Automatically Identifying Special and Common Unit Tests for Object-Oriented Programs

Tao Xie
North Carolina State University
Raleigh, NC

David Notkin
University of Washington
Seattle, WA
Automated Testing in the Absence of Specs

- Specifications help improve automated testing but they often don’t exist in practice
  - JML+JUnit [CL ECOOP 02], Korat [BKM ISSTA 02], …

- Without specs, test oracles are not generated for correctness checking
  - infeasible to manually inspect
  - Insufficient to rely only on uncaught exceptions

- Solution: infer specs from test executions and select tests against inferred specs
  - select tests that violate inferred specs [ASE 03]
  - identify special and common tests
Automated Testing in the Absence of Specs

- Specifications help improve automated testing but they often don’t exist in practice
  - JML+JUnit [CL ECOOP 02], Korat [BKM ISSTA 02], …

- Without specs, test oracles are not generated for correctness checking
  - infeasible to manually inspect
  - Insufficient to rely only on uncaught exceptions

- Solution: **infer** specs from test executions and **select** tests against inferred specs

  **Benefits of spec-based testing can be obtained without the pain of writing the specifications!**
Synopsis

- Common and special tests
  - common tests ⊥ common behavior
e.g., non-full and non-empty bounded stack
  - special tests ⊥ special behavior
e.g., full or empty bounded stack
- Characterize common/special behavior with inferred statistical algebraic abstractions
Synopsis

- Common and special tests
  - common tests † common behavior
    e.g., non-full and non-empty bounded stack
  - special tests † special behavior
    e.g., full or empty bounded stack
- Characterize common/special behavior with inferred statistical algebraic abstractions
  - algebraic abstractions: in the form of axioms e.g.,
    \( \text{top}(\text{push}(S, e).\text{state}).\text{retval} == e \)
Synopsis

- Common and special tests
  - common tests $\dagger$ common behavior
    e.g., non-full and non-empty bounded stack
  - special tests $\dagger$ special behavior
    e.g., full or empty bounded stack

- Characterize common/special behavior with inferred statistical algebraic abstractions
  - algebraic abstractions: in the form of axioms e.g.,
    \[\text{top}\left(\text{push}(S, e).\text{state}\right).\text{retval} == e\]
  - statistical abstractions: e.g., 6 violating tests and 47 satisfying tests,
    $\neq$ universal abstractions [HD ECOOP 03][ECGN TSE 01]
Special and Common Test Identification

Class bytecode → Test generation

Test generation → Method-call composition

Method-call composition → Statistical inference

Statistical inference → Test identification

Test identification → Special tests, Common tests
Sample Abstraction Templates

- f(S, args1).state != S
  - removeFirst(S).state != S

- f(S, args1).retval == const
  - add(S, e).retval == true

- g(f(S, args1).state, args2).retval == args1.i
  - indexOf(add(S, i, e1).state, e2).retval == i
Statistics of Abstraction Templates

- 13 templates for method-exit states
  - e.g., \( f(S, \text{args1}).\text{state} \neq S \)

- 11 templates for method returns
  - e.g., \( f(S, \text{args1}).\text{retval} == \text{const} \)

- Conditional extension to 20 templates
  - e.g., \( \text{contains}(\text{add}(S, \text{e1}).\text{state}, \text{e2}).\text{retval} == \text{true where } (\text{e1} == \text{e2}) \)

- Difference extension to 11 templates
  - e.g., \( \text{size}(\text{add}(S, \text{e}).\text{state}).\text{retval} == \text{size}(S).\text{retval} + 1 \)

- Our templates instantiate all 146 but 2 axioms inferred by Henkel&Diwan [ECOOP 03] for ArrayList
Special and Common Test Identification

Class bytecode → Test generation → Method-call composition → Statistical inference → Statistical properties → Test identification → Special tests, Common tests
Test Generation

- Generate method arguments with JCrasher [CS SPE 04]
- Breadth-first exploration of receiver-object states with method calls with Rostra [ASE 04]

```
new LinkedList()
removeFirst()
addFirst(1)
addFirst(2)
```
Test Generation

- Generate method arguments with JCrasher [CS SPE 04]
- Breadth-first exploration of receiver-object states with method calls with Rostra [ASE 04]

Iteration 1

```
new LinkedList()
removeFirst()
```

```
addFirst(1)
addFirst(2)

1
2
```
Test Generation

- Generate method arguments with JCrasher [CS SPE 04]
- Breadth-first exploration of receiver-object states with method calls with Rostra [ASE 04]
Special and Common Test Identification

Class bytecode → Test generation → Method-call composition → Statistical inference → Test identification

