LineUp: Automatic Thread Safety Checking

Madan Musuvathi

Joint work with
Sebastian Burckhardt, Chris Dern, Roy Tan
Motivation

- Concurrent applications are common place
- Correct synchronization is tricky
- Modular programming with “thread-safe” components
  - Hide synchronization details within each component
  - Expose a simpler sequential interface
  - Can be called concurrently without additional synchronization
Example of a Thread-Safe Component

- Component = state + set of operations
- Clients can concurrently call these operations without additional synchronization
- Concurrent Queue still behaves like a “queue”
In this talk

- A precise formalization of thread safety
  - Deterministic Linearizability

- An automatic method for checking thread safety
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
Assert( ? )
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
q.size() is 0 or 1
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
```

Assert:
- `q.size()` is 0 or 1
- `t` is 10 or <fail>
Let’s Write a Test

```java
q = new ConcurrentQueue();

q.push(10);

t = q.pop();

Assert:
    t = fail && q.size() = 1 &&
        q.peek() == 10 ||
        t = 10 && q.size() = 0
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10); t = q.pop();
q.push(20); u = q.pop();

Assert ( ? )
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
q.push(20);
u = q.pop();
```

Assert:
- `q.size() == 0 && t = 10 || t = 20` &&
- `u = 10 || t = 20` &&
  - `u != t`
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);

q.push(30);
q.push(40);
u1 = q.peek();
u2 = q.pop();

v1 = q.pop();
v2 = q.peek();
q.push(50);
q.push(60);

Assert (?)
```
We want to simply say...

```
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
q.push(50);
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

Assert:
ConcurrentQueue behaves like a queue
Linearizability [Herlihy & Wing ‘90]

ConcurrentQueue behaves like a queue

Concurrent behaviors of ConcurrentQueue are consistent with a sequential specification of a queue
Sequential Specification of a Queue

- A concise way to specify the behavior when operations are performed one at a time

- And so on ...
Linearizability [Herlihy & Wing '90]

ConcurrentQueue behaves like a queue

Concurrent behaviors of ConcurrentQueue are consistent with a sequential specification of a queue

Every operation appears to occur atomically at some point between the call and return
Linearizability

- Component is *linearizable* if every operation appears to occur atomically at some point between the call and return.

Thread 1
- push 10
- return
- push 20
- return

Thread 2
- pop
- return 10

Thread 3
- pop
- return “empty”
A Linearizability Violation

- History shown below is not linearizable

Thread 1
- push 20
- return
- pop
- return 20

Thread 2
- push 10
- return
- pop
- return empty
Solution using Linearizability

```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
qu.push(40);
u2 = q.pop();
q.push(50);
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

Assert:
Every concurrent behavior (for this test) is linearizable
Providing Sequential Specification is Hard

- **Complexity:**
  - API for java.util.concurrent.BlockingQueue\(<T>\) contains 28 functions

- **Logic:**
  - Even simple specifications require sophisticated logics
  - Need to make the accessible to programmers

- **Insight**
  - We have the code for the component
  - Learn the specification automatically
Deterministic Linearizability

ConcurrentQueue behaves like a queue

Concurrent behaviors of ConcurrentQueue are consistent with a sequential specification of a queue

Some deterministic sequential specification of a queue

Learn this by observing the code under sequential runs
The LineUp Proposition

- Be complete
  - all reported violations are conclusive refutations

- Be automatic
  - User specifies component interface (gives list of operation calls), tool does the rest

- Be reasonably sound
  - quantify unsoundness sources (missed bugs)
  - Demonstrate empirically that we find enough real bugs
Implementation (part 1: generate Unit Tests)

Example of a concurrent unit test:

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.Add(10)</td>
<td>q.TryTake()</td>
<td>q.TryTake()</td>
</tr>
<tr>
<td>q.Add(20)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q.TryTake()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>q.Clear()</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Example: queue operations

q.Add(10), q.Add(20), q.TryTake(), q.Clear()
Implementation (part 2: check Each Test)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
<th>Thread 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.Add(10)</td>
<td>q.Remove()</td>
<td>q.Remove()</td>
</tr>
<tr>
<td>q.Clear()</td>
<td>q.Clear()</td>
<td>q.Clear()</td>
</tr>
<tr>
<td>q.Add(10)</td>
<td></td>
<td>q.Add(20)</td>
</tr>
</tbody>
</table>

unit test

2-Phase Check

reported violating execution

complete successfully

component under test (bytecode)
THE Two-Phase check (Phase 1)

- Enumerate Serial Histories
  - record all observations (incl. return values) in file
- we are synthesizing a sequential specification!

