Automatic Atomicity Verification for Clients of Concurrent Data Structures

Mohsen Lesani

Todd Millstein   Jens Palsberg

Concurrent Data Structures

```
class ConcurrentHashmap<K, V> { // data structure
   V get(K k) { ... }
   void put(K k, V v) { ... }
   V remove(K k) { ... }
   V putIfAbsent(K k, V v) { ... }
   boolean replace(K k, V ov, V nv) { ... }
}
```

Atomicity for single method calls

Non-atomicity of multiple method calls

Client Composing Classes

```
class ConcurrentHistogram<K> {   // client
   private ConcurrentHashMap<K, Integer> m;

   V get(K k) {
      return m.get(k); }

   Integer inc(K key) {
      Integer i = m.get(key);
      if (i == null) {
         m.put(key, 1);
         return 1;
      } else {
         Integer ni = i + 1;
         m.put(key, i, ni);
         return ni;
}  }  }
```

Client Composing Classes

```
class ConcurrentHistogram<K> {   // client
   private ConcurrentHashMap<K, Integer> m;

   V get(K k) {
      return m.get(k); }

   Integer inc(K key) {
      while (true) {
         Integer i = m.get(key);
         if (i == null) {
            Integer r = m.putIfAbsent(key, 1);    // *
            if (r == null)
               return 1;
         } else {
            Integer ni = i + 1;
            boolean b = m.replace(key, i, ni);    // *
            if (b)
               return ni;
}  }  }  }
```

Client Composing Classes

- Atomicity violations in real-world applications

- Testing
- Verification

Condensability

A modular sufficient condition for atomicity of a common class of client classes that use a single data structure.

Purity

```
Integer inc(K key) {
    while (true) {
        Integer i = m.get(key);
        if (i == null) {
            Integer r = m.putIfAbsent(key, 1);     // *
            if (r == null)
                return 1;
        } else {
            Integer ni = i + 1;
            boolean b = m.replace(key, i, ni);     // *
            if (b)
                return ni;
}  }  }
```

Purity

```
Integer inc(K key) {
    while (true) {
        Integer i = m.get(key);
        if (i == null) {
            Integer r = m.putIfAbsent(key, 1);    // *
            if (r == null)
                return 1;
        } else {
            Integer ni = i + 1;
            boolean b = m.replace(key, i, ni);    // *
            if (b)
                return ni;
}   }   }
```

Paths

```
Integer i = m.get(key);
if (i == null) {
   Integer r = m.putIfAbsent(key, 1);    // *
   if (r == null)
      return 1;
} else {
   Integer ni = i + 1;
   boolean b = m.replace(key, i, ni);    // *
   if (b)
      return ni;
}
```

```
// First path
Integer i = m.get(key);
assume (i == null);
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
return 1;
```

```
// Second path
Integer i = m.get(key);
assume (!(i == null));
Integer ni = i + 1;
Boolean b = m.replace(key, i, ni);
assume (b);
return ni;
```

Moverness

```
// First path
Integer i = m.get(key);
assume (i == null);
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
return 1;
```

Moverness

```
// First path
Integer i = m.get(key);
assume (i == null);
                                            m.put(key, v);
                                            m.remove(key);
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
return 1;
```

Moverness

```
// First path

                                            m.put(key, v);

Integer i = m.get(key);

assume (i != null);

                                            m.remove(key);

Integer r = m.putIfAbsent(key, 1);

assume (r == null);

return 1;
```

Moverness

```
// First path
Integer i = m.get(key);
assume (i == null);
                                            m.put(key, v);
                                            m.remove(key);
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
return 1;
```

Moverness

```
// First path
Integer i = m.get(key);
assume (i == null);
                                            m.put(key, v);
Integer r = m.putIfAbsent(key, 1);
assume (r != null);
                                            m.remove(key);
return 1;
```

Moverness

```
// First path
Integer i = m.get(key);
assume (i == null);
                                            m.put(key, v);
                                            m.remove(key);
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
return 1;
```

Condensability

```
                                                    m.call();
// m0
Integer i = m.get(key);
assume (i == null);
// m1
                                                    m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                    m.call();
```

Condensability

```
                                                      m.call();
// m0
Integer i = m.get(key);
assume (i == null);
// m1
                                                      m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                      m.call();
```

