Automatic Atomicity Verification for Clients of Concurrent Data Structures

UCLA Technical Report #140011 *

Mohsen Lesani Todd Millstein Jens Palsberg

University of California, Los Angeles
{lesani, todd, palsberg}@cs.ucla.edu

Contents

* This is the companion technical paper to the paper with the same title published at CAV'14.

```
class AtomicMap<K, V> {
  V get(K k) { /*..*/ }
  void put(K k, V v) { /*..*/ }
  V putIfAbsent(K k, V v) { /*..*/ }
  boolean replace(K k, V ov, V nv) { /*..*/ }
}
```

```
class AtomicHistogram<K> {
  private AtomicMap<K, Integer> m;

  V get(K k) {
    return m.get();
  }
  // Other methods similarly delegated.

  Integer buggyInc(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+1);
      return ni;
    }
  }
}
```

Figure 1. The classes `AtomicMap` and `AtomicHistogram` with a buggy implementation of atomic increment

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

Figure 2. A correct implementation of atomic increment

1. Example

We now illustrate our static verification technique for atomicity of composing methods with an example.

For an execution X and an object o, $X|o$ denotes the sub-execution of X that includes all the method calls on o in X. An execution is sequential if it involves no interleaved method calls. Two executions are equivalent if the method calls (including the arguments and the return values) of the two executions are the same. An object is atomic, if every execution of every program is atomic for the object. A concurrent execution X of a program p is atomic for an object o, if there is an execution S of p (called the justifying execution) such that $S|o$ is sequential and $X|o$ and $S|o$ are equivalent. We defer the real-time-preservation requirement to later sections.

Now consider the atomic map class `AtomicMap<K, V>` from keys of type `K` to values of type `V` with the interface depicted in

$$(m', v) = m.get(k)$$
$$\Rightarrow$$
$$(\mathcal{A}_0)\ v = m(k)$$
$$(\mathcal{A}_1)\ m' = m$$

$$(m', v') = m.put(k, v)$$
$$\Rightarrow$$
$$(\mathcal{A}_2)\ m' = m[k \mapsto v]$$

Figure 3. Axioms for `get` and `put` methods

```
V putIfAbsent(K k, V v) {
  atomic {
    V v1 = get(k);
    if (v1 == null)
      put(k, v);
    return v1;
  }
}
```

```
boolean replace(K k, V ov, V nv) {
  atomic {
    V v = get(k);
    if (v != null && v.equals(ov)) {
      put(k, nv);
      return true;
    } else
      return false;
  }
}
```

Figure 4. The specification of `AtomicMap` in terms of the basic map interface

Figure 1. Figure 3 depicts the axioms characterizing the behavior of `get` and `put` methods. We use the notation $(m_2, v_2) = m_1.n(v_1)$ to denote the execution of method n with the argument v_1 on the abstract map state m_1 that results in the abstract map state m_2 and the returns value v_2. The abstract map state is a function that maps every key in the map to its value and every key not in the map to $null$. We use $m(k)$ to denote the value that the abstract map state m maps the key k to. We use $m[k \mapsto v]$ to denote a map state that maps the key k to the value v and otherwise agrees with the map state m. The axiom of the `get` method states that its returned value is the value of the input key in the pre-state and that the post-state is equal to the pre-state. The axiom of the `put` method states that the post-state is the pre-state updated with the mapping of the input key to the input value.

The documentation of concurrent classes usually presents the sequential specification of conditional methods. For example, the the sequential specification of `putIfAbsent` and `replace` methods is represented in terms of the `get` and `put` methods in Figure 4. The axioms for the `putIfAbsent` and `replace` methods in Figure 5 can be immediately derived from their specification in Figure 4 and the axioms of the `get` and `put` methods in Figure 3. The axiom for the `putIfAbsent` method states that the mapping of the input key k is updated to the input value v, if the pre-value of k is $null$. Otherwise, the map state remains unchanged. The old mapping for the key k is always returned. The axiom for the `replace` method states that the mapping of k is updated to the value v' if it is mapped to v. Otherwise, the map state remains unchanged. The return value indicates whether the map is changed.

Now, consider the atomic histogram class `AtomicHistogram<K>` that is essentially a map from keys of type `K` to values of type

$(m_2, v') = m_1.putIfAbsent(k, v)$
$\Rightarrow$
$(\mathcal{A}_3)\ v' = m_1(k)\ \wedge$
$(\mathcal{A}_4)\ ((m_1(k) = null)\ \wedge\ (m_2 = m_1[k \mapsto v]))\ \vee$
$(\neg(m_1(k) = null)\ \wedge\ (m_1 = m_2))$

$(m_2, b) = m_1.replace(k, v, v')$
$\Rightarrow$
$(\mathcal{A}_5)$
$((m_1(k) \neq null) \wedge (m_1(k) = v) \wedge (m_2 = m_1[k \mapsto v']) \wedge b) \vee$
$(((m_1(k) = null)\ \vee\ (m_1(k) \neq v))\ \wedge\ (m_1 = m_2)\ \wedge\ \neg b)$

Figure 5. Axioms for `putIfAbsent` and `replace` methods

`Integer` with an additional method `inc` that increments the value of a key. Our `AtomicHistogram<K>` depicted in Figure 1 provides the full interface of `AtomicMap<K, Integer>` through delegation but could hide part of it as well. The method `inc` is depicted in Figure 2.

Consider an instance `h` of class `AtomicHistogram<K>`. The object `h` has an instance `m` of class `AtomicMap<K, Integer>`. The methods `get`, `putIfAbsent` and `replace` of `h` mirror corresponding methods of `m`. We have that every instance of `AtomicMap<K, V>` is atomic. Thus, every execution on `h` that accesses only the interface `get`, `putIfAbsent` and `replace` (excluding `inc`) is atomic. The method `inc` implements atomic increment of the value of the input key using the interface of `m`. We want to show that every execution on `h` including the `inc` method is atomic. We call `inc` the composing method and `m` the basic object.

Loops in synchronization methods are usually pure [5, 16]. At a high level, a loop is pure if only the last iteration of the loop has externally observable effects and thus previous iterations can be safely removed from the execution history. Purity is an effective way to reduce verification of atomicity to loop-free programs. We adopt this notion and explain how we apply and check it in the implementation section. In fact, the while loop in the `inc` method of Figure 2 is pure. Therefore, we concentrate on a version of the method called the *exceptional variant* where the loop is replaced by its last iteration. Consider the exceptional variant of the `inc` method in Figure 6. Now, let us decompose it to two straight-line programs that represent the return paths. The two paths are depicted in Figure 7. Note that the paths are in the static single assignment (SSA) form. Our tool converts paths to the SSA form if they are not already in this form.