- Statistical properties
- Special tests
- Common tests

Abstraction templates
Method-Call Composition

- **Goal**: compose method-call pair to instantiate LHS or RHS of an abstraction template
  - template LHS:
    
    \[ g(f(S, \text{args1}).\text{state}, \text{args2}).\text{state} \]

  - abstraction LHS:
    
    \[ \text{removeFirst(addFirst}(S, \ e)\ .\text{state})\ .\text{state} \]

  - abstraction instance LHS:
    
    \[ \text{removeFirst(addFirst}(\text{new LinkedList()}\ ,\ 1)\ .\text{state})\ .\text{state} \]
Special and Common Test Identification

Class bytecode

Test generation

Method-call composition

Statistical inference

Statistical properties

Test identification

Special tests

Common tests

Abstraction templates
Each statistical abstraction is associated with 
#satisfying instances and #violating instances

- template: \( g(f(S, \text{args1}).\text{state}, \text{args2}).\text{state} = f(g(S, \text{args2}).\text{state}, \text{args1}).\text{state} \)
- abstraction:
  \[
  \text{removeLast}(\text{addFirst}(S, e).\text{state}).\text{state} \\
  \quad = \text{addFirst}(\text{removeLast}(S).\text{state}, e).\text{state}
  \]

- 117 satisfying instances
- 3 violating instances

When \( S \) is an empty `LinkedList`, the abstraction is violated.
Special and Common Test Identification

Class bytecode → Test generation → Method-call composition → Statistical inference → Abstraction templates

- Statistical properties
  - Special tests
  - Common tests

Test identification
Test Identification

- Universal property
  - no violating instances

- Common property
  - a minority of violating instances (<20% by default)

- Special test
  - a violating instance of a common property
  - a satisfying instance of a conditional universal property
    unique bounded stack
    \[
    \text{push}(\text{push}(S, e1).\text{state}, e2).\text{state} = \text{push}(S, e2).\text{state}
    \]
    where \((e1 == e2)\)

- Common test
  - a satisfying instance of a common property or universal property
Experience

- Developed the Sabicu tool for the approach
- Applied it on 10 ADT (data structures) with test generation of 5 iterations
- Inferred 3 axioms for int stack (inferred by Henkel & Diwan [ECOOP 03])
- Inferred 10 of 12 manually written axioms for unique bounded stack [SLA XP 02]
  - all 8 universal axioms
  - 2 of 4 conditional axioms
  - one inferred conditional axiom is missing from manually written ones.
## Some Statistics

<table>
<thead>
<tr>
<th>class</th>
<th>m</th>
<th>properties</th>
<th></th>
<th>tests</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>univ</td>
<td>cond-univ</td>
<td>comm</td>
</tr>
<tr>
<td>BinarySearchTree</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>BinomialHeap</td>
<td>12</td>
<td>21</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>FibonacciHeap</td>
<td>9</td>
<td>12</td>
<td>6</td>
<td>80</td>
</tr>
<tr>
<td>HashMap</td>
<td>13</td>
<td>81</td>
<td>9</td>
<td>19</td>
</tr>
<tr>
<td>HashSet</td>
<td>8</td>
<td>43</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>LinkedList</td>
<td>21</td>
<td>55</td>
<td>18</td>
<td>39</td>
</tr>
<tr>
<td>SortedList</td>
<td>24</td>
<td>55</td>
<td>14</td>
<td>44</td>
</tr>
<tr>
<td>TreeMap</td>
<td>15</td>
<td>84</td>
<td>9</td>
<td>17</td>
</tr>
<tr>
<td>IntStack</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>UBStack</td>
<td>9</td>
<td>10</td>
<td>2</td>
<td>6</td>
</tr>
</tbody>
</table>
Related Work

- Daikon by Ernst et al. [TSE 01]
  - infer axiomatic specs (universal properties)
- Tool by Henkel & Diwan [ECOOP 03]
  - infer axioms (universal properties)
- Strauss by Ammons et al. [POPL 02]
  - infer probabilistic FSMs from call sequences
- Static analysis tool by Engler et al. [SOSP 01]
  - infer common call sequence patterns and deviations from them.
- Test selection based on specs, structural info…
Conclusion

- Specs help improve automated testing but they often don’t exist in practice
- Automatically generated test inputs don’t have test oracles
- Our new approach infers statistical properties and uses them to identify special and common tests

- In future work, we plan to investigate
  - fault detection capability of selected tests
  - static/dynamic verification tools to refute inferred properties
Questions?
One Common Property

\[
 remove(\text{removeLast}(S).\text{state}, \text{m1}).\text{state} \\
\quad = \text{removeLast}(\text{remove}(S, \text{m1}).\text{state}).\text{state}
\]

408 satisfying instances
42 violating instances