Testing and Verifying Atomicity of Composed Concurrent Operations

Ohad Shacham
Nathan Bronson
Alex Aiken
Mooly Sagiv
Martin Vechev
Eran Yahav

Tel Aviv University
Stanford University
Stanford University
Tel Aviv University
ETH
Technion
Specifying and Verifying Software Composition

• Efficient libraries are widely available
• Composing software in a way which guarantee correctness:
  – Specification
  – Verification
  – Synthesis
  – Performance
Concurrent Data Structures

• Writing highly concurrent data structures is complicated

• Modern programming languages provide efficient concurrent collections with atomic operations
attr = new ConcurrentHashMap();
...

Attribute removeAttribute(String name){
    Attribute val = null;
    synchronized(attr) {
        found = attr.containsKey(name);
        if (found) {
            val = attr.get(name);
            attr.remove(name);
        }
    }
    return val;
}

Invariant: removeAttribute(name) returns the removed value or null if it does not exist
Invariants: removeAttribute(name) returns the removed value or null if it does not exist
Challenge

Testing and Verifying the atomicity of composed operations
Challenges in Testing

• Specifying software correctness
• Bugs occur in rarely executed traces
  – Especially true in concurrent systems
• Scalability of dynamic checking
  – large traces
• Hard to find programs to test
Challenges in Verification

• Specifying software correctness
• Many sources of unboundedness
  – Data
    • Integers
    • Stack
    • Heap
    • ...
  – Interleavings
• Scalability of static checking
  – Large programs
• Hard to find programs to verify
Testing atomicity of composed operations

OOPSLA’11
Challenge 1: Long traces

- Assume that composed operations are written inside encapsulated methods
- Modular testing
  - Unit testing in all contexts
  - Composed operations need to be correct in all contexts
- May lead to false warnings
if (m.contains(k))
    return m.get(k);
else
    return k;

• False warning in clients without remove
• Sometimes indicate “future bugs”
Challenge 2: Specification

• Check that composed operations are **Linearizable** [Herlihy & Wing, TOPLAS’90]
  – Returns the **same** result as **some** sequential run
removeAttribute("A") { 
    Attribute val = null;
    found = attr.containsKey("A"); 
    if (found) {
        val = attr.get("A");
        attr.put("A", o);
    }
    attr.remove("A");
    return val;
}

attr.put("A", o);
attr.remove("A");
removeAttribute("A") { 
    Attribute val = null;
    found = attr.containsKey("A"); 
    if (found) {
        val = attr.get("A");
        attr.put("A", o);
        attr.remove("A");
    }
    attr.remove("A");
    return val;
}

attr.put("A", o);
attr.remove("A");
But Linearizability errors only occur in rarely executed paths

```java
removeAttribute("A") {
    Attribute val = null;

    found = attr.containsKey("A") ;
    if (found) {
        val = attr.get("A");

        attr.put("A", o);
        attr.remove("A");

        attr.remove("A");
    }

    return val;
}
```
Linearizability errors only occur in rarely executed paths

• Only consider “atomic” executions of base collection operations [TACAS’10, Ball et. al.]

• Employ commutativity/influence of base collection operations
  – Operations on different key commute
  – Partial order reduction using the collection interface
## Influence table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Condition</th>
<th>Potential Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>get(k)</td>
<td>get(k) == null</td>
<td>put(k,*)</td>
</tr>
<tr>
<td>get(k)</td>
<td>get(k) != null</td>
<td>remove(k)</td>
</tr>
<tr>
<td>containsKey(k)</td>
<td>get(k) == null</td>
<td>put(k,*)</td>
</tr>
<tr>
<td>containsKey(k)</td>
<td>get(k) != null</td>
<td>remove(k)</td>
</tr>
<tr>
<td>remove(k)</td>
<td>get(k) == null</td>
<td>put(k,*)</td>
</tr>
<tr>
<td>remove(k)</td>
<td>get(k) != null</td>
<td>remove(k)</td>
</tr>
</tbody>
</table>
COLT Tester