Every path of the `inc` method takes effect at a single method call during its execution. This method call is called the atomicity point. We use heuristics to find the atomicity points. For example, a successful mutating method on the base object is usually the atomicity point. If the heuristics fail, the static analysis process can be repeated for each method call on the base object in the path. The atomicity points are annotated in the paths of Figure 7.

Let us consider an execution X of a program p. We show that X is atomic for h. We consider each path in turn. Let us assume that X includes an execution of the first path of the `inc` method.

Unfortunately, moverness cannot prove the atomicity of the path. Both the `get` and `putIfAbsent` method calls return `null`. Therefore, neither of them find the key `key` in the map `m`. Thus, executing the `get` method call in the pre-state of the `putIfAbsent` method call returns the same value `null`. But this doesn't mean that the `get` method can move right to the `putIfAbsent` method call. There can be method calls by other threads that `put` the key `key` to `m` and then `remove` it between the `get` and `putIfAbsent` method calls. In other words, the ABA problem can occur for the key `key` in the map `m`. The `get` method call is not a right mover to the

```
1  Integer inc(K key) {
2    Integer i = m.get(key);
3    if (i == null) {
4      Integer r = m.putIfAbsent(key, 1); // @P
5      if (r == null)
6        return 1;
7    } else {
8      Integer ni = i + 1;
9      Boolean b = m.replace(key, i, ni); // @P
10     if (b)
11       return ni;
12   }
13 }
```

Figure 6. The exceptional variant of `inc`

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

```
// Second path
Integer i = m.get(key);
assume (not (i = null));
Integer ni = i + 1;
Boolean b = m.replace(key, i, ni); // @P
assume (b);
return ni;
```

Figure 7. The two paths of the exceptional variant of `inc`

`put` method call. Therefore, it cannot move to the `putIfAbsent` method call. Although the path is atomic, moverness is too strong to prove the atomicity of it.

We have that `m` is an atomic data structure. Thus, X is atomic for `m`. Therefore, there is a sequential execution S of the program p for m such that $S|m$ is equivalent to $X|m$. We have that $S|m$ is equivalent to $X|m$ and X includes the first path. Thus, $S|m$ includes the two method calls on m from this path. Thus, $S|m$ is of the following form.

```
// m0
Integer i = m.get(key);
// m1
// Interleaving (other method calls on m)
// m2
Integer r = m.putIfAbsent(key, 1); // @P
// m3
```

Let the states `m0` and `m1` denote the pre-state and post-state of the method `m.get(key)`, and `m2` and `m3` denote the pre-state and post-state of the method `m.putIfAbsent(key, 1)` in S. (These states are added as comments to the snippet above.) In this explanation, we use ( ) to number points and use ⟦ ⟧ to reference them. From the assume statements of the first path and the equivalence of $S|m$ and $X|m$, we have (1) $i = null$ (2) $r = null$. From above we have
(3) $(m1, i) = m0.get(key)$,
(4) $(m3, r) = m2.putIfAbsent(key, 1)$.

To prove the atomicity of X for h, we need to show a justifying sequential execution. The three methods `get`, `putIfAbsent` and `replace` of `h` are just delegates to the same methods on `m`. It seems that the execution S of p (that is a sequential execution for `m`) could be used as a justifying sequential execution for atomicity of X for

h as well, but the problem is the composing method inc. The inc method involves two method calls on m that as shown above may not be adjacent in S. Thus S is not sequential for h. In other words, method calls on h from other threads may take effect between get and putIfAbsent of an inc method call.

We construct a sequential execution S' from S as follows: The atomicity point of this path of inc method is putIfAbsent. We remove the two method calls get and putIfAbsent and replace them with a call to inc at the place of putIfAbsent. Thus, the inc method starts from the state m2 (the pre-state of putIfAbsent) with the same input argument key.[1]

We now show the path that results from the sequential execution of inc in S'. We execute the inc method depicted in Figure 2 starting from the state m2 with the input value key. We denote the variables in this sequential execution with a prime. The first method call is

```
// m2
Integer i' = m.get(key);
// m4'
```

From the above method call, we have (5) $(m4', i') = m2.get(key)$. From $[\![\mathcal{A}_0]\!]$ (the axiom in Figure 3) on [5], we have (6) $i' = m2(key)$. From $[\![\mathcal{A}_1]\!]$ on [5], we have (7) $m2 = m4'$. From $[\![\mathcal{A}_3]\!]$ on [4], we have (8) $r = m2(key)$. From [6], [8] and [2], we have (9) $i' = null$. Thus, the if condition is satisfied and the if branch is executed:

```
// m2
Integer i' = m.get(key);
// m4'
Integer r' = m.putIfAbsent(key, 1);
// m3'
```

From above, we have (10) $(m3', r') = m4'.putIfAbsent(key, 1)$. From [4], [10] and [7] and that the methods are deterministic, we have (11) $m3' = m3$ and (12) $r' = r$. From [12] and [2], we have (13) $r' = null$. Thus, the second if condition is satisfied as well and we have

```
// m2
Integer i' = m.get(key);
// m4'
Integer r' = m.putIfAbsent(key, 1);
// m3'
return 1;
```

To prove the atomicity of X for h, we need to show that S' is a sequential execution of the program p and $S'|h$ is equivalent to $X|h$. We now explain why each of these requirements are satisfied. In the next section, we will formalize these requirements as a sufficient condition for atomicity that we call condensability.

We show that S' is a sequential execution of p. We already know that S is a sequential execution of p. To construct S' from S, we condensed the execution of the inc method call. We need to make sure that other method calls execute as before. First, as the shared state is the map m, we need to check that other method calls on m observe the same pre-state as before. We removed the get method call and replaced the putIfAbsent method call with a sequential execution of the inc method. We check the following two conditions:

1. Removing the method get, does not affect the pre-state of the method right after it in S. That is the pre-state and post-state of the removed method get are the same in S. We need to show that m0 = m1 this is given by $[\![\mathcal{A}_1]\!]$ and [3].
2. The state of m after the sequential execution of inc in S' is the same as the post-state of the putIfAbsent method call in S. We have to show that $m3' = m3$ that is already shown at [11].

Second, we need to check that the return value of the new execution of inc in S' is the same as the return value of its previous execution in S. We have already seen that the executions of inc in S' and X both return the same value 1. The executions of inc in X and S returns the same value as they take the same paths. The executions of inc in S' and S returns the same value.

We show the equivalence of $S'|h$ and $X|h$. We already know the equivalence of $S|m$ and $X|m$. All the method of h other than the inc method are delegates to method on m. Thus, we only need to show the equivalence of the argument and return value of the two inc method calls in X and S'. The two arguments are equal by construction and we have already seen that both method calls return the same value 1.

Thus, S' justifies the atomicity of X for h.

