Weak-Consistency Specification via Visibility Relaxation

Michael Emmi
SRI International

Constantin Enea
Univ. Paris Diderot
Motivation
Concurrent Objects

High-level abstractions
e.g. numeric & collection ADTs

Low-level performance
e.g. lock-free shared-memory access

Available on modern platforms
e.g. dozens in JDK

/**
 * a concurrent collection is thread-safe, but not governed by a single exclusion lock.
 */
package java.util.concurrent;

// we've considered these objects
class ConcurrentHashMap { … }
class ConcurrentSkipListMap { … }
class ConcurrentSkipListSet { … }
class ConcurrentLinkedQueue { … }
class LinkedTransferQueue { … }
class LinkedBlockingQueue { … }
class ConcurrentLinkedDeque { … }

// and there are several more
Weak Consistency

Performance optimization
avoid synchronization bottlenecks
weaken guarantees

Out in the wild
e.g. collections in JDK

Undermines reasoning
“Weakly consistent” is imprecise

package java.util.concurrent;
class ConcurrentSkipListSet { ... }

/**
 * Iterators and spliterators are
 * weakly consistent...
 *
 * They are guaranteed to traverse
 * elements as they existed upon
 * construction exactly once and may
 * (but are not guaranteed to) reflect
 * any modification subsequent to
 * construction.
 */
E.g. The Size Method

```java
new Thread(() -> {
    s.add(1);
    s.remove(2);
}).start();
```

```java
new Thread(() -> {
    s.add(2);
    var n = s.size();
}).start();
```

**Requirements?**
- allow \( n = 0 \)
- forbid \( n = -1, 42, 100, \ldots \)

**Generic methodologies?**
- not tied to sets nor sizes
- reuse existing functional spec

**ADT-admitted linearizations**
- add(1); remove(2); add(2); size() => 2
- add(1); add(2); remove(2); size() => 1
- add(1); add(2); size() => 2; remove(2)
- add(2); add(1); remove(2); size() => 1
- add(2); size() => 1; add(1); remove(2)
Visibility Relaxation

Axiomatic framework
Linearization + visibilities
Burckhardt et al.

Which criterion?
causal consistency
doesn’t allow $n = 0$
eventual consistency
doesn’t constrain $n$ at all

How to mix?
add & remove remain atomic

Linearization
add(1); add(2); remove(2); size() $\Rightarrow n$

<table>
<thead>
<tr>
<th>visibility of size</th>
<th>add(1)</th>
<th>add(2)</th>
<th>remove(2)</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
Contributions

Visibility Relaxation
an annotation language

Sequential happens-before consistency (SHBC)
effective consistency validation

Empirical study
derive JDK specifications

interface WeakSizeSet<E> {
    // complete visibility
    public boolean add(E elem);

    // complete visibility
    public boolean remove(E elem);

    // monotonic visibility
    public monotonic int size();
}
Visibility Relaxation
**Programs & Behaviors**

**Program Order (PO)**

per-thread invocation order

**Happens-Before (HB)**

PO with synchronization

**Outcome**

invocations’ return values

**Behavior**

HB with outcome

**Implementation**

maps programs to behaviors

---

Program

\{ add(1); remove(2) \} || \{ add(2); \textit{size()} \}

Implementation

\texttt{ConcurrentSkipListSet}

Behaviors/Outcomes

\[
\begin{array}{cccc}
\text{false} & \text{true} & \text{false} & 0 \\
\text{false} & \text{true} & \text{false} & 1 \\
\text{...}
\end{array}
\]
Linearizations

Linearization Order
- total order over invocations
- includes happens-before

Visibility Relation
- subsequence of linearized-before

ADT Consistency
- subsequence admits return value

---

Program
{ add(1); remove(2) } || { add(2); size() }

Happens-Before
add(1) \rightarrow remove(2)
add(2) \rightarrow size()

Linearization
add(2); add(1); remove(2); size()
visible to size

ADT sequence
add(2); remove(2); size() => 0
Predicates & Specifications

Visibility Predicates
lower bounds on visibility

Visibility Specification
one predicate per method

Consistency
only consider linearizations satisfying per-method predicates

weak
no constraints

basic
must see happened-before

monotonic
also must see those seen by happens-before

peer
also must see those which happened before seen

causal
also visibility is transitive

complete
must see all linearized before
E.g. The Size Method

Consistent w/ monotonic
size sees add(2)
and all seen by add(2)
i.e. none

Inconsistent w/ peer
size sees remove(2)
not HB-predecessor add(1)

Program
{ add(1); remove(2) } || { add(2); size() }

Happens-Before
add(1) -> remove(2)
add(2) -> size()

Linearization
add(2); add(1); remove(2); size()
visible to size

ADT sequence
add(2); remove(2); size() => 0
Assertion-Based Validation

Compute expected behaviors
for a given test program

Record observed behavior
return values & happens-before

Assert the inclusion
e.g. via hashing

```javascript
function expected({ po, hbs }, Impl, Spec) {
    for (let hb of hbs) {
        for (let { lin, vis } of hb.lins()) {
            if (!Spec.isSatisfied(lin, vis, hb))
                continue;
            let ret = {};
            for (let i of lin) {
                let seq = vis(i);
                let res = Impl.execute(seq);
                ret[i] = res[res.length - 1];
            }
            yield { hb, ret };}
    }
}
```
Sequential Happens-Before Consistency
vs. Linearizability

Real-Time (RT) Order
return action precedes call
platform agnostic

Happens-Before (HB) Order
platform dependent
e.g. Java volatile variables, locks

Sequential HB Consistency
linearizations of HB, not RT
extends SC from PO to HB
Real-Time

Runtime monitoring?
platform specs eschew guarantees
recording mechanisms interfere

Sound linearizability?
impossible w/o platform guarantees!