- program
- library spec
- CO extractor
- candidate COs
- key/value driver
- influence driver
- instrument linearizability checking
- Execution
- Timeout
- Non-Lin
Attribute removeAttribute(String name){
    Attribute val = null;
    found = attr.containsKey(name) ;
    if (found) {
        val = attr.get(name);
        attr.remove(name);
    }
    return val;
}

removeAttribute("A") {
    Attribute val = null;
    found = attr.containsKey("A") ;
    if (found) {
        val = attr.get("A");
        attr.remove("A");
    }
    return val;
}
removeAttribute("A") {
    Attribute val = null;
}

found = attr.containsKey("A");
    if (found) {
        val = attr.get("A");
    }

attr.remove("A");
}

return val;

attr.put("A", o); null
attr.remove("A"); o
removeAttribute("A") {
    Attribute val = null;
    found = attr.containsKey("A");
    if (found) {
        return val;
    }
    attr.put("A", o);
    attr.remove("A");
}

attr.put("A", o); null
attr.remove("A"); o

removeAttribute("A") {
    Attribute val = null;
    found = attr.containsKey("A");
    if (found) {
        return val;
    }
    attr.put("A", o);
    attr.remove("A");
}

attr.put("A", o); null
removeAttribute("A") {
    Attribute val = null;
    found = attr.containsKey("A");
    if (found) {
        return val;
    }
    attr.put("A", o);
    attr.remove("A");
}

attr.put("A", o); null
attr.remove("A"); o

Evaluation

• Use Google code search and Koders to search for collection operations methods with at least two operations
• Used simple static analysis to extract composed operations
  – 29% needed manual modification
• Check Linearizability of all public domain composed operations
• Extracted 112 composed operations from 55 applications
  – Apache Tomcat, Cassandra, MyFaces – Trinidad, …
• Each run took less than a second
• Without influence timeout always occur
53 Unknown
42 Non Linearizable
17 Open Non Linearizable
31 Linearizable
81 Non-Linearizable
Results

• Reported the bugs with fixes
• Even bugs in open environment
• As a result of the paper the Java library is being changed

“A preliminary version is in the pre-java8 "jsr166e" package as ConcurrentHashMapV8. We can't release the actual version yet because it relies on Java8 lambda (closure) syntax support. See links from http://gee.cs.oswego.edu/dl/concurrency-interest/index.html including:
http://gee.cs.oswego.edu/dl/jsr166/dist/jsr166edocs/jsr166e/ConcurrentHashMapV8.html

Good luck continuing to find errors and misuses that can help us create better concurrency components!”
Verifying atomicity of composed operations
Motivation

• Unbounded number of potential composed operations
  – There exists no “thick” interface

• Automatically prove Linearizability for composed operations beyond the ones provided
  – Already supports the existing interface
  – No higher order functions

• Zero false alarms
  – beyond modularity
Data independent [Wolper, POPL’86]

```java
Attribute removeAttribute(String name){
    Attribute val = null;
    found = attr.containsKey(name) ;
    if (found) {
        val = attr.get(name);
        attr.remove(name);
    }
    return val;
}
```
Verifying data independent operations using Linearization points in the code

Data independent

Verified using single input

Influence

CO adds one value

Single Mutation

Map elements are bounded
Verifying data independent operations

- Small model reduction
- Decidable when the local state is bounded
- Explore all possible executions using:
  - One input key and finite number of values
  - Influenced based environment uses single value
- Employ SPIN
31 Linearizable
81 Non-Linearizable
Summary

- Writing concurrent data structures is hard
- Employing atomic library operations is error prone
- Modular linearizability checking
- Leverage influence
- Leverage data independence

- Sweet spot
  - Identify important bugs together with a traces showing and explaining the violations
  - Hard to find
  - Prove the linearizability of several composed operations
  - Simple and efficient technique