Queue Implementation (Bytecode)

Unit Test

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add(200)</td>
<td>Add(400)</td>
</tr>
<tr>
<td>TryTake()</td>
<td>TryTake()</td>
</tr>
</tbody>
</table>

CHESS stateless model checker

200   400
200   400
200   400
200   400
200   400
400   200
400   200
THE Two-Phase check (Phase 2)

<table>
<thead>
<tr>
<th>Queue Implementation (Bytecode)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit Test</td>
</tr>
<tr>
<td>Thread 1</td>
</tr>
<tr>
<td>Add(200) TryTake()</td>
</tr>
</tbody>
</table>

# of fair executions is finite! [PLDI 07]

**CHESS stateless model checker**

- **Enumerate Concurrent Histories**
  - check against specification
  - report violations
The Two-Phase Check: theorems

- Completeness
  - A counterexample refutes deterministic linearizability (i.e. proves that no deterministic sequential specification exists).

- Restricted Soundness
  - If the component is not deterministically linearizable, there exists a unit test that fails.

- How sound in practice?
  E.g. how good are 100 random 3x3 tests?
### Results: Phase 1 / Phase 2

<table>
<thead>
<tr>
<th>Class</th>
<th>Root Causes</th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Smallest Failing testcase</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>#histories</td>
<td>time [min]</td>
<td>#</td>
</tr>
<tr>
<td></td>
<td></td>
<td>avg</td>
<td>max</td>
<td>avg</td>
</tr>
<tr>
<td>Lazy Initialization</td>
<td></td>
<td>60</td>
<td>68</td>
<td>0.07</td>
</tr>
<tr>
<td>ManualResetEvent(PE)</td>
<td>A</td>
<td>828</td>
<td>1680</td>
<td>3.31</td>
</tr>
<tr>
<td>ManualResetEvent</td>
<td></td>
<td>76</td>
<td>133</td>
<td>0.07</td>
</tr>
<tr>
<td>SemaphoreSlim</td>
<td></td>
<td>1511</td>
<td>1680</td>
<td>3.91</td>
</tr>
<tr>
<td>CountdownEvent</td>
<td></td>
<td>410</td>
<td>1680</td>
<td>1.20</td>
</tr>
<tr>
<td>ConcurrentDictionary</td>
<td></td>
<td>1678</td>
<td>1680</td>
<td>7.02</td>
</tr>
<tr>
<td>ConcurrentQueue(PE)</td>
<td>D</td>
<td>1680</td>
<td>1680</td>
<td>6.00</td>
</tr>
<tr>
<td>ConcurrentQueue</td>
<td></td>
<td>1680</td>
<td>1680</td>
<td>6.37</td>
</tr>
<tr>
<td>ConcurrentLinkedList(PE)</td>
<td>B</td>
<td>1674</td>
<td>1680</td>
<td>8.54</td>
</tr>
<tr>
<td>ConcurrentStack</td>
<td></td>
<td>1680</td>
<td>1680</td>
<td>3.49</td>
</tr>
<tr>
<td>BlockingCollection(PE)</td>
<td>C</td>
<td>850</td>
<td>1680</td>
<td>4.09</td>
</tr>
<tr>
<td>BlockingCollection</td>
<td></td>
<td>259</td>
<td>501</td>
<td>0.07</td>
</tr>
<tr>
<td>ConcurrentBag(PE)</td>
<td></td>
<td>1680</td>
<td>1680</td>
<td>4.18</td>
</tr>
<tr>
<td>ConcurrentBag</td>
<td>I</td>
<td>1680</td>
<td>1680</td>
<td>4.05</td>
</tr>
<tr>
<td>TaskCompletionSource(PE)</td>
<td>E</td>
<td>1432</td>
<td>1680</td>
<td>7.30</td>
</tr>
<tr>
<td>TaskCompletionSource</td>
<td></td>
<td>1209</td>
<td>1680</td>
<td>5.98</td>
</tr>
<tr>
<td>CancellationTokenSource</td>
<td></td>
<td>1680</td>
<td>1680</td>
<td>4.49</td>
</tr>
<tr>
<td>Barrier</td>
<td>F</td>
<td>1166</td>
<td>1680</td>
<td>4.10</td>
</tr>
<tr>
<td>Class</td>
<td>Root Causes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>--------------------------------------</td>
<td>-------------</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Non-det</td>
<td>Non-lin.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lazy Initialization</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ManualResetEvent (PE)</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ManualResetEvent</td>
<td>C</td>
<td>G</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SemaphoreSlim</td>
<td>D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CountdownEvent</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentDictionary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue (PE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentLinkedList (PE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentStack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockingCollection (PE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlockingCollection</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag (PE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource (PE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CancellationTokenSource</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Each letter is a separate root cause
Example: incorrect CAS

```c
volatile int state;

... int localstate = state;
int newstate = f(state); // compute new value
compare_and_swap(&state, localstate, newstate);
...```
<table>
<thead>
<tr>
<th>Class</th>
<th>Root Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bugs</td>
</tr>
<tr>
<td></td>
<td>Intentional</td>
</tr>
<tr>
<td></td>
<td>nondet.</td>
</tr>
<tr>
<td></td>
<td>nonlin.</td>
</tr>
<tr>
<td>Lazy Initialization</td>
<td>A</td>
</tr>
<tr>
<td>ManualResetEvent(PE)</td>
<td></td>
</tr>
<tr>
<td>ManualResetEvent</td>
<td></td>
</tr>
<tr>
<td>SemaphoreSlim</td>
<td></td>
</tr>
<tr>
<td>CountdownEvent</td>
<td></td>
</tr>
<tr>
<td>ConcurrentDictionary</td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue</td>
<td></td>
</tr>
<tr>
<td>ConcurrentLinkedList(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentStack</td>
<td></td>
</tr>
<tr>
<td>BlockingCollection(PE)</td>
<td></td>
</tr>
<tr>
<td>BlockingCollection</td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag</td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource(PE)</td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource</td>
<td></td>
</tr>
<tr>
<td>CancellationTokenSource</td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td>F</td>
</tr>
</tbody>
</table>

Results

- Cancel is not linearizable (may delay past return)
- Barrier is not linearizable (rendezvous not equivalent to any interleaved commit)
<table>
<thead>
<tr>
<th>Class</th>
<th>Root Causes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Bugs</td>
</tr>
<tr>
<td></td>
<td>nondet.</td>
</tr>
<tr>
<td>Lazy Initialization</td>
<td>A</td>
</tr>
<tr>
<td>ManualResetEvent(PE)</td>
<td>D</td>
</tr>
<tr>
<td>ManualResetEvent</td>
<td></td>
</tr>
<tr>
<td>SemaphoreSlim</td>
<td></td>
</tr>
<tr>
<td>CountdownEvent</td>
<td></td>
</tr>
<tr>