To show real-time-preservation of S', consider two method calls N_1 and N_2 on h such that N_1 is ordered before N_2 in X. As the atomicity point of each method call on h is within the method, the atomicity point n_1 of N_1 is ordered before the atomicity point n_2 of N_2 in X. The atomicity points n_1 and n_2 are method calls on m. Thus, as S is a real-time-preserving equivalent execution for $X|m$, n_1 is ordered before n_2 in S. Thus, from the construction of S' from S that serializes N_1 at n_1 and N_2 at n_2, we have that N_1 is ordered before N_2 in S'.

Now, let us consider the second path. Similar to the previous case, consider a atomicity order S including the second path of the inc method.

```
// Second path
// m0
Integer i = m.get(key);
// m1
// Interleaving
Integer ni = i + 1;
// Interleaving
// m2
Boolean b = m.replace(key, i, ni); // @P
// m3
return ni;
```

From the assumptions of the path, we have (1) $\neg(i = null)$ and (2) $b = true$. From above, we have (3) $(m1, i) = m0.get(key)$, (4) $ni = i + 1$, (5) $(m3, b) = m2.replace(key, i, ni)$.

Consider a sequential execution S' where get and replace are removed from S and the inc method executes sequentially in the place of replace.

Let us see the execution of inc starting at state m2 with the input argument key. The first method call is:

```
// m2
Integer i' = m.get(key);
// m4'
```

From the above method call, we have (6) $(m4', i') = m2.get(key)$. From $[\mathcal{A}_0]$ and [6], we have (7) $i' = m2(key)$. From $[\mathcal{A}_1]$ and [6], we have (8) $m2 = m4'$.

From $[\mathcal{A}_5]$ on [5], and [2], we have (9) $i = m2(key)$. From [7] and [9], we have (10) $i' = i$. From [1] and [10], we have (11) $\neg(i' = null)$. Thus, the if condition is not satisfied and the else branch is executed:

[1] If there are other instances of inc method calls in X, they are treated similarly. We focus on one inc method call for illustration purposes.

```
// m2
Integer i' = m.get(key);
// m4'
Integer ni' = i' + 1;
// m4'
Boolean b' = m.replace(key, i', ni');
// m3'
```

From above, we have (12) $ni' = i' + 1$, and (13) $(m3', b') = m4'.replace(key, i', ni')$. From [4], [12] and [10], we have (14) $ni' = ni$. From [5], [13], [8], [10] and [14] and that the methods are deterministic, we have (15) $m3' = m3$ and (16) $b' = b$. From [16] and [2], we have $b' = true$. Thus, the second if condition is satisfied and we have

```
// m2
Integer i' = m.get(key);
// m4'
Integer ni' = i' + 1;
// m4'
Boolean b' = m.replace(key, i', ni');
// m3'
return ni';
```

To prove the atomicity of X on `h`, we need to show that S' is equivalent to $X|h$, and real-time-preserving.

To prove the equivalence of S' and $X|h$, we have to show that the sequential execution of `inc` returns the same value as the interleaved path of `inc`. We have to show that $ni' = ni$ that is already shown at [14].

Legality of S' is implied by the legality of S, the construction of S' from S above and the following two invariants:

1. Removing the method `get`, does not affect the pre-state of the method right after it in S. That is the pre-state and post-state of the removed method `get` are the same in S. We need to show that `m0 = m1` this is given by [$\mathcal{A}_1$] and [3].
2. The post-state of the sequential execution of `inc` in S' is the same as the post-state of the `replace` call in S. We have to show that $m3' = m3$ that is already shown at [15].

Real-time-preservation of S' is implied by the real-time-presentation of S and that the atomicity point of a method call N on h is a method call n on m within the execution interval of N.

Thus, S' is an equivalent, real-time-preserving legal sequential execution for $X|h$. Thus, X is atomic for h.

2. Example Benchmarks

This section presents representative examples of composing methods. We show four cases: Increment, Memoize, Removal and Toggle. For each case, we show the sequential implementation followed by the atomic and non-atomic implementations.

2.1 Increment

Increment method initializes to one or increments the value of the input key.

Increment case appears in applications including ApacheCassandra (SuperColumn class), ApacheCassandra (ColumnFamily class), ApacheDerby (class BufferCache), and ApacheDerby (class ConcurrentCache).

2.2 Memoize

Memoize is a memoizing function.

Memoize case appears in the following applications including AdaptivePlanning (class CC), AdobeBlazeDS (class FIFOMessageQueue (two times)), AdobeBlazeDS (class GossipRouter), AmfSerializer (class DefaultExternalizer), and Annsor (class Annsor).

2.3 Removal

Remove method removes the input key from the map and returns the previous value of the key.

This example can be reduced to a single remove method call call on the map but several implementations of it can be seen in the industrial applications. This case appears in applications including ApacheTomcat (class ApplicationContext).

2.4 Toggle

Toggle method toggles the membership of an element in the set.

Program 1 Increment (Sequential)

```
// Sequential Version
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, ni);
      return ni;
   }
}
```

Program 2 Increment (Atomic)

```
@BaseObject("m")
public Integer inc1(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;
         if (m.replace(key, i, ni))
            return ni;
      }
   }
}
```

```
@BaseObject("m")
public Integer inc2(K key) {
   while (true) {
      Integer i = m.putIfAbsent(key, 1);
      if (i == null)
         return 1;
      else {
         Integer ni = i + 1;
         if (m.replace(key, i, ni))
            return ni;
      }
   }
}
```

```
@BaseObject("m")
public Integer inc3(K key) {
   Integer i = m.putIfAbsent(key, 1);
   if (i == null)
      return 1;
   else {
      while (true) {
         i = m.get(key);
         Integer ni = (i == null) ? 1 : i + 1;
         if (m.replace(key, i, ni))
            return ni;
      }
   }
}
```

```
@BaseObject("m")
@Result("+")
public Integer inc4(K key) {
   Integer i = m.putIfAbsent(key, 1);
   if (i != null) {
      Integer ni = i + 1;
      while (!m.replace(key, i, ni)) {
         i = m.get(key);
         ni = (i == null) ? 1 : i + 1;
      }
      return ni;
   }
   return 1;
}
```

Program 3 Increment (Non-atomic)

```
@BaseObject("m")
public Integer incX1(K key) {
   Integer i = m.get(key);
   if (i == null) {
      Integer r = m.put(key, 1);
   } else {
      Integer ni = i + 1;
      m.replace(key, i, ni);
   }
}
```

Program 4 Memoize (Sequential)

```
V compute(K k) {
   V v = m.get(k);
   if (v == null) {
      v = compute(k);
      m.put(k, v);
   }
   return v;
}
```

Program 5 Memoize (Atomic)

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoize1(K k) {
   V v = m.get(k);
   if (v == null) {
      v = compute(k);
      V v2 = m.putIfAbsent(k, v);
      if (v2 != null)
         v = v2;
   }
   return v;
}
```

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoize2(K k) {
   while (true) {
      V v = m.get(k);
      if (v == null) {
         v = compute(k);
         V r = m.putIfAbsent(k, v);
         if (r != null)
            continue;
      }
      return v;
   }
}
```

