq \\
& \quad \text{loop} \\
& \quad \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
& \quad \text{for } i = 1 \text{ to } range \text{ do } \\
& \quad \quad \text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in } \\
& \quad \quad \text{if } x \neq 0 \text{ then break}_2 x
\end{align*}\]

✔ satisfies **strong** queue axioms
Weak Queue Axioms

\[ G = < E, po, so, hb > \] is a consistent weak queue execution iff:

1. \( E \) contains queue events
2. \( so \) is 1-to-1; \( so \) relates matching enq/deq events
3. \( deq \) with \textbf{hb-earlier unmatched} enq cannot return \( \bot \)

\textbf{Diagram:}

- \( e \): \text{enq}(q, v)
- \( d \): \text{deq}(q, w)

\text{if} \quad \text{enq}(q, v) \xrightarrow{hb} \text{deq}(q, w) \quad \text{then} \quad w \neq \bot
Example Revisited

\[
\begin{align*}
q &:= \text{new-queue}() \\
s &:= \text{new-stack}() \\
enq(q, 1); &\quad a := \text{pop}(s) \\
push(s, 2); &\quad \text{if (} a == 2 \text{)} \quad b := \text{deq}(q) \quad \text{// should return 1}
\end{align*}
\]

```
enq(q, 1);
push(s, 2);
a := \text{pop}(s);
if (a == 2)
  b := \text{deq}(q)
```

```c
q := new-queue();
s := new-stack();
// should return 1
```
Example Revisited

```plaintext
define_new_queue_qs
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)

a := pop(s);
if (a == 2)
    b := deq(q)  // should return 1
```

Diagram:
- `q` new-queue(q)
- `s` new-stack(s)
- `e` enq(q, 1)
- `a` push(s, 2)
- `r` pop(s, 2)
- `d` deq(q, 1)
Example Revisited

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)
```

```plaintext
a := pop(s);
if (a == 2)
  b := deq(q) // should return 1
```

```plaintext
enq(q, 1);
push(s, 2)
```

```plaintext
a := pop(s);
if (a == 2)
  b := deq(q) // should return 1
```
Example Revisited

\[
q := \text{new-queue}(); \\
s := \text{new-stack}(); \\
\text{enq}(q, 1); \quad \text{if}(a == 2) \quad b := \text{deq}(q) \quad \text{// should return 1}
\]

```plaintext
e := \text{enq}(q, 1); \\
\text{push}(s, 2) \\
\text{pop}(s, 2); \\
\text{deq}(q, 1)
```

