Rostra:  
A Framework for Detecting Redundant Object-Oriented Unit Tests

¹ Tao Xie  ² Darko Marinov  ¹ David Notkin

¹ Dept. of Computer Science & Engineering, University of Washington,  
² MIT Computer Science and Artificial Intelligence Laboratory (UIUC)  
23 Sept. 2004  
ASE 2004, Linz, Austria
Motivation

• Tool generated test cases
  • Many test cases
  • Important to reduce by eliminating “redundant” test cases
  • Need automation

• Common approach
  • Identify “similar” test cases and eliminate
  • Without reducing “quality” of test suite*

• Object-oriented programs
  • Test case is a sequence of method calls on an object
  • Note: Unit tests only

*Some reduction in fault detection may be tolerated!
Example Code

```
public class IntStack {
    private int[] store;
    private int size;
    public IntStack() { ... }
    public void push(int value) { ... }
    public int pop() { ... }
    public boolean isEmpty() { ... }
    public boolean equals(Object o) { ... }
}
```

[Henkel&Diwan 03]
Example Tests

Test 1 (T1):
IntStack s1 =
    new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);

Test 2 (T2):
IntStack s2 =
    new IntStack();
s2.push(3);
s2.push(5);

Test 3 (T3):
IntStack s3 =
    new IntStack();
s3.push(3);
s3.push(2);
s3.pop();
**Same inputs \(\Rightarrow\) Same behavior**

Assumption: deterministic method

\[
\text{Input} = \text{object state } @\text{entry} + \text{Method arguments}
\]

Testing a method with the same inputs is unnecessary

**How to represent object states?**
Redundant Test Cases Defined

• Equivalent method executions
  • the same method names, signatures, and input (equivalent object states @entry and arguments)

• Redundant test case:
  • A test case is redundant for a test suite if the test suite has exercised method executions equivalent to all method executions exercised by the test case
Related Work

• State equivalence using observational equivalence [Bernot et al. 91, Doong&Frankl 94, Henkel&Diwan 03]
  • for verifying or inferring algebraic specifications
  • Expensive because of number of sequences
• State equivalence based on user-defined abstraction functions [Grieskamp et al. 02]
  • AsmLT tool for conformance testing
  • Need to define the function
Five State-Representation Techniques

- Method-sequence representations
  - WholeSeq
    - The entire sequence
  - ModifyingSeq
    - Ignore methods that don’t modify the state

- Concrete-state representations
  - WholeState
    - The full concrete state
  - MonitorEquals
    - Relevant parts of the concrete state
  - PairwiseEquals
    - `equals()` method used to compare pairs of states
WholeSeq Representation

Method sequences that create objects
Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):
    IntStack s1 =
        new IntStack();
    s1.isEmpty();
    s1.push(3);
    s1.push(2);
    s1.pop();
    s1.push(5);

Test 3 (T3):
    IntStack s3 =
        new IntStack();
    s3.push(3);
    s3.push(2);
    s3.pop();
**WholeSeq Representation**

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):
IntStack s1 = new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);

Test 3 (T3):
IntStack s3 = new IntStack();
s3.push(3);
**s3.push(2);**
s3.pop();
WholeSeq Representation

Method sequences that create objects
Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

**Test 1 (T1):**
```java
IntStack s1 = new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);
```

**Test 3 (T3):**
```java
IntStack s3 = new IntStack();
s3.push(3);
s3.push(2);
s3.pop();
```

`isEmpty(<init>()).state`
WholeSeq Representation

Method sequences that create objects

Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):
IntStack s1 =
    new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);

Test 3 (T3):
IntStack s3 =
    new IntStack();
s3.push(3);
s3.push(2);
s3.pop();
WholeSeq Representation

Method sequences that create objects
Notation: methodName(entryState, methodArgs).state [Henkel&Diwan 03]

Test 1 (T1):
IntStack s1 =
   new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);

Test 3 (T3):
IntStack s3 =
   new IntStack();
s3.push(3);
s3.push(2);
s3.pop();

push(isEmpty(<init>( ).state ).state, 3).state  push(<init>( ).state, 3).state
ModifyingSeq Representation

