On Library Correctness under Weak Memory Consistency
Specifying and Verifying Concurrent Libraries under Declarative Consistency Models

Azalea Raad  Marko Doko  Lovro Rožić  Ori Lahav  Viktor Vafeiadis
Max Planck Institute for Software Systems (MPI-SWS)
Tel Aviv University

plv.mpi-sws.org/yacovet/

azalea@mpi-sws.org  SoundAndComplete.org  @azalearaad
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
Declarative Framework Desiderata

• **Agnostic** to memory model
  ‣ support *both* SC and WMC specs
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
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)  // returns 1
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 (linearity) 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*

```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 (linearizability) 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* \(\Rightarrow\) *towers of abstraction*

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

enq(q,1);  a:=pop(s);
push(s,2)   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*

```plaintext
q:=new-queue();
s:=new-stack();
enq(q,1);
push(s,2) a:=pop(s);
if(a==2) b:=deq(q) // returns 1
```
• Define a (partial) happens-before relation $hb$ on events

- $(e_1, e_2) \in hb \iff e_1.end \prec_\text{time} e_2.begin$

  -- e.g. $(a, b) \in hb$ \quad $(a, c) \not\in hb$
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** $\iff \exists to. to$ **totally** orders events
  - $hb \subseteq to$
  - $to$ is a **legal** sequence (library-specific)
    - e.g. $to$ is a **FIFO** sequence
• 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

\[ a \quad b \quad c \quad \checkmark \text{ linearisable} \]
Why Not Linearisability?

- assumes $\lt_{\text{time}}$ order -- not present under WMC
- requires $\text{total}$ order to on all events -- not always possible under WMC
Why Not Linearisability?

✘ assumes \(<_{\text{time}}\) order -- not present under WMC

✘ requires \(\text{total}\) order \(\text{to}\) on \(\text{all}\) events -- not always possible under WMC

\[
x := 1; \quad b := y \quad \text{// 0}
\]

\[
y := 1; \quad d := x \quad \text{// 0}
\]

(SB)
Why Not Linearisability?

- Assumes \( \langle \text{time} \rangle \) order -- not present under WMC
- Requires \textit{total} order \textit{to} on \textit{all} events -- not always possible under WMC

\[
\begin{align*}
\text{a} & : x := 1; \\
\text{b} & : y := 0 \\
\text{c} & : y := 1; \\
\text{d} & : d := x \quad \text{// 0}
\end{align*}
\]
Why Not Linearisability?

- Assumes \( \langle \text{time} \rangle \) order -- not present under WMC
- Requires \textit{total} order to on \textit{all} events -- not always possible under WMC

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

\( \text{(SB)} \)
Why Not Linearisability?

✘ assumes \( \langle \text{time} \rangle \) order -- not present under WMC

✘ requires \textit{total} order \textit{to} on \textit{all} events -- not always possible under WMC

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

(SB)
Why Not Linearisability?

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

```plaintext
a x:=1;  
b b:=y // 0  
c y:=1;  
d d:=x // 0
```

(SB) not linearisable
Why Not Linearisability?

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

Per-location linearisability?

- Cannot model \(\text{weak specs}\), e.g. C11, TSO

\[
\begin{align*}
a & : x := 1; \\
b & : y := 1; \\
\text{SB} & : d := x; \\
c & : y := 1; \\
& : d := x
\end{align*}
\]
Our Solution

✔ no particular memory model
✔ no \textless_{\text{time}}\ order
✔ no total order on events
✔ per-library specification
Our Solution

- **no** particular memory model
- **no** $\lt_{\text{time}}$ order
- **no total** order on events
- **per-library** specification

$\rightarrow$ set of library executions

$$\{G \mid G \text{ satisfies certain } \textit{axioms}\}$$

library execution
Library Executions

Example: **Queue** Library

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

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

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

- **n** `new-queue(q)`
- **e** `enq(q, 1)`
- **d** `deq(q, 1)`
**Library Executions**

**Example: Queue Library**

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

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

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

**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) then
  b := deq(q) // may read 1 or empty (⊥)
```

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

- **New queue** (`n`): `new-queue(q)`
- **Enqueue** (`e`): `enq(q, 1)`
- **Dequeue** (`d`): `deq(q, 1)`
- **Pop** (`p`): `pop(s)`
- **Head of queue** (`h`): `deq(q)`
- **Synchronization order** (`so`): `deq(q)` is synchronous with `enq(q, 1)`
- **Program order** (`po`): `enq(q, 1)` is before `deq(q, 1)`

