Advances in Coverage-Based Test Suite Reduction
About Me

- Ph.D., University of Maryland, College Park (2008).
  - Research interests include Software Testing, Program Analysis, Software Tools, and Distributed Systems.
- Professional Software Developer
  - Microsoft, Lockheed Martin, Amazon.com, etc.
Agenda

- Background
- Call Stack Coverage for Test Suite Reduction
- Fault Correlation and the Average Probability of Detecting Each Fault
- Other Advances and Future Directions
Automated Test Case Generation Techniques

- Code-based (Parasoft, Agitar, etc.)
- Model-based (GUITAR, etc.)
- May generate enormous volume of tests

New Development Methodologies

- Continuous integration
- Rapid test cycles

Automated test case generation may result in too many tests to run in a given build/test/deploy process.
Test Suite Reduction

- Reduce the number of test cases in a test suite, and:
- Maintain as much of the original suite’s fault detection effectiveness as possible.
- Most common approaches are based on maintaining coverage relative to some criterion.
  - **Coverage Requirements** are logical or program elements that must be exercised by test cases.
  - Examples: Branches, lines, dynamic program invariants, etc.
- Traditionally evaluated against conventional, batch-oriented applications, using test suites built using category-partition or similar methods.
Characteristics of Modern Software

- Object- and aspect-oriented
- Use of reflection
- Use of callbacks
- Multithreading
- Extensive use of libraries and frameworks
- Multi-language development
- Event-reactive paradigm
  - Handler code may be invoked from multiple contexts

- An effective test coverage technique should account for these factors.
Dissertation Contributions

- Test suite reduction technique based on the call stack coverage criterion.
  - Formal model of call stacks, including notion of maximum-depth call stack.
- Empirical studies of test suite reduction in modern versus conventional software applications.
- Development of new metrics for looking at the problem of test suite reduction.
- Guidance for practitioners considering test suite reduction.
- Improvements to the practice of GUI test automation.
- Reusable tools and data.
Call Stacks

- Sequence of active calls associated with each thread of a running program.

- Stack where:
  - Methods are pushed on when they are called.
  - Methods are popped off when they return or throw an exception.
Call Stack - Example

(Ljava/lang/Object;ILjava/lang/Object;II)V Ljava/lang/System;arraycopy
([BII)V Ljava/io/BufferedOutputStream;write
([BII)V Ljava/io/PrintStream;write
()V Lsun/nio/cs/StreamEncoder$CharsetSE;writeBytes
()V Lsun/nio/cs/StreamEncoder$CharsetSE;implFlushBuffer
()V Lsun/nio/cs/StreamEncoder;flushBuffer
()V Ljava/io/OutputStreamWriter;flushBuffer
()V Ljava/io/PrintStream;newLine
(Ljava/lang/String;)V Ljava/io/PrintStream;println
([Ljava/lang/String;)V LHelloWorldApp;main

Full Method Signature (Canonical Representation)
Using call stacks as a coverage criterion addresses challenges posed by modern software applications.

Call stacks:
- Are easily collected in a multi-language and/or multi-threaded environment.
- Automatically identify and resolve reflective and virtual method calls, woven aspects, and callbacks.
- Capture differences in context when methods are called.

Note that this application only uses dynamic call stacks.
Efficient data structure is the calling context tree (CCT).

- Nodes are methods and edges are method calls.
- Traverse all paths to leaves to find maximum-depth call stacks.
- Multithreaded extension is to maintain one CCT per thread and merge at the end.

JavaCCTAgent (http://sourceforge.net/projects/javacctagent)
- Tool for collecting CCTs for Java programs
Calling Context Tree

java/io/OutputStreamWriter;flushBuffer
java/io/PrintStream;newLine
java/io/PrintStream;println
HelloWorldApp;main

java/io/BufferedWriter;newLine
java/io/PrintStream;newLine
java/io/PrintStream;println
LHelloWorldApp;main

java/io/PrintStream;write
java/io/PrintStream;print
java/io/PrintStream;println
HelloWorldApp;main
Traditional Test Suite Reduction Metrics

- **% Size Reduction**
  - $100 \times (1 - \frac{\text{Size}_{\text{Reduced}}}{\text{Size}_{\text{Full}}})$

- **% Fault Detection Reduction**
  - $100 \times (1 - \frac{\text{FaultsDetected}_{\text{Reduced}}}{\text{FaultsDetected}_{\text{Full}}})$

→ *Test coverage is not explicitly used in these metrics.*
One might expect a correlation between coverage requirements and the faults exposed by test cases that hit them. But no existing measure explores this notion.