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoize3(K k) {
   V v = m.get(k);
   while (true) {
      if (v != null)
         return v;
      else {
         v = compute(k);
         V vp = m.putIfAbsent(k, v);
         if (vp == null)
            return v;
         v = vp;
      }
   }
}
```

Program 6 Memoize (Non-atomic)

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoizeX1(K k) {
   if (!m.containsKey(k)) {
      v = compute(k);
      m.putIfAbsent(k, v);
   }
   v = m.get(k);
   return v;
}
```

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoizeX2(K k) {
   V v = m.get(k);
   if (v == null) {
      v = compute(k);
      m.putIfAbsent(k, v);
      v = m.get(k);
   }
   return v;
}
```

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoizeX3(K k) {
   V v = m.get(k);
   if (v == null) {
      v = compute(k);
      m.putIfAbsent(k, v);
   }
   return v;
}
```

Program 7 Removal (Sequential)

```
Attribute removeAttribute(String name) {
   boolean found = m.containsKey(name);
   Attribute val = null;
   if (found) {
      val = m.get(name);
      m.remove(name);
   }
   return val;
}
```

Program 8 Removal (Atomic)

```
Attribute removeAttribute1(String name) {
   while (true) {
      boolean found = m.containsKey(name);
      Attribute val = null;
      if (found) {
         val = m.remove(name);
         if (val == null)
            continue;
      }
      return val;
   }
}
```

```
Attribute removeAttribute2(String name) {
   Attribute val = m.get(name);
   if (val != null)
      val = m.remove(name);
   return val;
}
```

```
Attribute removeAttribute3(String name) {
   return m.remove(name);
}
```

Program 9 Removal (Non-atomic)

```
@BaseObject("m")
Attribute removeAttribute(String name) {
  Attribute val = null;
  found = m.containsKey(name);
  if (found) {
    val = m.get(name);
    m.remove(name);
  }
  return val;
}
```

Program 10 Toggle (Sequential)

```
public void toggle(V v) {
   if (s.contains(v)) {
      s.remove(v);
   } else {
      s.add(v);
   }
}
```

Program 11 Toggle (Atomic)

```
@BaseObject("s")
public void toggle(V v) {
   while (true) {
      if (s.contains(v)) {
         b = s.remove(v);
         if (b)
            return;
      } else {
         boolean b = s.add(v);
         if (b)
            return;
      }
   }
}
```

3. Condensability Theorem

As the labels are unique in an execution, we define the following functions on labels. The functions obj_X, $name_X$, $thread_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the thread identifier, the first and the second argument, and the return value associated with the labels.

We remind the definitions form the paper body.

DEFINITION 1 (Atomicity). *An execution X of a program p is atomic for an object o if and only if there exists an execution S of p (called the justifying execution of X for o)[2] such that*

- *$S|o$ is sequential i.e. $S|o \in Sequential$,*
- *$S|o$ is equivalent to $X|o$ i.e. $S|o \equiv X|o$, and*
- *$S|o$ is real-time-preserving i.e. $\prec_{X|o} \subseteq \prec_{S|o}$.[3]*

An object o is atomic iff every execution of every program is atomic for o.

DEFINITION 2. *Consider an object c composing an atomic object o. A method m of c is condensable if and only if for every execution X and justifying execution S of X for o, for every execution e of m in S, there is a method call $\mathcal{P}(e)$ on o in e such that*

- *All the method calls on o in e other than $\mathcal{P}(e)$ are accessors.[4] and*
- *Let s be the sequential execution of m with the same arguments as in e with the same pre-state for o as $\mathcal{P}(e)$ in S,*
  - *s results in the same post-state for o as $\mathcal{P}(e)$ in S.*
  - *s results in the same return value as e.*

An object is condensable if and only if all of its methods are condensable.

THEOREM 1 (Condensability). *Every condensable composing object is atomic.*

Proof:
HYPOTHESIS:
(1) The object o is atomic.
(2) The object c is composing o.
(3) The object c is condensable.
DESIRED CONCLUSION:
The object c is atomic.

We assume that
(4) p is a program.
(5) X is an execution of p.
(As an example of X, consider Figure 8(a). The method calls u_1, u_2 and u_3 on c and their paths e_1, e_2 and e_3 are depicted in Figure 8(b).)
By definition 1, we need to prove that
X is atomic for c.

By Definition 1 on [1], we have
There exists execution S such that
(6) S is a justifying execution X for o.
(7) S is an execution of p.
(8) $S|o \in Sequential$
(9) $S|o \equiv X|o$
(10) $\prec_{X|o} \subseteq \prec_{S|o}$
(As an example of $S|o$, consider Figure 8(c).)

By definition 2 on [3], we have
(11) Every method of c is condensable.
From Definition 2 on [11] and [6], we have
(12) For every method call u on c with execution e in S, there exits a method call $\mathcal{P}(e)$ such that
(13) $\mathcal{A}(e) \cup \{\mathcal{P}(e)\}$ is the set of method calls on o in e.
(14) $\mathcal{A}(e)$ are accessor method calls.
(15) $s(u)$ is the sequential execution of u with the same arguments as in e starting from the pre-state of $\mathcal{P}(e)$ in S.
(16) The post-state of $s(u)$ for o is equal to the post-state of $\mathcal{P}(e)$ for o in S.
(17) The value that $s(u)$ returns is equal to the value that e returns.

(18) We construct the execution S' from $S|o$ as follows:
For every method call u on c with execution e in S:
Remove $M(e)$ and
Replace $\mathcal{P}(e)$ with $s(u)$.
(As an example of S', consider Figure 8(d).)

We first show that
(19) S' is an execution of p.
From [18], we have
(20) S' calls the same methods on c as S.
(21) S' is a sequence of $s(u)$'s.
We show that
(22) If $s(u')$ is the method call that immediately precedes the method call $s(u)$ in S', then the state of o that $s(u)$ starts from is equal to the post-state of $s(u')$ for o in S'.
We assume that
(23) $s(u)$ is a method call in S'
(24) $s(u')$ is the method call that immediately precedes $s(u)$ in S'
We show that
The state of o that $s(u)$ starts from is equal to the post-state of $s(u')$ for o in S'.
From [16], we have
(25) The post-state of $s(u')$ in S' is equal to the post-state of $\mathcal{P}(e')$ in S.
By the construction [18] on [24] and [8], we have
(26) There is no e'' such that $\mathcal{P}(e'')$ is between $\mathcal{P}(e')$ and $\mathcal{P}(e)$ in S.
From [26], [13] and [14], we have
(27) The post-state of $\mathcal{P}(e')$ in S is equal to the pre-state of $\mathcal{P}(e)$ in S.
From [15], we have
(28) $s(u)$ starts from a state of o that is equal to the pre-state of $\mathcal{P}(e)$ in S.
From [28], [27] and [25], we have
The state of o that $s(u)$ starts from is equal to the post-state of the $s(u')$.
We show that
(29) The return value of u in S' is equal to the return value of u in S.
From [9] and [10], we have
(30) The method calls on o by the method call u in X are in S as well and have the same order as in X.
Therefore,
(31) The method call u is executed in S and

[2] Note that compared to atomicity that requires $S|o$ to be in the sequential specification of o, atomicity requires $S|o$ to be a sequential execution on the object o itself.