Condensability

```
                                                    m.call();
// m0
Integer i = m.get(key);
assume (i == null);
// m1
                                                    m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                    m.call();
```

Condensability

```
                                                    m.call();
// m0
Integer i = m.get(key);
assume (i == null);
// m1
                                                    m.call();
// m2
inc()
// m3
return 1;
                                                    m.call();
```

Condensability implies Atomicity

```
                                                     m.call();
// m0
Integer i = m.get(key);
assume (i == null);
// m1
                                                     m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                     m.call();
```

Condensability implies Atomicity

```
                                                      m.call();
// m0

// m0
                                                      m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                      m.call();
```

Condensability implies Atomicity

```
                                                  m.call();
// m0



// m0
                                                  m.call();
// m2
____________________________________
inc()
____________________________________
// m3
return 1;
                                                  m.call();
```

Condensability

A client object is condensable if every execution of every method of it is condensable.

Theorem: Every condensable object is atomic.

Condensed Execution

The sequential execution of the entire method at the condensation point

Condensed Execution

```
                                                    m.call();
// m0



// m0
                                                    m.call();
// m2
Integer r = m.putIfAbsent(key, 1);
assume (r == null);
// m3
return 1;
                                                    m.call();
```

Condensed Execution

```
m.call();

// m0

// m0

m.call();

// m2
```

```
Integer i = m.get(key);
if (i == null) {
   Integer r = m.putIfAbsent(key, 1);
   if (r == null)
      return 1;
} else {
   Integer ni = i + 1;
   boolean b = m.replace(key, i, ni);
   if (b)
      return ni;
}
```

```
m.call();
```

Condensed Execution

```
                                                  m.call();
// m0

// m0
                                                  m.call();
// m2
```

```
Integer i = m.get(key);
// m2
if (i == null) {
   Integer r = m.putIfAbsent(key, 1);
   if (r == null)
      return 1;
}
```

```
                                                  m.call();
```

Condensed Execution

```
                                                         m.call();
// m0

// m0
                                                          m.call();
// m2
______________________________________________________
Integer i = m.get(key);
// m2
Integer r = m.putIfAbsent(key, 1);
// m3
if (r == null)
   return 1;
______________________________________________________
                                                           m.call();
```

Condensed Execution

```
                                                        m.call();
// m0

// m0
                                                         m.call();
// m2
```

```
Integer i = m.get(key);
// m2
Integer r = m.putIfAbsent(key, 1);
// m3
return 1;
```

```
                                                          m.call();
```

Checking Condensability

- Representing as constraints
  - Axioms for the properties of the base data structure
  - Paths
  - Condensability conditions

Sequential Specification

```
class ConcurrentHashMap<K, V> { // data structure
   V get(K k) { /*..*/ }
   void put(K k, V v) { /*..*/ }
   V putIfAbsent(K k, V v) { /*..*/ }
}
```

$$((v, m) = m.get(k)) \Rightarrow (v = m(k) \wedge m' = m)$$

$$(m' = m.put(k, v)') \Rightarrow (m' = m[k \mapsto v])$$

$$\begin{aligned}&(m', v') = m.putIfAbsent(k, v) \Rightarrow \\ &\quad v' = m(k) \wedge \\ &\quad ((m(k) = null) \ \wedge \ (m' = m[k \mapsto v])) \vee \\ &\quad (\neg(m(k) = null) \ \wedge \ (m = m'))\end{aligned}$$

Constraints

Assertions:

Let $P_i = (b, \overline{m}, r)$:

1. b

Forall k: $0 \leq k < |\overline{m}|$

Let $m_k = (y = o.n(x))$:

2. $(o_{2*k+1}, y) = o_{2*k}.n(x)$

Forall j: $0 \leq j < |P|$

Let $P_j = (b^j, \overline{m^j}, r^j)$:

3. $p^j = p \wedge$

4. $o_0^j = o_{2*l}$

Forall k: $0 \leq k < |\overline{m^j}|$

Let $m_k = (y = o.n(x))$:

5. $(o_{k+1}^j, y^j) = o_k^j.n(x^j)$

6. $b^j \Rightarrow$

$post_s = o^j_{|\overline{m^j}|} \wedge$

$ret_s = r^j$

Obligations:

Let $P_i = (b, \overline{m}, r)$:

