Hoare-style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects

Ilya Sergey

joint work with
Aleks Nanevski, Anindya Banerjee, and Germán Andrés Delbianco
Non-overlapping calls to methods of a concurrent object should appear to take effect in their sequential order.
Laws of Order: Expensive Synchronization in Concurrent Algorithms Cannot be Eliminated

Hagit Attiya
Technion
hagit@cs.technion.il

Rachid Guerraoui
EPFL
rachid.guerraoui@epfl.ch

Danny Hendler
Ben-Gurion University
hendlerd@cs.bgu.ac.il

Petr Kuznetsov
TU Berlin/Deutsche Telekom Labs
pkuznets@acm.org

Maged M. Michael
IBM T. J. Watson Research Center
magedm@us.ibm.com

Martin Vechev
IBM T. J. Watson Research Center
mtvechev@us.ibm.com
An alternative to linearizability?

The advent of multicore processors as the standard computing platform will force major changes in software design.

BY NIR SHAVIT

Data Structures in the Multicore Age

Relaxing the correctness condition would allow one to implement concurrent data structures more efficiently, as they would be free of synchronization bottlenecks.
Alternatives to linearizability

- Quiescent Consistency [Aspnes-al:JACM94]
- Quasi-Linearizability [Afek-al:OPODIS10]
- Quantitative Relaxation [Henzinger-al:POPL13]
- Concurrency-Aware Linearizability [Hemed-Rinetzky:PODC14]
- Quantitative Quiescent Consistency [Jagadeesan-Riely:ICALP14]
- Local Linearizability [Haas-al:arXiv15]
- …
Challenges of diversity

• *Composing* different conditions (CAL, QC, QQC) in a single program, which uses multiple objects;

• Providing *syntactic* proof methods for establishing all these conditions (akin to *linearization points*);

• Employing these criteria for *client-side reasoning* (uniformity).
Hoare-style Specifications as Correctness Conditions for Non-Linearizable Concurrent Objects

Ilya Sergey

joint work with
Aleks Nanevski, Anindya Banerjee, and Germán Andrés Delbianco
If the initial state satisfies $P$, then, after $e$ terminates, the final state satisfies $Q$ (no matter the interference manifested by $C$).
Hoare-style Specifications

- **Compositional** — substitution principle;
- **Syntactic proof method** — inference rules;
- **Uniform** — reasoning about objects and their clients in the same proof system.
Concurrency-Aware Linearizability (CAL):

Effects of some concurrent method calls should appear to happen simultaneously.

Quiescent Consistency (QC):

Method calls separated by a period of quiescence should appear to take effect in their order.

Quantitative Quiescent Consistency (QQC):

The number of out-of-order method results is bounded by the number of interfering threads (with a constant factor).
Simple Counting Network

```python
def getAndInc() : nat
```
def getAndInc() : nat = {
    n ← &x;
    b ← CAS(x, n, n + 1);
    if b then
        return n;
    else getAndInc();
}
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
Sequential Execution \((T_1)\)

```python
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

<table>
<thead>
<tr>
<th>bal</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>x+1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}

Sequential Execution ($T_1$)
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}

Sequential Execution (T₁)

T₁.b₁ = 0
T₁.res₁ = 0
T₁.b₂ = 1
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}

Sequential Execution ($T_1$)

bal 0

x 2
x+1 3

$T_1.b_1 = 0$
$T_1.b_2 = 1$
$T_1.res_1 = 0$
$T_1.res_2 = 1$
Concurrent Execution \((T_1, T_2)\)