[3] Note the real-time-preservation is usually implicitly assumed as a condition for atomicity.

[4] More precisely, these method calls can be mutators as far as they do not affect the return value of later method calls.

T_1	T_2
$inv(l_1 \triangleright c.inc(key))$	
$\quad inv(l_{11} \triangleright o.get(key))$	
$\quad ret(l_{11} \triangleright null)$	
	$inv(l_2 \triangleright c.get(key))$
	$\quad inv(l_{21} \triangleright o.get(key))$
	$\quad ret(l_{21} \triangleright null)$
	$ret(l_2 \triangleright null)$
$\quad inv(l_{12} \triangleright o.putIfAbsent(key, 1))$	
	$inv(l_3 \triangleright c.get(key))$
	$\quad inv(l_{31} \triangleright o.get(key))$
	$\quad ret(l_{31} \triangleright 1)$
	$ret(l_3 \triangleright 1)$
$\quad ret(l_{12} \triangleright null)$	
$ret(l_1 \triangleright 1)$	

(a) Execution History X

$u_1 =$ $l_1 \triangleright c.inc(key)$	$u_2 =$ $l_2 \triangleright c.get(key)$	$u_3 =$ $l_3 \triangleright c.get(key)$
$e_1 =$	$e_2 =$	$e_3 =$
$inv(l_1 \triangleright c.inc(key))$	$inv(l_2 \triangleright c.get(key))$	$inv(l_3 \triangleright c.get(key))$
$\quad inv(l_{11} \triangleright o.get(key))$	$\quad inv(l_{21} \triangleright o.get(key))$	$\quad inv(l_{31} \triangleright o.get(key))$
$\quad ret(l_{11} \triangleright null)$	$\quad ret(l_{21} \triangleright null)$	$\quad ret(l_{31} \triangleright 1)$
$\quad inv(l_{12} \triangleright o.putIfAbsent(key, 1))$	$ret(l_2 \triangleright null)$	$ret(l_3 \triangleright 1)$
$\quad ret(l_{12} \triangleright null)$		
$ret(l_1 \triangleright 1)$		

(b) Method Calls u and Their executions e

$inv(l_{11} \triangleright o.get(key))$
$ret(l_{11} \triangleright null)$
$inv(l_{21} \triangleright o.get(key))$
$ret(l_{21} \triangleright null)$
$inv(l_{12} \triangleright o.putIfAbsent(key, 1))$
$ret(l_{12} \triangleright null)$
$inv(l_{31} \triangleright o.get(key))$
$ret(l_{31} \triangleright 1)$

(c) The order $S|o$

$inv(l'_2 \triangleright c.get(key))$
$\quad inv(l'_{21} \triangleright o.get(key))$
$\quad ret(l'_{21} \triangleright null)$
$ret(l'_2 \triangleright null)$
$inv(l'_1 \triangleright c.inc(key))$
$\quad inv(l'_{11} \triangleright o.get(key))$
$\quad ret(l'_{11} \triangleright null)$
$\quad inv(l'_{12} \triangleright o.putIfAbsent(key, 1))$
$\quad ret(l'_{12} \triangleright null)$
$ret(l'_1 \triangleright 1)$
$inv(l'_3 \triangleright c.get(key))$
$\quad inv(l'_{31} \triangleright o.get(key))$
$\quad ret(l'_{31} \triangleright 1)$
$ret(l'_3 \triangleright 1)$

(d) Justifying order S' for c

Figure 8. Atomicity Order Construction. $\mathcal{P}(e_1) = l_{12}$, $\mathcal{P}(e_2) = l_{21}$ and $\mathcal{P}(e_3) = l_{31}$

and takes the same path as X.

Therefore,

(32) The return value if u in S is equal to the return value of u in X.

From [17], ([18], [15]) and ([12]) we have

(33) The return value of u in S' is equal to the return value of u in X.

From [32] and [33], we have

The return value of u in S' is equal to the return value of u in S.

From [7], [20], [21], [22], and [29], we have

S' is an execution of p.

We show that

(34) $S'|c \in Sequential$

By the construction [18] and [8], we have that

(35) Method calls $s(u)$ on c in S' are non-interleaved.

Thus,

$S'|c \in Sequential$

We show that

(36) $S'|c \equiv X|c$

By the construction [18], we have

(37) For all u:

$u \in X$ and $obj_X(u) = c$ and e is the execution of u in X and $\mathcal{P}(e) \in S$

$\Leftrightarrow$

$u \in S'$ and $obj_{S'}(u) = c$

From [13], we have

(38) For all u:

$u \in X$ X is atomic for c.
and $obj_X(u) = c$
and e is the execution of u in X
$\Rightarrow$
$\mathcal{P}(e) \in X|o$.

From [38] and [9], we have

(39) For all u:
$u \in X$
and $obj_X(u) = c$
and e is the execution of u in X
$\Rightarrow$
$\mathcal{P}(e) \in S$.

From [37] and [39], we have

(40) For all u:
$u \in X$
and $obj_X(u) = c$
$\Leftrightarrow$
$u \in S'$
and $obj_{S'}(u) = c$

From [17], ([18], [15]) and ([12]) we have

(41) For all u:
$u \in X$
and $obj_X(u) = c$
$\Rightarrow$
$arg_{S'}(u) = arg_X(u)$
$ret_{S'}(u) = ret_X(u)$

From [40] and [41], we have

$S'|c \equiv X|c$

We now show that

(42) $\prec_{X|c} \subseteq \prec_{S'|c}$.