<td>ConcurrentDictionary</td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentQueue</td>
<td></td>
</tr>
<tr>
<td>ConcurrentLinkedList(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentStack</td>
<td></td>
</tr>
<tr>
<td>BlockingCollection(PE)</td>
<td></td>
</tr>
<tr>
<td>BlockingCollection</td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag(PE)</td>
<td></td>
</tr>
<tr>
<td>ConcurrentBag</td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource(PE)</td>
<td></td>
</tr>
<tr>
<td>TaskCompletionSource</td>
<td></td>
</tr>
<tr>
<td>CancellationTokenSource</td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td></td>
</tr>
</tbody>
</table>

Results

- **nondet.**: bag may return any element
- **nondet.**: (weak spec) Count may return 0 and TryTake may return false even if not empty
(A) Incorrect use of CAS causes state corruption. (B) RemoveLast() uses an incorrect lock-free optimization. (C) Call to SemaphoreSlim includes a timeout parameter by mistake. (D) ToArray() can livelock when crossing segment boundaries. Note that the harness for this class performs a particular pre-test sequence (add 31 elements, remove 31 elements). (E) Insufficient locking: thread can get preempted while trying to set an exception. (F) Barrier is not a linearizable data type. Barriers block each thread until all threads have entered the barrier, a behavior that is not equivalent to any serial execution. (G) Cancel is not a linearizable method: The effect of the cancellation can be delayed past the operation return, and in fact even past subsequent operations on the same thread. (H) Count() may release a lock it does not own if interleaved with Add(). (I) Bag is nondeterministic by design to improve performance: the returned value can depend on the specific interleaving. (J) Count may return 0 even if the collection is not empty. The specification of the Count method was weakened after Line-Up detected this behavior. (K) TryTake may fail even if the collection is not empty. The specification of the TryTake method was weakened after Line-Up detected this behavior. (L) SetResult() throws the wrong exception if the task is already reserved for completion by somebody else, but not completed yet.
Related Work / Other Approaches

- Two-Phase Check
  - CheckFence [PLDI 07], less automatic, only SC

- Race Detection
  - The bugs we found do not manifest as data races

- Atomicity (Conflict-Serializability) Checking
  - Many false alarms (programmers are creative)

- Traditional Linearizability Verification
  - less automatic (proofs, specs, commit points)
  - does not detect incorrect blocking
Extending Classic Linearizability

We check deadlocks more tightly.

- **[Herlihy, Moss]**: Deadlocked histories are considered linearizable even if operations do not block in sequential specification.

- **[Line-Up]**: Deadlocked histories qualify as linearizable *only* if sequential specification allows all pending operations to block.
Comparison to Atomicity Checking

- Dynamic Atomicity Checking
  - Reports not conflict-serializable executions
  - stronger: implies deterministic linearizability

- Produced too many false warnings
  - Implementations exhibited not conflict-serializable executions, for perfectly legitimate reasons
Comparison to Race Detection

- Popular: Data Race Detection
- Not effective in this case
  - Code was data-race free due to disciplined use of ‘volatile’ qualifier
  - Bugs did not manifest as data races
Conclusion

• Find concurrency bugs ✓
  • Cover as many types as possible ✓
  • Simple mistakes (e.g. forgot to lock) ✓
  • Design mistakes (bugs in lock-free algorithm) ✓

• Make good use of tester’s time ✓
  • Automation important (much to test, little time) ✓
  • Black-box: do not read code ✓
  • Give concise, precise “counterexamples” ✓
Conclusion

• Success:
  • Found tricky concurrency bugs in public production code
  • Bugs were fixed

• Deterministic linearizability is a useful thread-safety criterion
  • Can be checked automatically
  • In our case, better than race detection or atomicity checking

• High-value addition to CHESS
  • Dramatically automates the process
  • CHESS source release now available!
**Strawman Proposal**

- Write lots of testcases like this one: (where q is a thread-safe queue)
  
<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>q.AddFirst(10)</td>
<td>q.AddLast(20)</td>
</tr>
<tr>
<td>q.RemoveLast()</td>
<td>q.RemoveFirst(10)</td>
</tr>
</tbody>
</table>

- Use Chess (stateless model checker) to cover all interleavings

**Pro:**
- can find bugs this way (test above found bug!)

**Contra:**
- not clear how to write these tests. Automation?
- This test deadlocked incorrectly

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mre.Wait();</code></td>
<td><code>mre.Set(); mre.Reset();</code></td>
</tr>
<tr>
<td></td>
<td><code>mre.Set();</code></td>
</tr>
</tbody>
</table>

- revealing this pernicious typo.
testing thread-safety of components

- Active Components
  - Contain threads
  - Initiate calls to other components

- Passive Components
  - Do not contain threads
  - Are called by other components
  - We want to test these in isolation
Passive components

- State is hidden
- Read/Change state via *operations* only
- Either not thread-safe...
  - Caller must use synchronization to avoid concurrent operations
- ... or thread-safe
  - Implementation contains sufficient synchronization to handle concurrent operations
Thread-Safe PASSIVE Components

- Various Implementations Possible
  - Single global lock
  - Fine-grained locking
  - Lock-free algorithms (e.g. compare-and-swap)
  - Replication
  - Request queue & internal worker thread
  - All of the above
- Implementations should share common ‘abstract’ behavior
Let’s Write a Test

q = new ConcurrentQueue();

q.push(10);

t = q.pop();

Assert( ? )
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
q.size() is 0 or 1
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
Assert:
q.size() is 0 or 1
and t is 10 or <fail>
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();

Assert:
t = fail && q.size() = 1 && q.peek() == 10 || t = 10 && q.size() = 0
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
q.push(20);
u = q.pop();
```