```plaintext
\text{enq}(q, 1); \\
\text{push}(s, 2) \\
\text{pop}(s, 2); \\
\text{deq}(q, 1)
```
Weak Queue Axioms

$G = <E, po, so, hb>$ is consistent a weak queue execution iff:

1. $E$ contains queue events
2. $so$ is 1-to-1; $so$ relates matching $enq/deq$ events
3. $deq$ with $hb$-earlier unmatched $enq$ cannot return $\bot$
   
   \[ if \quad \begin{array}{c}
   e \quad enq(q, v) \\
   \downarrow so \\
   \end{array} \quad ^{hb} \quad \begin{array}{c}
   d \quad deq(q, w) \\
   \end{array} \quad then \quad w \neq \bot \]

4. weak FIFO guarantee

\[ \begin{array}{c}
   e \quad enq(q, v) \\
   \downarrow so \\
   \end{array} \quad ^{hb} \quad \begin{array}{c}
   a \quad enq(q, w) \\
   \downarrow so \\
   \end{array} \quad \begin{array}{c}
   d \quad deq(q, v) \\
   \downarrow so \\
   \end{array} \quad ^{hb} \quad \begin{array}{c}
   r \quad deq(q, w) \\
   \end{array} \]
**Weak vs. Strong Queue Axioms**

- **Not** equivalent

  \[ \Rightarrow \text{C11 Herlihy-Wing queue} \]

\[
\begin{align*}
\text{new-queue}() & \triangleq \\
& \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) & \triangleq \\
& \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
& \text{store}(q + i + 1, v, \text{rel}); \\
\text{deq}(q) & \triangleq \\
& \text{loop} \\
& \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
& \text{for } i = 1 \text{ to } range \text{ do } \\
& \text{let } x = \text{atomic-xchg}(q + i, 0,\text{ acq}) \text{ in } \\
& \text{if } x \neq 0 \text{ then } \text{break}_2 x
\end{align*}
\]

- **✔ satisfies weak axioms**

- **✘ does not satisfy strong axioms**
**Weak vs. Strong** Queue Axioms

- **Not** equivalent

  $\Rightarrow$ C11 *Herlihy-Wing* queue

\[
\begin{align*}
\text{new-queue}(q) &\triangleq \\
&\quad \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) &\triangleq \\
&\quad \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
&\quad \text{store}(q + i + 1, v, \text{rel}); \\
\text{deq}(q) &\triangleq \\
&\quad \text{loop} \\
&\quad \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
&\quad \text{for } i = 1 \text{ to } range \text{ do } \\
&\quad \quad \text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in } \\
&\quad \quad \text{if } x \neq 0 \text{ then break}_2 x
\end{align*}
\]

- ✔ satisfies **weak** axioms
- ✔ satisfies **strong** axioms
Weak vs. Strong Queue Axioms

- **Not** equivalent

\[ \Rightarrow \text{C11 Herlihy-Wing queue} \]

\[
\begin{align*}
\text{new-queue}() & \triangleq \\
& \quad \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) & \triangleq \\
& \quad \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
& \quad \text{store}(q + i + 1, v, \text{rel}); \\
\text{deq}(q) & \triangleq \\
& \quad \text{loop} \\
& \quad \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
& \quad \text{for } i = 1 \text{ to } range \text{ do } \\
& \quad \quad \text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in } \\
& \quad \quad \text{if } x \neq 0 \text{ then break}_2 x
\end{align*}
\]

- Why weak axioms?

\[ \Rightarrow \text{strong enough for certain uses: single-producer-single-consumer} \]
**Alternative Strong Queue Axioms**

\[ G = < E, po, so, hb > \] is a consistent queue execution iff:

1. \( E \) contains queue events
2. \( so \) is 1-to-1; \( so \) relates matching \( \text{enq/deq} \) events

3. \( \exists \) \( to \). \( to \) totally orders \( E \),  
\( hb \subseteq to \) \( to \) is FIFO

4. \( \text{weak} \) FIFO guarantee  
(\( \text{weak acyclicity} \) axiom)
Alternative Strong Queue Axioms

$G = \langle E, po, so, hb \rangle$ is a consistent queue execution iff:

1. $E$ contains queue events
2. so is 1-to-1; so relates matching enq/deq events
3. $\exists$ to. to totally orders $E$, $hb \subseteq to$. to is FIFO

- too strong $\rightarrow$ weak axioms
- difficult to find to witness

4. weak FIFO guarantee
   (weak acyclicity axiom)
**Alternative Strong** Queue Axioms

\[ G = < E, po, so, hb > \] is a consistent queue execution iff:

1. \( E \) contains queue events
2. \( so \) is 1-to-1; \( so \) relates matching enq/deq events
3. \( \exists \) to. to totally orders \( E \),
   \[ \text{hb} \subseteq \text{to} \quad \text{to is FIFO} \]
   - too strong \( \blacktriangleright \) weak axioms
   - difficult to find to witness
4. \( \text{deq with hb-earlier unmatched} \)
   \( \text{enq cannot return } \bot \)
5. **strong** FIFO guarantee
   \( \text{(strong acyclicity axiom)} \)
   ⇒ see our paper!
6. **weak** FIFO guarantee
   \( \text{(weak acyclicity axiom)} \)
A declarative framework:

✓ **Agnostic** to memory model
  ➡ support both SC and WMC specs

✓ **General**
  ➡ port existing SC *(linearisability)* specs
  ➡ port existing WMC specs (e.g. C11, TSO)
  ➡ built from the **ground up**: assume no pre-existing libraries or specs
Recap

A declarative framework:

✓ **Agnostic** to memory model
  ➡ support *both* SC and WMC specs

✓ **General**
  ➡ port *existing* SC (*linearisability*) specs
  ➡ port *existing* WMC specs (e.g. C11, TSO)
  ➡ built from the *ground up*: assume *no pre-existing* libraries or specs

? **Compositional**
  ? verify *client programs*
  ? verify *library implementations*