```python
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
```

<table>
<thead>
<tr>
<th>bal</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>x+1</td>
<td>1</td>
</tr>
</tbody>
</table>
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}

Concurrent Execution \((T_1, T_2)\)

\[ T_1.b_1 = 0 \]
\[ T_2.b_1 = 1 \]
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}

Concurrent Execution (T₁, T₂)

T₁.b₁ = 0
T₂.b₁ = 1
T₂.res₁ = 1
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
def getAndInc() : nat = {
    b ← flip(bal);
    res ← fetchAndAdd2(x + b);
    return res;
}
Correctness Conditions for Counting Network

- \( R_0 \): calls to `getAndInc()` take effect in their sequential order
- \( R_1 \): different calls return distinct results (strong concurrent counter)
- \( R_2 \): two calls, separated by period of quiescence, take effect in their sequential order (QC)
- \( R_3 \): results of two calls in the same thread are out of order by no more than 2 * (number of calls interfering with both) (QQC)
Invariants of the Counting Network

• Every flip of the balancer grants thread a capability to add 2 to a counter (x or x+1);

• Each of the counters (x and x+1) changes continuously wrt. even/odd values;

• Threads, which gained capabilities but haven’t yet incremented, cause one counter to “run ahead” of another one, leading to out-of-order anomalies.
Real and Auxiliary State

- Hoare-style specs constrain state, auxiliary or real

- **Real state** — heap (pointers bal, x, x+1);

- **Auxiliary state** — any fictional splittable resource, represented as a PCM (S, ⊕, 0), e.g.,
  - Tokens — disjoint sets;
  - Histories — partial maps with nat as domain.
• New *unique* tokens are emitted upon calling \texttt{flip()};

• Calling \texttt{fetchAndAdd2()} \textit{consumes} a token and \textit{adds} an entry to the history.
Interference-capturing histories

\[ \eta = \{ \ldots, t \mapsto (l, z), \ldots \} \]

“timestamp”, a value written to a counter \( x \) or \( x+1 \) (0, 1, 2, etc.)
Interference-capturing histories

\[ \eta = \{ \ldots, t \mapsto (t, z), \ldots \} \]

sets of tokens, held by interfering threads at the moment the entry has been written
Interference-capturing histories

\[ \eta = \{ \ldots, t \mapsto (l, [Z]), \ldots \} \]

a token, spent to increment \( x \) or \( x+1 \) from \( t-2 \) to \( t \)
Notation for *Subjective* Histories and Tokens

- $\chi_s, \chi_o$ — histories, contributed by *self* and *other* threads;
- $\tau_s, \tau_o$ — tokens, held by *self* and *other* threads;
- $\eta, \iota$ — logical variables for histories and tokens.
Specification of \texttt{getAndInc()} 

\{ \exists \, \iota, \, z, \quad \tau_s = \emptyset, \; \chi_s = \eta_s, \quad \\
\quad \eta_0 \subseteq \chi_0, \quad \\
\quad \iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0) \quad \} 

\texttt{getAndInc()} 

\{ \exists \, \iota, \, Z, \quad \tau_s = \emptyset, \; \chi_s = \eta_s \cup \text{res} + 2 \mapsto (\iota, \, Z), \quad \\
\quad \eta_0 \subseteq \chi_0, \; \iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0), \quad \\
\quad \text{last}(\eta_s \cup \eta_0) < \text{res} + 2 + 2 \mid \iota \cap \iota_0 \mid \} \; @ \; C
Specification of `getAndInc()`

\[
\{ \quad \begin{array}{l}
T_S = \emptyset, \quad \chi_S = \eta_S, \\
\eta_o \subseteq \chi_o, \\
l_o \subseteq T_o \cup spent(\chi_o \setminus \eta_o)