State-modifying method sequences that create objects

**Test 1 (T1):**
```
IntStack s1 = new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);
```

**Test 3 (T3):**
```
IntStack s3 = new IntStack();
s3.push(3);
s3.push(2);
s3.pop();
```
WholeState Representation
The entire concrete state reachable from the object

Test 1 (T1):
IntStack s1 =
    new IntStack();
    s1.isEmpty();
    s1.push(3);
    s1.push(2);
    s1.pop();
    s1.push(5);

Test 2 (T2):
IntStack s2 =
    new IntStack();
    s2.push(3);
    s2.push(5);

store.length = 3
store[0] = 3
store[1] = 2
store[2] = 0
size = 1

5

s1.push

store.length = 3
store[0] = 3
store[1] = 0
store[2] = 0
size = 1

5

s2.push
MonitorEquals Representation

The relevant part of the concrete state defined by equals (invoking \texttt{obj.equals(obj)} and monitor field accesses)

\textbf{Test 1 (T1)}:
\begin{verbatim}
IntStack s1 = new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.push(2);
s1.pop();
s1.push(5);
\end{verbatim}

\textbf{Test 2 (T2)}:
\begin{verbatim}
IntStack s2 = new IntStack();
s2.push(3);
s2.push(5);
s2.push(5);
\end{verbatim}
The results of `equals` invoked to compare pairs of states

**Test 1 (T1):**
```
IntStack s1 =
    new IntStack();
s1.isEmpty();
s1.push(3);
s1.push(2);
s1.pop();
s1.push(5);
```

**Test 2 (T2):**
```
IntStack s2 =
    new IntStack();
s2.push(3);
s2.push(5);
```

- **Fundamental difference between MonitorEquals and PairwiseEquals**
- MonitorEquals monitors field accesses during execution of the `equals()` method and compares the monitored parts
- PairwiseEquals relies only on the output of the `equals()` method
- Example of sets
Detected Redundant Tests

Test 1 (T1):
IntStack s1 = new IntStack();
s1.isEmpty(); s1.push(3); s1.push(2); s1.pop(); s1.push(5);

Test 2 (T2):
IntStack s2 = new IntStack();
s2.push(3); s2.push(5);

Test 3 (T3):
IntStack s3 = new IntStack();
s3.push(3); s3.push(2); s3.pop();

<table>
<thead>
<tr>
<th>technique</th>
<th>detected redundant tests w.r.t. T1</th>
</tr>
</thead>
<tbody>
<tr>
<td>WholeSeq</td>
<td></td>
</tr>
<tr>
<td>ModifyingSeq</td>
<td>T3</td>
</tr>
<tr>
<td>WholeState</td>
<td>T3</td>
</tr>
<tr>
<td>MonitorEquals</td>
<td>T3, T2</td>
</tr>
<tr>
<td>PairwiseEquals</td>
<td>T3, T2</td>
</tr>
</tbody>
</table>
Experiment:
Evaluated Test Generation Tools

- ParaSoft Jtest 4.5
  - A commercial Java testing tool
  - Generates tests with method-call lengths up to three

- JCrasher 0.2.7
  - An academic robustness testing tool
  - Generates tests with method-call lengths of one
Questions to Be Answered

• How much do we benefit after applying Rostra on tests generated by Jtest and JCrasher?

• Does redundant-test removal decrease test suite quality?
## Experimental Subjects