Forall k: $0 \leq k < |\overline{m}|, k \neq l$

7. $o_{2*k} = o_{2*k+1} \wedge$

8. $post_s = o_{2*l+1} \wedge$

9. $ret_s = r$

p : Input parameter

x, y, r, ret_s : Variable

$o, post_s$: Object state variable

b : Condition

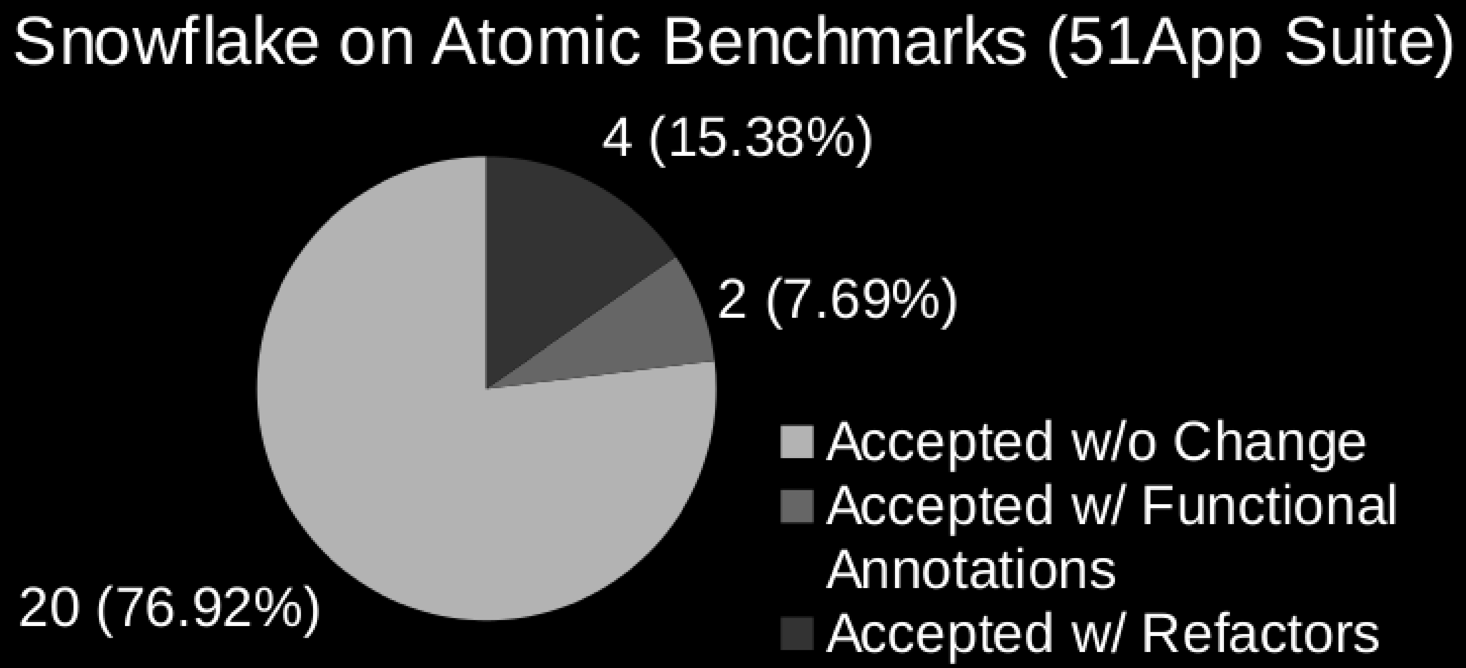
Snowflake

Automatic Verification Tool for Atomicity

- Input
  - A client class in Java
  - Specification of the used data structure
  - Functional annotations
- Generates the set of proof obligations that are sufficient for condensability.
- Uses Z3 SMT solver to solve the constraints.
- If the proof obligations are discharged, the method is verified to be atomic.

Results

Snowflake rejected all the 86 non-atomic benchmarks.



Thanks for your attendance.

Condensability

Consider a client method *M* that uses an atomic object *o*. Intuitively, a call to *M* in a concurrent execution *e* is condensable if there is a method call m in *M*'s execution on *o* such that

- All the other method calls other than m in *e* are accessors
- the sequential (condensed) execution of the entire method *M* at the place of m in *e* results in
  - the same final state of *o* as *m* and
  - the same return value as the original execution of *M*.

A client object is condensable if every execution of every method of it is condensable.

Theorem: Every condensable object is atomic.