\end{array} \quad \}
\]

`getAndInc()`

\[
\{ \exists \, l, \, z, \quad T_S = \emptyset, \quad \chi_S = \eta_S \cup res+2 \mapsto (l, \, z), \\
\eta_o \subseteq \chi_o, \quad l_o \subseteq T_o \cup spent(\chi_o \setminus \eta_o), \\
last(\eta_s \cup \eta_o) < res + 2 + 2 \mid l \cap l_o \mid \} \quad @ \ C
\]
Specification of `getAndInc()`

\[
\left\{ \begin{array}{c}
\mathcal{T}_s = \emptyset, \quad \mathcal{X}_s = \eta_s, \\
\eta_0 \subseteq \chi_0, \\
\iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0)
\end{array} \right. 
\]

boundary on initial other-history and tokens

\[
\left\{ \begin{array}{c}
\exists \iota, z, \quad \mathcal{T}_s = \emptyset, \quad \mathcal{X}_s = \eta_s \cup \text{res}+2 \mapsto (\iota, z), \\
\eta_0 \subseteq \chi_0, \quad \iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0), \\
\text{last}(\eta_s \cup \eta_0) < \text{res} + 2 + 2 | \iota \cap \iota_0 | \end{array} \right\} @ C
Specification of \texttt{getAndInc()} \( (\)\)

\[
\{ \quad T_s = \emptyset, \quad \chi_s = \eta_s, \\
\quad \eta_o \subseteq \chi_o, \\
\quad \iota_o \subseteq T_o \cup \text{spent}(\chi_o \setminus \eta_o) \quad \} 
\]

getAndInc()

\[
\{ \exists \ i, \ z, \quad T_s = \emptyset, \quad \chi_s = \eta_s \cup \text{res} + 2 \mapsto (i, z), \\
\quad \eta_o \subseteq \chi_o, \quad \iota_o \subseteq T_o \cup \text{spent}(\chi_o \setminus \eta_o), \\
\quad \text{last}(\eta_s \cup \eta_o) < \text{res} + 2 + 2 \mid \iota \cap \iota_o \mid \} \quad @ \quad C
\]

final tokens and self-history
Specification of \( \text{getAndInc}() \)

\[
\{ \quad \tau_s = \emptyset, \quad \chi_s = \eta_s,
\eta_o \subseteq \chi_o,
\iota_o \subseteq \tau_o \cup \text{spent}(\chi_o \setminus \eta_o) \quad \} \\
\]

getAndInc()

\[
\{ \exists \ i, \ z, \quad \tau_s = \emptyset, \quad \chi_s = \eta_s \cup \text{res} + 2 \mapsto (i, \ z),
\eta_o \subseteq \chi_o, \ i_o \subseteq \tau_o \cup \text{spent}(\chi_o \setminus \eta_o),
\text{last}(\eta_s \cup \eta_o) < \text{res} + 2 + 2 \mid i \cap i_o \} \quad @ \ C
\]

constraining final other-history and tokens
Specification of \texttt{getAndInc( )}

\[
\{ \exists \, l, z, \quad \tau_s = \emptyset, \; \chi_s = \eta_s, \\
\quad \chi_o \subseteq \chi_o, \\
\quad \eta_0 \subseteq \chi_o, \; \iota_0 \subseteq \tau_o \cup \text{spent}(\chi_o \setminus \eta_0), \\
\quad \text{last}(\eta_s \cup \eta_o) < \text{res} + 2 + 2 \mid \iota \cap \iota_0 \} \quad @ \; C
\]

result + 2 is greater than any previous value of the counters, recorded in history (modulo past and present interference)
Implications of the derived spec

Trivial from invariants: each result corresponds to a new history entry

• **R₁**: different calls return distinct results *(strong concurrent counter)*

• **R₂**: two calls, separated by *period of quiescence*, take effect in their sequential order *(QC)*

• **R₃**: results of *two calls* in the same thread are out of order by no more than 2 * (number of calls *interfering with both*) *(QQC)*
Implications of the derived spec

• **R₁**: different calls return *distinct* results (**strong concurrent counter**)

• **R₂**: two calls, separated by *period of quiescence*, take effect in their sequential order (**QC**)

• **R₃**: results of *two calls* in the same thread are out of order by no more than $2 \times$ (number of calls *interfering with both*) (**QQC**)
Exercising Quiescent Consistency