**Graphs:**
- **Left:** `n`, `e`, `d`
- **Right:** `n`, `e`, `d`
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 (⊥)
```

$G_{queue} = \langle E, po, so, hb \rangle$

- **events**
- **program-order**
- **happens-before order**
- **synchronisation-order**

```
new-queue(q)
-
enq(q, 1)
-
deq(q, 1)
-
new-queue(q)
-
enq(q, 1)
-
deq(q, ⊥)
```
Example: **Queue Library**

$q := \text{new-queue}();$
$s := \text{new-stack}();$

$\text{enq}(q, 1);$  
$\text{push}(s, 2)$

$\text{a} := \text{pop}(s);$

if ($a == 2$)

$b := \text{deq}(q)$ // may read 1 or empty (⊥)

---

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

- **$E$**: events
- **$po$**: program-order
- **$so$**: synchronisation-order
- **$hb$**: happens-before order (coming soon!)

---

- $\text{new-queue}(q)$
- $\text{enq}(q, 1)$
- $\text{deq}(q, 1)$

- $\text{new-queue}(q)$
- $\text{enq}(q, 1)$
- $\text{deq}(q, ⊥)$

- $x$
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 (⊥)
```

How to eliminate this "**incorrect**" behaviour?
Program Executions

\[ [P] = \{ \mathcal{G}_P = < (E, po, so) > | \ldots \} \]

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

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

- **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

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

semantics of \( P \)

execution of \( P \)

\[
q := \text{new-queue}();
\]

\[
s := \text{new-stack}();
\]

\[
enq(q, 1); \quad \text{a} := \text{pop}(s); \quad \text{if}(\text{a} == 2)
\]

\[
push(s, 2) \quad \text{b} := \text{deq}(q) \quad \text{// should return 1}
\]
Program Executions

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

semantics of \(P\)

execution of \(P\)

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

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

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

semantics of \( P \)

execution of \( P \)

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

\[
q := \text{new-queue}();
\]

\[
s := \text{new-stack}();
\]

\[
enq(q, 1); \quad a := \text{pop}(s);
\]

\[
push(s, 2) \quad \text{if}(a == 2) \quad b := \text{deq}(q) \quad \text{// should return 1}
\]
Program Executions

$$\text{⟦P⟧} = \left\{ G_P = < E, po, so > \mid \cdots \right\}$$

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

semantics of $P$

execution of $P$

allow libraries to constrain each other via $hb$!
Program Executions

\[
\begin{align*}
[P] &= \left\{ G_P = < E, po, so > \mid \cdots \right\} \\
\text{semantics of } P & \quad \text{execution of } P
\end{align*}
\]

allow libraries to constrain each other via \( hb \)!

How? \( hb \) defined on program executions
From **Program** to **Library** Executions

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

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

\[ G_P = \langle E, \text{ po }, \text{ so } \rangle \]
\[ E_{\text{queue}} \cup E_{\text{stack}} \]

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

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

\( G_{queue} \oplus G_{stack} \)

\( E_{queue} \cup E_{stack} \)

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

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

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

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

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

\[ G_{queue} = \langle E_{queue}, \rangle \]
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

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

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

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

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

\[
G_{\text{queue}} = \langle E_{\text{queue}}, \text{po}_{\text{queue}}, \text{so}_{\text{queue}} \rangle
\]
From \textbf{Program} to \textbf{Library} Executions

\[ G_P = < E, \text{po}, \text{so} > \]

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

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

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

\[ G_{\text{queue}} = < E_{\text{queue}}, \text{po}_{\text{queue}}, \text{so}_{\text{queue}}, \text{hb} > \]
From **Program** to **Library** Executions

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

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

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

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

\[ G_{queue} = \langle E_{queue}, po_{queue}, so_{queue}, hb_? \rangle \]

\[ hb_{queue} \]
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
```

```
q := new-queue(q)

s := new-stack(s)

e := enq(q, 1)
r := pop(s, 2)

d := deq(q, 1)
a := push(s, 2)
b := pop(s, 2)
c := deq(q, ⊥)
```
Example Revisited

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