Leverage happens-before?
LIN becomes SHBC
Platform Properties

**Real-Time Soundness (RTS)**
happens-before implies real-time

**Real-Time Consistency (RTC)**
real-time implies happens-before (*without interference*)

**Real-Time Limit Consistency**
every admitted real-time order is captured* by a happens-before order

*given sufficient instrumentation
Real-Time Instrumentation

**Memory-Based**

- requires instruction barriers
- requires atomic read-call & ret-write
- requires location independence (PSO)

**Clock-Based**

- requires high-precision
- requires negligible latency
- requires atomic read-call & ret-read

```java
// invocation 1
boolean[] before1 = {
    done[I3],
    done[I4]
};
s.add(1);
done[I1] = true;

// invocation 2
boolean[] before2 = {
    done[I3],
    done[I4]
};
s.add(2);
done[I2] = true;

// reconstruct order
boolean[][] before = {
    before1,
    before2
};
```
Equivalences

LIN implies SHBC
for RTS platforms

SHBC implies LIN
per execution, on RTC platforms

SHBC implies LIN
per object, on RTCL platforms
Empirical Study
Hypotheses

Atomicity
JDK methods not generally atomic

Specification
with visibility annotations

Validation
SHBC uncovers violations
Empirical Setup

7 JDK collections
Maps, Sets, Queues, Deques

Random Test Generation
2 threads, 3–6 invocations, 1–2 values
100K programs per object

Stress Testing
1 second per test program

Simplification
without synchronization

<table>
<thead>
<tr>
<th>outcome</th>
<th>atomic?</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>null, 0, 0, 1</td>
<td>✓</td>
<td>949</td>
</tr>
<tr>
<td>null, 0, 1, 1</td>
<td>✓</td>
<td>746,263</td>
</tr>
<tr>
<td>null, 0, 1, null</td>
<td>✓</td>
<td>2,614,780</td>
</tr>
<tr>
<td>null, null, 1, 0</td>
<td>✓</td>
<td>14,833</td>
</tr>
<tr>
<td>null, null, 2, 0</td>
<td>✗</td>
<td>35</td>
</tr>
</tbody>
</table>
50+ non-atomic methods
roughly 40% of those tested

Some predictable
docs mention weak consistency
  e.g. size, iterator, elements, ...

Others unexpected
breaks internal invariants
  e.g. clear

weak-memory behaviors
  e.g. final keyword missing from peekLast, ...

<table>
<thead>
<tr>
<th>program / method</th>
<th>ConcurrentHashMap</th>
<th>outcome</th>
<th>frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>{put(0,0); put(1,1); put(1,1)}</td>
<td></td>
<td>{p</td>
<td>N,N,N,N,[]</td>
</tr>
<tr>
<td>{put(0,0);remove(1)}</td>
<td></td>
<td>{put(1,0);co</td>
<td>N,0,N,F</td>
</tr>
<tr>
<td>{get(1); containsValue(1)}</td>
<td></td>
<td>{put(1,</td>
<td>1,F,N,N,1</td>
</tr>
<tr>
<td>{put(0,1);put(1,0)}</td>
<td></td>
<td>{elements()}</td>
<td>N,N,[]</td>
</tr>
<tr>
<td>{put(0,1);put(1,0)}</td>
<td></td>
<td>{entrySet()}</td>
<td>N,N,[1=0]</td>
</tr>
<tr>
<td>{put(1,1)</td>
<td></td>
<td>{put(1,2); isEmpty()</td>
<td>N,1,T</td>
</tr>
<tr>
<td>{put(0,1);put(1,1)}</td>
<td></td>
<td>{keys()}</td>
<td>N,N,[1]</td>
</tr>
<tr>
<td>{keys()}</td>
<td></td>
<td>{put(0,1);put(1,1)}</td>
<td>[1],N,N</td>
</tr>
<tr>
<td>{put(1,0); put(1,1); mappingCount(</td>
<td>N,N,2,0</td>
<td>52 / 2,231,190</td>
<td></td>
</tr>
<tr>
<td>{put(1,0); put(1,1); size()}</td>
<td></td>
<td>{remo</td>
<td>N,N,2,0</td>
</tr>
<tr>
<td>{put(0,1);put(1,1)}</td>
<td></td>
<td>{toString()}</td>
<td>N,N,1=1</td>
</tr>
<tr>
<td>{put(0,1);put(1,0)}</td>
<td></td>
<td>{values()}</td>
<td>N,N,[]</td>
</tr>
</tbody>
</table>
JDK Specification

84 complete
mostly single-element operations

29 monotonic
meaning of “weakly consistent?”

3 weak
isEmpty, toArray, toString

18 inconsistent
most indicate bugs
few are intended

e.g. ConcurrentHashMap

complete
put, get, remove, containsKey, replace, putIfAbsent

monotonic
contains, containsValue, keys, values, elements, entrySet, keySet, toString

weak
isEmpty

inconsistent
clear, size, mappingCount
JDK Validation

**SHBC is effective**
identifies violations w/o real-time

**SHBC is efficient**
millions of executions per second

**Randomness useful**
e.g. unexpected argument combos
Conclusion

Visibility relaxation
generic yet precise semantics

Sequential happens-before consistency
efficient validation
integration with modern platforms

Empirical study
effective specification and validation