```
(res1, _) ← (getAndInc() || e1);
(res2, _) ← (getAndInc() || e2);
return (res1, res2);

{ ¿ res1 < res2? }
```
Generic spec for Interference

\[
\{ T_S = \emptyset, \ X_S = \emptyset, \ I_0 \subseteq T_0 \cup \text{spent}(X_0) \} \\
e_i \\
\{ \exists \eta_i, \ T_S = \emptyset, \ X_S = \eta_i, \ I_0 \subseteq T_0 \cup \text{spent}(X_0) \} @ C
\]
Spec for parallel composition

\{ \tau_s = \emptyset, \chi_s = \eta_s, \eta_0 \subseteq \chi_0, \iota_0 \subseteq \tau_0 \cup spent(\chi_0 \setminus \eta_0), \ldots \} \\

\text{getAndInc()} \ || \ e_i

\{ \exists \iota, z, \eta_i, \tau_s = \emptyset, \chi_s = \eta_s \cup \eta_i \cup \text{res+2} \mapsto (\iota, z), \eta_0 \subseteq \chi_0, \iota_0 \subseteq \tau_0 \cup spent(\chi_0 \setminus \eta_0), \text{last}(\eta_s \cup \eta_o) < \text{res.1} + 2 + 2 | \iota \cap \iota_0 | \} \ @ \ C
Spec for parallel composition

\{ \tau_s = \emptyset, \chi_s = \eta_s, \eta_0 \subseteq \chi_0, \iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0), \ldots \} \\

getAndInc() || e_i

\{ \exists \ i, z, \eta_i, \ \tau_s = \emptyset, \chi_s = \eta_s \cup \eta_i \cup \text{res+2} \mapsto (i, z), \eta_0 \subseteq \chi_0, \\
\iota_0 \subseteq \tau_0 \cup \text{spent}(\chi_0 \setminus \eta_0), \\
\text{last}(\eta_s \cup \eta_0) < \text{res.1 + 2 + 2} | \iota \cap \iota_0 | \} @ C
\{ \tau_s = \emptyset, \chi_s = \eta_s, \ldots \} \\

(res_1, -) \leftarrow (getAndInc() || e_1);

\{ \exists \eta_1, \tau_s = \emptyset, \chi_s = \eta'_s, \eta_0 \subseteq \chi_0, \chi_s = \eta'_s \cup \eta_1 \cup res_1 + 2 \mapsto -, \eta_0 = \chi_0 \text{ and } \iota_0 = \tau_0 \}

(res_2, -) \leftarrow (getAndInc() || e_2);

\{ \exists \eta_1, \eta_2, \iota, \tau_s = \emptyset, \chi_s = \eta'_s, \eta_0 \subseteq \chi_0, \chi_s = \eta'_s \cup \eta_2 \cup res_2 + 2 \mapsto -, \iota_0 \subseteq \tau_0 \cup spent(\chi_0 \setminus \eta_0), \text{last}(\eta'_s \cup \eta_0) < res_2 + 2 + 2 | \iota \cap \iota_0 |} \\

\textbf{return} \ (res_1, \ res_2);
\{(T_S = \emptyset, \chi_S = \eta_S, \ldots)\}

\((\text{res}_1, -) \leftarrow (\text{getAndInc()} \ || \ e_1)\);

\{(\exists \eta_1, T_S = \emptyset, \chi_S = \eta'_S, \eta_0 \subseteq \chi_0, \text{ where } \eta'_S = \eta_S \cup \eta_1 \cup \text{res}_1 + 2 \mapsto -, \eta_0 = \chi_0 \text{ and } l_0 = T_0)\}

\{(\exists \eta_1, \eta_2, l, T_S = \emptyset, \chi_S = \eta'_S, \eta_0 \subseteq \chi_0, \chi_S = \eta'_S \cup \eta_2 \cup \text{res}_2 + 2 \mapsto -, l_0 \subseteq T_0 \cup \text{spent}(\chi_0 \setminus \eta_0), \text{last}(\eta'_S \cup \eta_0) < \text{res}_2 + 2 + 2 | l \cap l_0|)\}

\text{return} \ (\text{res}_1, \text{res}_2);
\{ T_s = \emptyset, \, \chi_s = \eta_s, \ldots \} \\

(res_1, -) \leftarrow (getAndInc() \mid\mid e_1); \\

\{ \exists \, \eta_1, \, T_s = \emptyset, \, \chi_s = \eta'_s, \, \eta_0 \subseteq \chi_0, \ \\
\text{where } \eta'_s = \eta_s \cup \eta_1 \cup res_1 + 2 \mapsto -, \, \eta_0 = \chi_0 \text{ and } \, l_0 = T_0 \} \\

(res_2, -) \leftarrow (getAndInc() \mid\mid e_2); \\