```
enq(q, 1);
push(s, 2)
```
How does \textit{hb} exclude the \textit{RHS} execution?

\textit{library axioms!}
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*}

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

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

1. \(E\) contains queue events

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

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

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

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

\[
\begin{array}{c}
\text{enq}(q, v) & \text{deq}(q, w) \\
e & d
\end{array}
\]

\(\text{then} \quad v = w\)
Queue Axioms

\[ G = \langle E, po, so, hb \rangle \] is a consistent \textbf{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 \textbf{1-to-1}

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

\[
\text{if } \quad \begin{align*}
\text{e} & \quad \text{enq}(q, v) \\
\text{e} & \quad \text{enq}(q, v)
\end{align*} \quad \text{then } \quad v = w
\]

\[
\begin{align*}
\text{d} & \quad \text{deq}(q, w) \\
\text{f} & \quad \text{deq}(q, \bot)
\end{align*} \quad \text{for all } \quad v
\]
Queue Axioms

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

1. $E$ contains queue events

   $v, w \in Val$

2. so is 1-to-1

   so relates matching $enq/deq$ events;

   \[
   \begin{array}{c}
   \text{if} \quad \text{enq}(q, v) \quad \text{so} \quad \text{deq}(q, w) \\
   \text{then} \quad v = w \\
   \end{array}
   \]

   \[
   \begin{array}{c}
   \text{if} \quad \text{enq}(q, v) \quad \text{so} \quad \text{deq}(q, \bot) \\
   \text{for all} \quad v \\
   \end{array}
   \]

3. $\exists$ to. to totally orders $E$,

   \[
   \begin{array}{c}
   \text{hb} \subseteq \text{to} \quad \text{and} \quad \text{to} \text{ is a FIFO sequence} \\
   \end{array}
   \]
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
```

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

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

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

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

\[ \text{hb} \subseteq \text{to} \]
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
```

```
qb, po    po, hb

new-queue(q)

s
new-stack(s)

enq(q, 1)

push(s, 2)

pop(s, 2)

deq(q, ⊥)
```

```
hb = (po ∪ so)^+   hb ⊆ 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
```

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

\[ hb \subseteq to \]
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
```
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 $E$,
   
   $hb \subseteq to$ and $to$ is a FIFO sequence

✗ too strong
✗ difficult to find $to$ witness
Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

\[
\begin{align*}
\text{new-queue()} & \triangleq \\
& \quad \text{let } q = \text{alloc}(+\infty) \text{ in } q \\
\text{enq}(q, v) & \triangleq \\
& \quad \text{let } i = \text{fetch-add}(q, 1, \text{rel}) \text{ in } \\
& \quad \text{store}(q + i + 1, v, \text{rel}); \\
\text{deq}(q) & \triangleq \\
& \quad \text{loop} \\
& \quad \text{let } range = \text{load}(q, \text{acq}) \text{ in } \\
& \quad \text{for } i = 1 \text{ to } range \text{ do} \\
& \quad \quad \text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in } \\
& \quad \quad \text{if } x \neq 0 \text{ then } \text{break}_2 x
\end{align*}
\]
**Why Not Strong** Queue Axioms?

C11 Herlihy-Wing Queue Implementation

\[ \text{new-queue}() \triangleq \]
\[
\text{let } q = \text{alloc}(+\infty) \text{ in } q
\]

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

\[ \text{deq}(q) \triangleq \]
\[
\text{loop}\\
\text{let } range = \text{load}(q, \text{acq}) \text{ in }\\
\text{for } i = 1 \text{ to } range \text{ do }\\
\text{let } x = \text{atomic-xchg}(q + i, 0, \text{acq}) \text{ in }\\
\text{if } x \neq 0 \text{ then break}\_2 x
\]
Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

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

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

\[\times\text{ does not satisfy strong queue axioms}\]
Why Not Strong Queue Axioms?

C11 Herlihy-Wing Queue Implementation

\[
\text{new-queue()} \triangleq \\
\quad \text{let } q = \text{alloc}(+\infty) \text{ in } q
\]

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

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

\text{wanted}

\text{Weaker queue axioms}

Why weak axioms?

\text{strong enough} for certain uses: \text{single-producer-single-consumer}
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 $E$,
   
   $hb \subseteq to$ and $to$ is a FIFO sequence

✘ too strong ☻ weak axioms (see our paper)

✘ difficult to find $to$ witness
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 \( E \),
   \[ hb \subseteq to \quad \text{and} \quad to \text{ is a FIFO sequence} \]

- too strong ➕ **weak axioms** (see our paper)
- difficult to find to witness

**equivalent acyclicity axiom**
(see our paper)
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*
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]{@mpi-sws.org}  
[@azalearaad]{@SoundAndComplete.org}