Assert ( ? )
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t = q.pop();
q.push(20);
u = q.pop();
```

**Assert:**
```java
q.size() == 0 &&
t = 10 || t = 20 &&
u = 10 || t = 20 &&
u != t
```
Let’s Write a Test

```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

Assert (?)
Wouldn’t it be nice if we could just say…

```
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
```

```
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
```

```
v1 = q.pop();
q.push(50);
v2 = q.peek();
q.push(60);
```

**Assert:**
ConcurrentQueue behaves like a queue
Informally, this is “thread safety”

ConcurrentQueue behaves like a queue

A piece of code is thread-safe if it functions correctly during simultaneous execution by multiple threads.
Formally, this is “Linearizability” [Herlihy & Wing ‘90]

ConcurrentQueue behaves like a queue

Concurrent behaviors of ConcurrentQueue

are consistent with

a sequential specification of a queue

Every operation appears to occur atomically at some point between the call and return
So, simply check linearizability

```
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
q.push(50);
v1 = q.pop();
v2 = q.peek();
q.push(60);
```

Assert:
Linearizability wrt a given sequential specification
Automatic Thread Safety Checking

```java
q = new ConcurrentQueue();
q.push(10);
t1 = q.pop();
t2 = q.peek();
q.push(20);
q.push(30);
u1 = q.peek();
q.push(40);
u2 = q.pop();
v1 = q.pop();
v2 = q.peek();
v2 = q.pop();
q.push(60);
```

Assert:
**ConcurrentQueue is Thread Safe**

Automatically learn how “a queue” behaves
Conclusions

• Beware of race conditions when you are designing your programs
  • Think of all source of nondeterminism
  • Reason about the space of program behaviors

• Use tools to explore the space
  • Cuzz: Randomized algorithm for large programs
  • CHESS: systematic algorithm for unit testing
  • Thread-safety as a correctness criterion