We assume that

(43) $u \prec_{X|c} u'$

We show that

$u \prec_{S'|c} u'$

From [43], we have

(44) $u \prec_X u'$
(45) $u \in X$
(46) $obj_X(u) = c$
(47) $u' \in X$
(48) $obj_X(u') = c$

Let

(49) e is the execution of u in X
(50) e' is the execution of u' in X

From [44], [49], and [50], we have

(51) $e \Subset_X e'$

From [51] and [13], we have

(52) $\mathcal{P}(e) \prec_X \mathcal{P}(e')$

From [12] and [13] on [45], [46] and [49], we have

(53) $\mathcal{P}(e) \in X|o$

From [12] and [13] on [47], [48] and [50], we have

(54) $\mathcal{P}(e') \in X|o$

From [52], [53] and [54], we have

(55) $\mathcal{P}(e) \prec_{X|o} \mathcal{P}(e')$

From [55], and [10], we have

(56) $\mathcal{P}(e) \prec_{S|o} \mathcal{P}(e')$

From [18], on [56], [49] and [50], we have

(57) $u \prec_{S'} u'$

From [36] on [46] and [48], we have

(58) $obj_{S'}(u) = c$
(59) $obj_{S'}(u') = c$

From [57], on [58], [59], we have

$u \prec_{S'|c} u'$

By Definition 1 on [5], [19], [34], [36], and [42], we have that

```
@BaseObject("m")
@ArgFunctional(object="this", method="compute")
V memoize(K k) {
    V v = m.get(k);
    if (v == null) {
        v = this.compute(k);
        V v2 = m.putIfAbsent(k, v);
        if (v2 != null)
            v = v2;
    }
    return v;
}
```

Figure 9. The composing method `memoize`

$$(m_2, b) = m_1.containsKey(k)$$
$$\Rightarrow$$
$$m_1 = m_2 \ \wedge$$
$$(m_1(k) \neq null) \Leftrightarrow b$$

$$(m_2, v) = m_1.remove(k)$$
$$\Rightarrow$$
$$v = m_1(k) \ \wedge$$
$$((m_1(k) \neq null) \ \wedge \ (m_2 = m_1[k \mapsto null])) \ \vee$$
$$((m_1(k) = null) \ \wedge \ (m_2 = m_1))$$

Figure 10. Axioms for `containsKey` and `remove` methods

4. Checking Condensability

Snowflake computes the paths of the input composing method, checks the purity of its loops and checks the condensability of each of its paths. In this section, we describe these steps in turn. First, we look at the user input.

4.1 User Input

To verify the atomicity of a composing method, the user provides the composing method and axioms that characterize the methods of the base object. Our tool has embedded axioms for Java concurrent map and set data structures. An annotation identifies the base object in the composing method.

4.1.1 Annotations

Consider Figure 9 that is a composing method that implements a concurrent memoizing function. The user identifies the base object with the `BaseObject` annotation. The base object in the `memoize` method is `m`. Snowflake supports optional annotations to declare *functional* method calls and *static* receivers. A functional method call is a method call whose return value is a function of the state of its receiving object and arguments. The user can declare functional method calls with the `Functional` annotation. An argument-functional method call is a method call whose return value is a function of the state of its arguments. The user can declare argument-functional method calls with the `ArgFunctional` annotation. For example, the method call `this.compute` in the `memoize` method is an argument-functional method call. Its return value is a function of the input key. Furthermore, the user can declare static receivers. The state of a static receiver remains unchanged. These optional annotations help Snowflake recognize functional computations and leverage this knowledge to deduce equalities.

4.1.2 Axioms

Snowflake benefits from the properties of conditional atomic methods of the basic object. We already presented the axioms of `get`, `put`, `putIfAbsent` and `replace` methods of the atomic map. Figure 10 states the the axioms for `containsKey` and `remove` methods of the atomic map. The axiom of `containsKey` states that it is an accessor method and returns whether the map contains the key. The axiom of `remove` states that it removes the input key from the map if the map contains the key and returns the previous value of the key.

$$(s_2, b) = s_1.contains(e)$$
$$\Rightarrow$$
$$s_1 = s_2 \ \wedge$$
$$s_1(e) \Leftrightarrow b$$

$$(s_2, b) = s_1.add(e)$$
$$\Rightarrow$$
$$(\neg s_1(e) \ \wedge \ (s_2 = s_1[e \mapsto true]) \ \wedge \ b) \ \vee$$
$$(s_1(e) \ \wedge \ (s_2 = s_1) \ \wedge \ \neg b)$$

$$(s_2, b) = s_1.remove(e)$$
$$\Rightarrow$$
$$(s_1(e) \ \wedge \ (s_2 = s_1[e \mapsto false]) \ \wedge \ b) \ \vee$$
$$(\neg s_1(e) \ \wedge \ (s_2 = s_1) \ \wedge \ \neg b)$$

Figure 11. Axioms for `contains`, `add` and `remove` methods

Figure 11 states the axioms for the `contains`, `add` and `remove` methods of the atomic set. The abstract set state is a function that maps every element in the set to $true$ and every element not in the set to $false$. The axiom of `contains` states that it is an accessor method and returns whether the set contains the element. The axiom of `add` states that it adds the element to the set if the set does not contain the element and returns whether adding was successful. The axiom of `remove` states that it removes the element from the set if the set contains the element and returns whether removing was successful.

Note that in the Java concurrent package, classes `ConcurrentHashMap` and `ConcurrentSkipListMap` that implement `ConcurrentMap` interface, support the atomic operations:

```
V putIfAbsent(K key, V value)
    // If the specified key is not already
    // associated with a value, associate
    // it with the given value.
boolean remove(Object key, Object value)
    // Removes the entry for a key only
    // if currently mapped to a given value.
    // It returns true if the value was removed.
public V remove(Object key)
    // Removes the key (and its corresponding
    // value) from this table.
    // It returns the value to which the key
    // had been mapped in this table, or
    // null if the key did not have a mapping.
V replace(K key, V value)
    // Replaces the entry for a key only
    // if currently mapped to some value.
boolean replace(K key, V oldValue, V newValue)
    // Replaces the entry for a key only
    // if currently mapped to a given value.
```

The class `CopyOnWriteArrayList` supports the following atomic operations:

```
boolean addIfAbsent(E e)
    // Append the element if not present.
boolean remove(Object o)
    // Removes the first occurrence of the
    // specified element from this list,
    // if it is present.
```

$$
\begin{array}{rcl}
s & ::= & s;\ s \\
& | & \textbf{if}\ b\ s_1\ \textbf{else}\ s_2 \\
& | & \textbf{while}\ b\ s \\
& | & y = o.n(x) \\
& | & \textbf{break} \\
& | & \textbf{continue} \\
& | & \textbf{return}\ x
\end{array}
$$

$$EPaths(y = o.n(x)) = \{y = o.n(x)\}$$
$$EPaths(\textbf{break}) = \{\textbf{break}\}$$
$$EPaths(\textbf{continue}) = \{\textbf{continue}\}$$
$$EPaths(\textbf{return}\ x) = \{\textbf{return}\ x\}$$
$$EPaths(s_1;\ s_2) = EPaths_c(s_1) \cup EPaths_b(s_1) \cup EPaths_r(s_1) \cup EPaths_n(s_1) \cdot EPaths(s_2)$$
$$EPaths(\textbf{if}\ b\ s_1\ \textbf{else}\ s_2) = assume(b) \cdot EPaths(s_1) \cup assume(\neg b) \cdot EPaths(s_2)$$
$$EPaths(\textbf{while}\ b\ s) = assume(\neg b) \cup assume(b) \cdot EPaths_n(s) \cdot assume(\neg b) \cup assume(b) \cdot EPaths_b(s) \cup assume(b) \cdot EPaths_r(s)$$
$$NPaths(\textbf{while}\ b\ s) = assume(b) \cdot EPaths_n(s) \cup assume(b) \cdot EPaths_c(s)$$