```plaintext
q := new-queue();
s := new-stack();

enq(q, 1);
push(s, 2)

a := pop(s);
if (a == 2)
  b := deq(q)
```

Diagram:

- Elimination-Stack
- ExchArray
- Weak-Stack
- LockedQ
- HW-Queue
- Exchanger
- Mutex
- MRSW-Lock

C11
Part II.

Concurrent Library *Verification* under WMC
Program Outcomes

\[ \llbracket P \rrbracket = \{ G_P \mid ... \} \]

\[ \text{outcomes}(P) = \{ \text{val}(G_P) \mid G_P \in \llbracket P \rrbracket \} \]

values of local variables
Program Outcomes

\[ [P] = \{ G_P \mid \ldots \} \]

\[ \text{outcomes}(P) = \{ \text{val}(G_P) \mid G_P \in [P] \} \]

values of local variables

\[
q := \text{new-queue}();
\]
\[
s := \text{new-stack}();
\]
\[
enq(q, 1);
\]
\[
push(s, 2)
\]
\[
a := \text{pop}(s);
\]
\[
\text{if}(a == 2)
\]
\[
b := \text{deq}(q)
\]
```
\[
[P] = \{ G_P | ... \}
\]

**Client Verification**

\[
\text{assert } (\text{true})
\]

\[
\begin{align*}
q &:= \text{new-queue}(); \\
s &:= \text{new-stack}(); \\
enq(q, 1); &\quad a := \text{pop}(s); \\
push(s, 2) &\quad \text{if}(a == 2) \\
&\quad b := \text{deq}(q)
\end{align*}
\]

**assert** \((\neg (a = 2 \land b = \perp))\)

\[
\text{outcomes}(P) = \{ \text{val}(G_P) | G_P \in [P] \}
\]

values of *local* variables
**Implementation Verification?**

\[
\begin{align*}
\text{new-queue}() & \triangleq \\
& \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) & \triangleq \\
& \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
& \text{store}(q + i + 1, v, \text{rel}) \\
\text{deq}(q) & \triangleq \\
& \text{loop} \\
& \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
& \text{for } i = 1 \text{ to } range \text{ do } \\
& \quad \text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in } \\
& \quad \text{if } x \neq 0 \text{ then break}_2 x
\end{align*}
\]
**Library Implementation**

\[ I_{\text{queue}} : \text{Method}_{\text{queue}} \rightarrow \text{Prog} \]

**C11** Herlihy-Wing Implementation (HWQ)

\[
\begin{align*}
\text{new-queue}() & \triangleq \\
& \text{let } q = \text{alloc}(\infty) \text{ in } q \\
\end{align*}
\]

\[
\begin{align*}
\text{enq}(q,v) & \triangleq \\
& \text{let } i = \text{fetch-add}(q,1,\text{rel}) \text{ in } \\
& \text{store}(q+i+1,v,\text{rel});
\end{align*}
\]

\[
\begin{align*}
\text{deq}(q) & \triangleq \\
& \text{loop} \\
& \text{let } \text{range} = \text{load}(q,\text{acq}) \text{ in } \\
& \text{for } i = 1 \text{ to } \text{range} \text{ do } \\
& \text{let } x = \text{atomic-xchg}(q+i,0,\text{acq}) \text{ in } \\
& \text{if } x \neq 0 \text{ then break}_2 x
\end{align*}
\]
Translation

\[
I_{queue} : \text{Prog} \rightarrow \text{Prog}
\]
Implementation **Soundness**

$I_{queue}$ sound $\iff$ for all $P$:

\[
\text{outcomes}(\left\lfloor P \right\rfloor_{I_{queue}}) \subseteq \text{outcomes}(P)
\]
Implementation **Soundness**