\{ \exists \, \eta_1, \eta_2, \, l, \, T_s = \emptyset, \, \chi_s = \eta'_s, \, \eta_0 \subseteq \chi_0, \, \chi_s = \eta'_s \cup \eta_2 \cup res_2 + 2 \mapsto - \ \\
\quad \, l_0 \subseteq \emptyset, \ \\
\quad \text{last}(\eta'_s \cup \eta_0) < res_2 + 2 + 2 | \eta_0 | \} \\

\textbf{return} \ (res_1, \, res_2);
\[
\{ T_S = \emptyset, \; \chi_S = \eta_S, \; \ldots \} \\

(res_1, \; -) \leftarrow (getAndInc() \; || \; e_1); \\

\{ \exists \eta_1, \; T_S = \emptyset, \; \chi_S = \eta'_S, \; \eta_o \subseteq \chi_o, \\
\text{where } \eta'_S = \eta_S \cup \eta_1 \cup res_1 + 2 \mapsto - , \; \eta_0 = \chi_0 \text{ and } l_0 = T_0 \} \\

(res_2, \; -) \leftarrow (getAndInc() \; || \; e_2); \\

\{ \exists \eta_1, \eta_2, \; l, \; T_S = \emptyset, \; \chi_S = \eta'_S, \; \eta_0 \subseteq \chi_0, \; \chi_S = \eta'_S \cup \eta_2 \cup res_2 + 2 \mapsto - , \\
\text{where } l_0 = \emptyset, \\
\text{last}(\eta'_S \cup \eta_0) < res_2 + 2 + 2 | l \cap l_0 | \} \\

\text{return} \; (res_1, \; res_2);
\{ T_S = \emptyset, \, X_S = \eta_S, ... \} \\

(res_1, -) \leftarrow (\text{getAndInc()} \mid \mid e_1); \\

\{ \exists \, \eta_1, T_S = \emptyset, \, X_S = \eta'_S, \, \eta_0 \subseteq X_0, \\
\text{where} \, \eta'_S = \eta_S \cup \eta_1 \cup \text{res}_1+2 \mapsto -, \, \eta_0 = X_0 \text{ and } l_0 = T_0 \} \\

(res_2, -) \leftarrow (\text{getAndInc()} \mid \mid e_2); \\

\{ \exists \, \eta_1, \eta_2, l, T_S = \emptyset, \, X_S = \eta'_S, \, \eta_0 \subseteq X_0, \, X_S = \eta'_S \cup \eta_2 \cup \text{res}_2+2 \mapsto - \\
\quad l_0 = \emptyset, \\
\text{last}(\eta'_S \cup \eta_0) < \text{res}_2 + 2 + 2 \mid l \cap l_0 \mid \} \\

\text{return} \, (\text{res}_1, \, \text{res}_2);
\{ T_s = \emptyset, \, \chi_s = \eta_s, \, \ldots \} \\

(res_1, -) \leftarrow (getAndInc() \mid \mid e_1); \\

{\exists \eta_1, T_s = \emptyset, \, \chi_s = \eta'_s, \, \eta_0 \subseteq \chi_0, \, \text{where } \eta'_s = \eta_s \cup \eta_1 \cup res_1 + 2 \mapsto - , \eta_0 = \chi_0 \, \text{and } l_0 = T_0} \\

(res_2, -) \leftarrow (getAndInc() \mid \mid e_2); \\

{\exists \eta_1, \eta_2, I, \, T_s = \emptyset, \, \chi_s = \eta'_s, \, \eta_0 \subseteq \chi_0, \, \chi_s = \eta'_s \cup \eta_2 \cup \eta_1 \cup res_2 + 2 \mapsto - , \, l_0 = \emptyset, \, \text{last}(\eta'_s \cup \eta_0) < res_2 + 2 + 2 \mid \emptyset \mid} \\

return \, (res_1, \, res_2);
\{ T_s = \emptyset, \; \chi_s = \eta_s, \; \ldots \; \}

(res_1, \; -) \leftarrow (getAndInc() \; || \; e_1);

\{ \exists \eta_1, T_s = \emptyset, \; \chi_s = \eta_s', \; \eta_o \subseteq \chi_o, \; where \eta_s' = \eta_s \cup \eta_1 \cup \text{res}_1 + 2 \mapsto \; -, \; \eta_0 = \chi_0 \; and \; l_0 = \tau_0 \}

(res_2, \; -) \leftarrow (getAndInc() \; || \; e_2);

\{ \exists \eta_1, \eta_2, l, T_s = \emptyset, \; \chi_s = \eta_s', \; \eta_0 \subseteq \chi_0, \; \chi_s = \eta_s' \cup \eta_2 \cup \text{res}_2 + 2 \mapsto \; -, \; l_0 = \emptyset, \; \text{last}(\eta_s' \cup \eta_0) < \text{res}_2 + 2 \}

return (res_1, \; \text{res}_2);
\[ \{ T_S = \emptyset, \; \chi_S = \eta_S, \; \ldots \} \]