Figure 12. Paths

The set classes `ConcurrentSkipListSet` and `CopyOnWriteArraySet` support the following atomic operations:

```
boolean add(E e)
   // Adds the specified element to this set
   // if it is not already present.
boolean remove(Object o)
   // Removes the specified element from this
   // set if it is present.
```

4.2 Paths

As the first step, Snowflake computes the set of paths of the composing method. We adopt the terminology of paths from [5] and [16]. A path of a loop is *exceptional*, if it is executed as the last iteration of the loop. An exceptional loop path ends in a break or return statement or the condition of the loop evaluates to false at the end of the path. A path of a loop is *normal* if it is not exceptional. Figure 12 depicts a simple language and the set of paths of its programs. The operator $\cdot$ is concatenation of a pair of sequences and is also lifted to a pair of sets of sequences. The language includes method call, sequence, conditional, while loop, break, continue and return statements. $EPaths(s)$ denotes the set of exceptional paths and $NPaths(s)$ denotes the set of normal paths of s. $EPaths_b(s)$ denotes the subset of $EPaths(s)$ that end in **break**, $EPaths_c(s)$ denotes the subset of $EPaths(s)$ that end in **continue**, $EPaths_r(s)$ denotes the subset of $EPaths(s)$ that end in a **return** statement, and $EPaths_r(s)$ denotes the subset of $EPaths(s)$ that does not end in a **break**, **continue**, or **return** statement. Note that $EPaths_b(s)$, $EPaths_c(s)$, $EPaths_r(s)$, and $EPaths_n(s)$ are a disjoint partition of $EPaths(s)$.

After the paths are computed, **break** and **continue** statements at the end of the paths are ignored. Let us denote a path with the triple (b, M, r) where b is the conjunction of the assumed conditions of the path, $M = \overline{m}$ is the sequence of method calls $y = o.n(x)$ of the path and r is the returned variable of the path.

```
@BaseObject("m")
public Integer inc(K key) {
   Integer i = m.putIfAbsent(key, 1);
   if (i != null) {
      Integer ni = i + 1;
      while (!m.replace(key, i, ni)) {
         i = m.get(key);
         ni = (i == null) ? 1 : i + 1;
      }
      return ni;
   }
   return 1;
}
```

Figure 13. An alternative implementation of `inc`

4.3 Purity

Informally, in a pure normal path, no shared variable is mutated and if a local variable is written, it is overwritten outside the path before it is read. For example, the while loop of the `inc` method presented in the example above is pure. Figure 13 shows another atomic implementation of the `inc` method. The while loop in this implementation is impure as the variable `ni` carries over to the next iteration of the loop.

5. Future Work

The atomic methods can be synthesized from the sequential versions. The pattern is a loop. Inside the loop copy the sequential version. Whenever a write has to happen, use a method (like putIfAbsent) that atomically checks the validity of the conditions that are checked since the beginning of this iteration of the loop. If the method returns failure of the validation, continue from the beginning of the loop. Otherwise continue (or return).

6. Related Works

6.1 Testing and Automatic Verification of Atomicity

Flanagan and Qadeer [6, 7] presented a type system that can check the atomicity of data-race-free lock-based programs. The type system is based on Lipon's [10] theory of right and left movers. An action is a right mover if whenever an action from another thread immediately follows it in an execution, the two instructions can be swapped and still result in the same final state and return values. The notion of a left mover is defined similarly. An action is both mover if it is both right or left mover. Flanagan and Qadeer consider data-race-free programs and hence regard a lock method call as a right mover, an unlock method call as a left mover and a variable access as a both mover. Their type system checks that every atomic statement consists of zero or more both or right movers, followed by at most one arbitrary operation, followed by zero or more both or left movers. The type system can check the atomicity of lock-based but not non-blocking synchronization.

Wang and Stoller [16] extended the moverness approach to support non-blocking synchronization by paired load-link (LL) and store-conditional (SC) instructions. An SC instruction succeeds and updates the accessed location only if the value of the location is unchanged since its matching LL instruction. Therefore, successful pairs of LL and SC on the same location do not interleave. Thus, similar to lock and unlock method calls, LL and SC are right and left movers respectively. They presented a static analysis technique to verify atomicity in a small language that supports LL an SC primitives. They also extended the notion of purity [5] of statements to loops. They did not implement an automatic tool but manually proved the atomicity of three non-blocking algorithms including a non-blocking queue algorithm.

In contrast to paired synchronization operations, single operations such as CAS are used more often in the implementation of concurrent data structures. Unfortunately, the ABA problem [13] prevents direct application of moverness to non-blocking algorithms that employ single synchronization operations. Consider a thread that reads the value A of a location and later executes a CAS on the location with the value A as the expected value. The CAS operation may succeed even though the value of the location may have changed to B and then back to A by other threads since the read operation. The read operation that reads the value A can not move past of the write operation that writes value B. Thus, the initial read and the CAS are not necessarily left and right movers. Wang and Stoller suggested rewriting algorithms to store a counter value together with the value of the location to simulate the pairing. As we illustrated by an example, condensability can prove atomicity of methods that employ single synchronization operations. In addition, this paper presents an automatic verification tool.

Elmas et al. [3, 4] presented a proof technique and an interactive proof assistant called QED for verification of properties of concurrent programs. The key idea of QED is to simplify the program by rewriting it to consist of larger atomic statements. The user iteratively issues commands that transform the atomic statements of the program either by reduction or abstraction. Reduction uses moverness properties to replace a sequence of atomic statements by a single compound atomic statement. Abstraction replaces an atomic statement with a more relaxed version of it such that the new behavior of the atomic statement is a superset of its old behavior. Finally, the assertions of the resulting program can be verified either using traditional sequential or simple concurrent program logics. Reduction and abstraction preserve or expand the behavior of the original program. Therefore, the assertions proved for the transformed programs are valid in the original program as well. QED has successfully verified non-blocking deque and stack algorithms. In comparison to Snowflake that is applicable to composing methods and is automatic, QED is applicable to more general problems but needs user interaction.