$I_{queue} \text{ sound } \iff \text{ for all } P :$

\[
\text{outcomes}(\bigcup P \bigcap I_{queue}) \subseteq \text{outcomes}(P)
\]

\[
\{ \text{val}(G_i) \mid G_i \in \bigcup P \bigcap I_{queue} \} \subseteq \{ \text{val}(G_s) \mid G_s \in \llbracket P \rrbracket \}
\]

\[
\text{outcomes}(P) = \{ \text{val}(G_P) \mid G_P \in \llbracket P \rrbracket \}
\]
**Implementation Soundness**

$I_{\text{queue}}$ **sound** ⇔ for all $P$:

\[ \text{outcomes}(\downarrow P \downarrow I_{\text{queue}}) \subseteq \text{outcomes}(P) \]

\[ \{ \text{val}(G_i) \mid G_i \in \downarrow P \downarrow I_{\text{queue}} \} \subseteq \{ \text{val}(G_s) \mid G_s \in \downarrow P \} \]

\[ \forall G_i \in \downarrow P \downarrow I_{\text{queue}}. \exists G_s \in \downarrow P. \text{val}(G_i) = \text{val}(G_s) \]

\[ \text{outcomes}(P) = \{ \text{val}(G_P) \mid G_P \in \downarrow P \} \]
Example: HWQ Soundness

Given $G_i \in \llbracket \llbracket P \rrbracket I_{\text{queue}} \rrbracket$

Find $G_s \in \llbracket P \rrbracket$

such that: $\text{val}(G_i) = \text{val}(G_s)$
Example: HWQ Soundness

\[
P \models I_{\text{queue}}
\]

\[
q := I_{\text{queue}}(\text{new-queue}()); \\
s := \text{new-stack}(); \\
I_{\text{queue}}(\text{enq}(q, 1)); \quad | \quad a := \text{pop}(s); \\
\text{push}(s, 2)
\]

\[
P \models I_{\text{queue}}
\]

\[
q := \text{new-queue}(); \\
s := \text{new-stack}(); \\
\text{enq}(q, 1); \quad | \quad a := \text{pop}(s); \\
\text{push}(s, 2) \quad | \quad \text{if}(a == 2) \\
\quad b := \text{deq}(q)
\]

Given \( G_i \in I_{\text{queue}} \) \( \models P \)

\[\frac{\text{such that: } \ \text{val}(G_i) = \text{val}(G_s)}{G_{L_1} \ \text{sats. } \ L_1 \ \text{axioms}}\]

\[\text{...}\]

\[G_{L_n} \ \text{sats. } \ L_n \ \text{axioms}\]
Example: HWQ Soundness

Given $G_i \in \mathcal{P}_{\text{queue}}$

We know:

- $(G_i)_{C11}$ sats. $C11$ axioms
- $(G_i)_{\text{stack}}$ sats. $\text{stack}$ axioms

Find $G_s \in \mathcal{P}$

Show:

- $(G_s)_{\text{queue}}$ sats. $\text{queue}$ axioms
- $(G_s)_{\text{stack}}$ sats. $\text{stack}$ axioms
Example: HWQ Soundness

Given $G_i \in \mathcal{P}_{\text{stack}}$.

Find $G_s \in \mathcal{P}$.

We know:

$(G_i)_{C11}$ sats. $C11$ axioms

$(G_i)_{\text{stack}}$ sats. $\text{stack}$ axioms

Show:

$(G_s)_{\text{queue}}$ sats. $\text{queue}$ axioms

$(G_s)_{\text{stack}}$ sats. $\text{stack}$ axioms

main proof obligation
Example: HWQ Soundness

**Given** \( G_i \in \left[ \lesssim P \right]_{queue} \)

![Diagram showing a queue and stack with operations: push, pop, new-stack, new-queue, enq, deq.]

**Find** \( G_s \in \left[ P \right] \)

![Diagram showing a queue and stack with operations: push, pop, new-stack, new-queue, enq, deq.]