\[(res_1, -) \leftarrow (getAndInc() \; || \; e_1);\]

\[\{ \exists \eta_1, T_S = \emptyset, \; \chi_S = \eta_s', \; \eta_o \subseteq \chi_o, \]

where \( \eta_s' = \eta_s \cup \eta_1 \cup res_1 + 2 \mapsto - \), \( \eta_o = \chi_o \) and \( l_o = T_o \}\]

\[(res_2, -) \leftarrow (getAndInc() \; || \; e_2);\]

\[\{ \exists \eta_1, \eta_2, l, T_S = \emptyset, \; \chi_S = \eta_s', \; \eta_o \subseteq \chi_o, \chi_S = \eta_s' \cup \eta_2 \cup res_2 + 2 \mapsto - \]

\[l_o = \emptyset, \]

\[last(\eta_s' \cup \eta_o) < res_2 + 2 \}\]

\text{return} \; (res_1, \; res_2);
\{( \tau_s = \emptyset, \chi_s = \eta_s, \ldots \)\}

\((\text{res}_1, -) \leftarrow (\text{getAndInc()} \mid \mid \text{e}_1);\)

\{\exists \eta_1, \tau_s = \emptyset, \chi_s = \eta_1', \eta_0 \subseteq \chi_0, \text{ where } \eta_1' = \eta_s \cup \eta_1 \cup \text{res}_1 + 2 \mapsto -, \eta_0 = \chi_0 \text{ and } \iota_0 = \tau_0\}

\((\text{res}_2, -) \leftarrow (\text{getAndInc()} \mid \mid \text{e}_2);\)

\{\exists \eta_1, \eta_2, \iota, \tau_s = \emptyset, \chi_s = \eta_1', \eta_0 \subseteq \chi_0, \chi_s = \eta_1' \cup \eta_2 \cup \text{res}_2 + 2 \mapsto -, \iota_0 = \emptyset, \text{ res}_1 + 2 < \text{res}_2 + 2 \}\)

\text{return } (\text{res}_1, \text{res}_2);\)
\{ T_s = \emptyset, \chi_s = \eta_s, \ldots \} \\

\{(res_1, -) \leftarrow (\text{getAndInc()} || e_1); \}

\{\exists \eta_1, T_s = \emptyset, \chi_s = \eta'_s, \eta_o \subseteq \chi_o, \}
where \eta'_s = \eta_s \cup \eta_1 \cup res_1 + 2 \mapsto -, \eta_o = \chi_o and l_0 = t_0 \}

\{(res_2, -) \leftarrow (\text{getAndInc()} || e_2); \}

\{\exists \eta_1, \eta_2, l, T_s = \emptyset, \chi_s = \eta'_s, \eta_o \subseteq \chi_o, \chi_s = \eta'_s \cup \eta_2 \cup res_2 + 2 \mapsto - \}
\quad l_0 = \emptyset,

\textbf{res_1 < res_2} \\

\textbf{return} \ (res_1, res_2);
Implications of the derived spec

- **R₁**: different calls return *distinct* results (*strong concurrent counter*)

- **R₂**: two calls, separated by *period of quiescence*, take effect in their sequential order (*QC*)

- **R₃**: results of *two calls* in the same thread are out of order by no more than $2 \times$ (number of calls *interfering with both*) (*QQC*)
Summary of the proof pattern

- Express interference that matters via auxiliary state — tokens;
- Capture past interference and results in auxiliary histories;
- Assume closed world to bound interference.
Not discussed today

- Full formal specification of the counting network;
- Formal proofs of QC and QQC properties for the network;
- Discussion on applying the technique for QC-queues;
- Spec and verification of `java.util.concurrent.Exchanger`;
- Verification of an exchanger client in the spirit of `concurrency-aware linearizability` (CAL).
To take away

Hoare-style Specifications for Non-linearizable Concurrent Objects

- Compositional — substitution principle;
- Syntactic proof method — inference rules;
- Uniform — reasoning about objects and their clients in the same proof system.

Specification is in the eye of the beholder.

Thanks!