Vafeiadis [15] presented an automatic verification tool called CAVE to prove the linearizability of concurrent data structures. It checks the atomicity and functional correctness of the input implementation with respect to the input specification. It instruments the methods of the implementation with copies of the methods of the specification at the linearization points. It inserts assertions that check the equality of the return values of the implementation and the specification. It uses abstract interpretation to validate that assertions are not violated in the most general client of the data structure. The tool can particularly check the atomicity of pure methods whose linearization points are in the other concurrent methods. The tool has successfully verified several concurrent stack and queue implementations. CAVE verifies the linearizability of the methods of the data structure all together and cannot provide concrete counterexample traces when it fails. On the hand, Snowflake modularly verifies a method at a time and if it cannot verify a method, it reports the failing path. In addition, CAVE relies on abstract interpretation and thus its applicability is dependent to the efficiency of its abstract domains. This is while Snowflake relies on optimized off-the-shelf SMT solvers and is applicable when the axiomatization of the methods of the base data structure is available.

Burckhardt et al. [2] presented an automatic bug-finding tool called Line-up for atomicity of deterministic concurrent data structures. Line-up first generates a set of test programs. A test program is a set of concurrent threads each calling a sequence of methods on the data structure. For each test program, it first schedules and records the deterministic results of all the possible sequential executions of the test program. Then, it uses CHESS [12] to enumerate and schedule concurrent executions of the test program. It reports a concurrent execution as a bug, if there is no equivalent execution in the set of recorded sequential executions. Line-up has found atomicity bugs in the concurrent data structures of the .Net framework. Line-up is a testing tool while Snowflake is verification tool.

6.2 Testing and Automatic Verification of Atomicity for Composing Operations

Shacham et al. [14] presented a tool called Colt to test the atomicity of composing methods that extend the interface of a data structure. Colt searches for non-atomic executions of the composing method by testing it against an open environment. It benefits from the non-commutativity specification of the methods of the data structure to reduce the number of executions that it explores. It guides the execution by scheduling method calls in the environment that are non-commutative to the method calls in the composing method. Colt identified bugs in a suite of real life applications. Colt and Snowflake share the goal of assisting programmers to check the atomicity of composing methods. On the other hand, Colt is a testing tool that find non-atomicity bugs while Snowflake is a verification tool that proves atomicity.

Liu et al. [11] presented a bug finding tool called ICFinder to automatically find incorrect composing blocks of code in large industrial applications. They applied heuristics such as interdependence of fields and methods of libraries to find incorrect compositions. Then, they apply existing static and dynamic techniques to increase the accuracy of their bug finding tool. They reported finding bug in industrial applications.

6.3 Automatic Implementation of Composing Operations

In contrast to Snowflake and Colt that works with unchanged libraries, the following approaches wrap existing libraries with additional mechanisms to automatically provide atomicity for composing methods and thus, incur inevitable overhead.

Herlihy and Koskinen [9] presented a technique that extends linearizable data structures to transactional data structures. They regard atomic composing operations as transactions calling methods on transactional data structures. The idea is that they benefit from the moverness specification of the data structure to allow concurrent execution of commutative method calls. They approximate the moverness specification to a set of abstract locks such that non-commutating method calls acquire the same lock. To prevent deadlocks, acquiring a lock can time out. Thus, a transaction can abort and its previous method calls should be rolled back. Thus, each successful method call, adds its reverse method call to a transaction undo log. An aborting transaction executes the undo log before releasing the locks and retrying. They show that their approach can deliver higher performance over data structures that are implemented using common STM system.

Bronson et al. [1] presented a technique called transactional predication that builds transactional sets and maps from linearizable sets and maps. The key idea is to represent the membership of an element in a set with an STM-managed location called predicate that stores a boolean value. The transactional set uses a linearizable map that maps elements to their predicate. An element is a member of the transactional set if and only if the underlying linearizable map maps the element to true. Methods of the transactional set access the linearizable map directly but access the predicates using STM. The tentative updates of a transaction (involving method calls on the transactional set) to the predicates take effect only if the transaction succeeds to commit. This technique combines high-performance of lock-based and non-blocking data structures and the composability of STM. They report that predicated collections are faster than collections implemented using STM.

Golan-Gueta et al. [8] presented a technique that extends linearizable data structures to support atomic composing operations of clients. The key idea is that the client declares an over-approximation of the set of methods that it will call in the future (hence called foresight information) and the library benefit from it to ensure atomicity. The library maintains a partial order for the composing operations. The execution of a method m by a thread t is safe if for every method m' that may be called in the future by another thread t', either m is a right mover for m' or t is before t' in the partial order or ordering t before t' does not incur a cycle in the partial order. If the execution of a method is not safe, the library blocks the call until its execution is safe. Thus, an equivalent sequential order can be constructed from the maintained partial order. They implement an extension of a Java map data structure by mapping sets of methods calls to the nodes of a lock-tree. Every foresight call from a client acquires the lock that is associated with the set of methods that it declares. They showed the scalability of their approach on several applications.

References

[1] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 6–15, New York, NY, USA, 2010. ACM.

[2] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '10, pages 330–340, New York, NY, USA, 2010. ACM.

[3] Tayfun Elmas, Shaz Qadeer, Ali Sezgin, Omer Subasi, and Serdar Tasiran. Simplifying linearizability proofs with reduction and abstraction. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 296–311, Berlin, Heidelberg, 2010. Springer-Verlag.

[4] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 2–15, New York, NY, USA, 2009. ACM.

[5] Cormac Flanagan, Stephen N. Freund, and Shaz Qadeer. Exploiting purity for atomicity. *IEEE Trans. Software Eng.*, 31(4):275–291, 2005.

[6] Cormac Flanagan and Shaz Qadeer. A type and effect system for atomicity. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, PLDI '03, pages 338–349, New York, NY, USA, 2003. ACM.

[7] Cormac Flanagan and Shaz Qadeer. Types for atomicity. In *Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, TLDI '03, pages 1–12, New York, NY, USA, 2003. ACM.

[8] Guy Golan-Gueta, G. Ramalingam, Mooly Sagiv, and Eran Yahav. Concurrent libraries with foresight. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 263–274, New York, NY, USA, 2013. ACM.

[9] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 207–216, New York, NY, USA, 2008. ACM.

[10] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, December 1975.

[11] Peng Liu, Julian Dolby, and Charles Zhang. Finding incorrect compositions of atomicity. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 158–168, New York, NY, USA, 2013. ACM.

[12] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[13] International Business Machines Corporation. Product Publications. *IBM System/370 Extended Architecture: Principles of Operation*. IBM Corporation, 1983.

[14] Ohad Shacham, Nathan Bronson, Alex Aiken, Mooly Sagiv, Martin Vechev, and Eran Yahav. Testing atomicity of composed concurrent operations. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, pages 51–64, New York, NY, USA, 2011. ACM.

[15] Viktor Vafeiadis. Automatically proving linearizability. In *Proceedings of the 22nd international conference on Computer Aided Verification*, CAV'10, pages 450–464, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 61–71, New York, NY, USA, 2005. ACM.