We know:
- \((G_i)_{C11}\) sats. \(C11\) axioms
- \((G_i)_{stack}\) sats. \(stack\) axioms

Show:
- \((G_s)_{queue}\) sats. \(queue\) axioms
- \((G_s)_{stack}\) sats. \(stack\) axioms

*almost redundant*:
- \((G_i)_{stack} =?= (G_s)_{stack}\)

*main* proof obligation
Example: HWQ Soundness

We know:

\[(G_i)_{\text{C11}} \text{sats. C11 axioms}\]
\[(G_i)_{\text{stack}} \text{sats. stack axioms}\]

Find \(G_s \in [P]\)

\[\text{main proof obligation}\]
\[(G_s)_{\text{queue}} \text{sats. queue axioms}\]
\[(G_s)_{\text{stack}} \text{sats. stack axioms}\]

Wanted \textit{compositional} soundness proof

\[\text{almost redundant}: (G_i)_{\text{stack}} = (G_s)_{\text{stack}}\]
Example: HWQ Soundness

Given \( G_i \in \left[ \bigcap P \right]_{\text{queue}} \)

Find \( G_s \in \left[ P \right] \)

\[
(G_i)_{\text{stack}} = <E_{\text{stack}}, po_{\text{stack}}, so_{\text{stack}}, (hb_i)_{\text{stack}}>

(G_s)_{\text{stack}} = <E_{\text{stack}}, po_{\text{stack}}, so_{\text{stack}}, (hb_s)_{\text{stack}}>

\]
Example: HWQ Soundness

Given \( G_i \in \bigcap \{ P \mid I_{queue} \} \)

Find \( G_s \in \{ P \} \)

\((G_i)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_i)_{stack} \rangle\)

\((G_s)_{stack} = \langle E_{stack}, po_{stack}, so_{stack}, (hb_s)_{stack} \rangle\)
Example: HWQ Soundness

Given $G_i \in \mathbb{L}P \mathbb{L}_{\text{queue}}$

$$\begin{align*}
(G_i)_{\text{stack}} &= \langle E_{\text{stack}}, p_{\text{stack}}, s_{\text{stack}}, (h_{\text{bi}})_{\text{stack}} \rangle \\
(G_s)_{\text{stack}} &= \langle E_{\text{stack}}, p_{\text{stack}}, s_{\text{stack}}, (h_{\text{bs}})_{\text{stack}} \rangle
\end{align*}$$

Find $G_s \in \mathbb{L}P$

$$h_{bs} = (h_{b})_{\text{stack}} = ( ( p_{o} \cup s_{o} )^+ )_{\text{stack}}$$

show: $h_{bi} = h_{bs}$
Example: HWQ Soundness

\[ (G_i)_{\text{stack}} = \langle E_{\text{stack}}, p_{\text{stack}}, s_{\text{stack}}, (hb_i)_{\text{stack}} \rangle \]

\[ (G_s)_{\text{stack}} = \langle E_{\text{stack}}, p_{\text{stack}}, s_{\text{stack}}, (hb_s)_{\text{stack}} \rangle \]

**show:** \( hb_i = hb_s \)

\[ hb_s = (hb)_{\text{stack}} = ( (p_{\text{stack}} \cup s_{\text{queue}} \cup s_{\text{stack}} )^+ )_{\text{stack}} \]
Example: HWQ Soundness

Given $G_i \in \mathcal{P} \mathcal{I}_{\text{queue}}$

Find $G_s \in \mathcal{P}$

$\begin{align*}
(G_i)_{\text{stack}} &= <E_{\text{stack}}, po_{\text{stack}}, so_{\text{stack}}, (hb_i)_{\text{stack}} > \\
(G_s)_{\text{stack}} &= <E_{\text{stack}}, po_{\text{stack}}, so_{\text{stack}}, (hb_s)_{\text{stack}} >
\end{align*}$

show: $hb_i = hb_s$

sufficient: $G_s \cdot so_{\text{queue}} \subseteq G_i \cdot hb_i$

$hbs = (hb)_{\text{stack}} = ( (po \cup so_{\text{queue}} \cup so_{\text{stack}})^+ )_{\text{stack}}$
Local Soundness

$I_{queue}$ sound $\iff$ for all $P$:

$\forall G_i \in \llbracket P \rrbracket_{I_{queue}}. \exists G_s \in \llbracket P \rrbracket. \text{val}(G_i) = \text{val}(G_s)$
**Local Soundness**

\[ I_{queue} \text{ sound } \iff \forall P : \exists G_s \in (P) . \forall G_i \in \llbracket P \rrbracket_{I_{queue}} . \exists G_s \in (P) . \text{val}(G_i) = \text{val}(G_s) \]