<table>
<thead>
<tr>
<th>class</th>
<th>methods</th>
<th>public methods</th>
<th>ncnb loc</th>
<th>Jtest tests</th>
<th>JCrasher tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>IntStack</td>
<td>5</td>
<td>5</td>
<td>44</td>
<td>94</td>
<td>6</td>
</tr>
<tr>
<td>UBStack</td>
<td>11</td>
<td>11</td>
<td>106</td>
<td>1423</td>
<td>14</td>
</tr>
<tr>
<td>ShoppingCart</td>
<td>9</td>
<td>8</td>
<td>70</td>
<td>470</td>
<td>31</td>
</tr>
<tr>
<td>BankAccount</td>
<td>7</td>
<td>7</td>
<td>34</td>
<td>519</td>
<td>135</td>
</tr>
<tr>
<td>BinSearchTree</td>
<td>13</td>
<td>8</td>
<td>246</td>
<td>277</td>
<td>56</td>
</tr>
<tr>
<td>BinominalHeap</td>
<td>22</td>
<td>17</td>
<td>535</td>
<td>6205</td>
<td>438</td>
</tr>
<tr>
<td>DisjSet</td>
<td>10</td>
<td>7</td>
<td>166</td>
<td>779</td>
<td>64</td>
</tr>
<tr>
<td>FibonacciHeap</td>
<td>24</td>
<td>14</td>
<td>468</td>
<td>3743</td>
<td>150</td>
</tr>
<tr>
<td>HeapMap</td>
<td>27</td>
<td>19</td>
<td>597</td>
<td>5186</td>
<td>47</td>
</tr>
<tr>
<td>LinkedList</td>
<td>38</td>
<td>32</td>
<td>398</td>
<td>3028</td>
<td>86</td>
</tr>
<tr>
<td>TreeMap</td>
<td>61</td>
<td>25</td>
<td>949</td>
<td>931</td>
<td>1000</td>
</tr>
</tbody>
</table>
Assumptions About Subjects

• Method-sequence representations assume that each method does not modify argument state
• MonitorEquals and PairwiseEquals representations assume a user-defined `equals()`
## Quality of Original Test Suites

<table>
<thead>
<tr>
<th></th>
<th>Jtest-generated tests</th>
<th>JCrasher-generated tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Avg num uncaught exceptions</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Avg Branch cov</td>
<td>77%</td>
<td>52%</td>
</tr>
<tr>
<td>Avg mutant killing ratio (600 mutants)</td>
<td>53%</td>
<td>30%</td>
</tr>
</tbody>
</table>
Elapsed Real Time in Minimizing Jtest-Generated Tests (in secs)
Elapsed Real Time in Minimizing JCrasher-Generated Tests (in secs)

- IntStack
- UBStack
- ShoppingCart
- BankAccount
- BinSearchTree
- BinomialHeap
- DisjSet
- FibonacciHeap
- HashMap
- LinkedList
- TreeMap

Legend:
- WholeSeq
- ModifyingSeq
- WholeState
- MonitorEquals
- PairwiseEquals
The last three techniques detect around 90% redundant tests
Detected redundancy in increasing order for five techniques
Redundancy among JCrasher-generated Tests

- The last three techniques detect over 50% on half subjects
- JCrasher generates fewer tests and shorter tests
Quality of Minimized Test Suites

- All five techniques on JCrasher preserve all measurements.
- The first three techniques on Jtest preserve all measurements.
- Two equals techniques on Jtest decrease (with only small loss in 2 programs)
  - in branch cov %
  - in mutant killing %
Comparison of Five Techniques

• Time and space taken to find redundant tests
  • from a couple of seconds to several minutes across subjects
  • in roughly increasing order except for pairwiseEquals (being the least expensive)

• The number of redundant tests found
  • in increasing order
Conclusions

• Redundant tests add cost without any benefit
• Existing test generation tools can be potentially improved (by incorporating Rostra framework)
• The experimental results have shown
  • High redundancy among their generated tests
  • Removing them does not decrease test suite quality
• Rostra framework useful in test minimization, assessment, selection, and generation
Evolution of ParaSoft Jtest

- Version 4.5 (released in March 2002) allows method-call lengths (1 — 3) [studied in this work]
- Version 5.0 (released in Feb 2004) allows method-call length of only 1
- ParaSoft notified the authors last week that Version 6.0 (internal version, not yet released) has addressed the test redundancy issue identified by the authors and added back the option to generate long call sequence
Questions?
Threats to Validity

• Representative of true practice?
  • Subject programs, third-party test generation tools

• Instrumentation effects that bias the results
  • Faults on tools (Rostra, Jtest, JCrasher, measurement tools)
Applications

• **Assessment:** compare the quality of different test suites.
• **Selection:** select a subset of automatically generated tests to augment an existing test suite.
• **Minimization:** minimize an automatically generated test suite for correctness inspection and regression executions.
• **Generation:** avoid generating and executing redundant tests
Related Work

- Test selection or minimization with some loss in the quality of test suites [Rothermel et al. 98, Chang & Richardson 99, Harder et al. 03, Xie & Notkin 03]
- for regression testing or test inspection