(when \( P \) calls \( \text{queue}, L_1, ..., L_n \))
**Local Soundness**

(when $P$ calls `queue, L1, ..., Ln)

$I_{\text{queue}}$ sound $\iff$ for all $P$:

\[
\forall G_i \in \llbracket P \rrbracket_{I_{\text{queue}}} . \ \exists G_s . \ \text{val}(G_i) = \text{val}(G_s)
\]

$(G_s)_{\text{queue}}$ sats. `queue axioms

$(G_s)_{L1}$ sats. `L1 axioms

$\ldots$

$(G_s)_{Ln}$ sats. `Ln axioms

$I_{\text{queue}}$ locally sound $\iff$ for all $P$:

\[
\forall G_i \in \llbracket P \rrbracket_{I_{\text{queue}}} . \ \exists G_s . \ \text{val}(G_i) = \text{val}(G_s)
\]

$(G_s)_{\text{queue}}$ sats. `queue axioms

$G_s . \ \text{so}_{\text{queue}} \subseteq G_i . \ \text{hb}$
Local Soundness

I_{queue} sound \iff \text{for all } P:
\forall G_i \in \llbracket P \rrbracket_{I_{queue}}. \exists G_s. \text{val}(G_i) = \text{val}(G_s)

(G_s)_{queue} \text{ sats. } \text{queue axioms}

(G_s)_{L_1} \text{ sats. } L_1 \text{ axioms}

\ldots

(G_s)_{L_n} \text{ sats. } L_n \text{ axioms}

\text{I}_{queue} \text{ locally sound } \iff \text{for all } P:
\forall G_i \in \llbracket P \rrbracket_{I_{queue}}. \exists G_s. \text{val}(G_i) = \text{val}(G_s)

(G_s)_{queue} \text{ sats. } \text{queue axioms}

G_s. so_{queue} \subseteq G_i. hb

Given for free!
Theorem (Compositionality)

\[ I \text{ locally-sound} \implies I \text{ sound} \]

*Proof.* Mechanised in Coq.
Case Studies

- **portable existing declarative specification spinlock implementation using CAS\textsubscript{acq} and W\textsubscript{rel}
- **push/pop may fail** if contention on stack top
- **strong** locking queue

- **scalable** stack implementation
- an array of exchanger objects
- exchanger object from \texttt{java.util.concurrent}
- ported **existing** declarative specification
- multiple-reader-single-writer lock implementation
- **weak & strong** Herlihy-Wing queue implementations

- Elimination-Stack
  - ExchArray
  - Weak-Stack
  - LockedQ
  - HW-Queue
  - Exchanger
  - Mutex
  - MRSW-Lock
  - C11
Summary

A declarative **specification** and **verification** framework:

✓ **Agnostic** to memory model
  ➡ support both SC and WMC specs

✓ **General**
  ➡ port existing SC (*linearisability*) specs
  ➡ port existing WMC specs (e.g. C11, TSO)
  ➡ built from the **ground up**: assume no pre-existing libraries or specs

✓ **Compositional**
  ➡ *vertical* composition to verify library implementations
  ➡ *horizontal* composition to verify client programs
Future Work

• Support *load buffering* behaviour
  — allow \((po \cup so)^+\) cycles

• MM-agnostic, general *program logic*
  — port existing WMC logics, e.g. RSL, GPS, …

• MM-agnostic (stateless) *model-checking*
  — a generalisation of existing approaches, e.g. RCMC, …
Thank You for Listening!

A declarative **specification** and **verification** framework:

- **✓ Agnostic** to memory model
  - support *both* SC and WMC specs

- **✓ General**
  - port *existing* SC (*linearisability*) specs
  - port *existing* WMC specs (e.g. C11, TSO)
  - built from the *ground up*: assume *no pre-existing* libraries or specs

- **✓ Compositional**
  - *vertical* composition to verify *library implementations*
  - *horizontal* composition to verify *client programs*

---

Azalea mmpi-sws.org
SoundAndComplete.org