Proposal: *Average Probability of Detecting Each Fault*

- Captures the likelihood that coverage-equivalent reduced test suites will detect the same faults as their original counterparts.
- Driven by the frequency that coverage requirements get hit by fault-detecting test cases (*fault correlation*).
- Varies greatly by coverage criterion.
  - Useful for selecting the best coverage criterion for test suite reduction.
Intuition: Certain coverage requirements are more likely to be associated with fault-producing program states.

From the coverage matrix and fault matrix, we can calculate the fault correlation.

Given:
1. The set of test cases.
2. A specific known fault.
3. A specific coverage requirement.

Fault correlation is the ratio of (test cases that hit the coverage requirement and detect the fault) to (test cases that merely hit the coverage requirement).
From fault correlations, we can calculate the...

**Average the expected probability of finding each fault** across all known faults in an experiment.

→ *Evaluated in the subsequent experiments.*
Experiments

1. Compare size and fault detection reduction of call-stack-reduced suites to suites reduced based on other criteria.
2. Compare fault detection of call-stack-reduced suites to suites of the same size created using other approaches.
3. Evaluate the impact of including coverage of third-party library code in test suite reduction.
4. Compare call-stack-based reduction in conventional versus event-driven applications.
5. Test whether certain coverage criteria are more highly associated with faults.
Experimental and Analytical Process
Experimental Infrastructure

- **Subject Applications**
  - TerpOffice
  - Space
  - nanoxml

- **Coverage Tools**
  - Java CCTAgent
  - Detours-based library for CCT collection in Win32 applications
  - jcoverage / Cobertura

- **JavaGUIReplayer**

- **Test Suite Reduction Implementation**
  - HGS algorithm (implemented in C#)
  - Custom test harnesses to tie these tools together
### Subject Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Source Language</th>
<th>Execution Style</th>
<th>Programming Style</th>
<th>Test Universe Size</th>
<th># Detectable Faults (Versions)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TerpPaint (TP)</td>
<td>Java</td>
<td>Event-Driven (GUI)</td>
<td>Object-Oriented</td>
<td>1500</td>
<td>43</td>
</tr>
<tr>
<td>TerpWord (TW)</td>
<td>Java</td>
<td>Event-Driven (GUI)</td>
<td>Object-Oriented</td>
<td>1000</td>
<td>18</td>
</tr>
<tr>
<td>TerpSpreadsheet (TS)</td>
<td>Java</td>
<td>Event-Driven (GUI)</td>
<td>Object-Oriented</td>
<td>1000</td>
<td>101</td>
</tr>
<tr>
<td>Space</td>
<td>C</td>
<td>Conventional</td>
<td>Procedural</td>
<td>13585</td>
<td>34</td>
</tr>
<tr>
<td>nanoxml</td>
<td>Java</td>
<td>Conventional</td>
<td>Object-Oriented</td>
<td>216</td>
<td>9</td>
</tr>
</tbody>
</table>

Good subjects are hard to find. You need:
- Test cases
- Known faults
## Subject Application Metrics

<table>
<thead>
<tr>
<th></th>
<th>Includes Library Data?</th>
<th>TerpPaint (TP)</th>
<th>TerpWord (TW)</th>
<th>TerpSpreadsheet (TS)</th>
<th>Space</th>
<th>Nanoxml</th>
</tr>
</thead>
<tbody>
<tr>
<td># Call Stacks Observed</td>
<td>Yes</td>
<td>413166</td>
<td>569933</td>
<td>333882</td>
<td>453</td>
<td>6617</td>
</tr>
<tr>
<td># Methods Observed</td>
<td>Yes</td>
<td>12277</td>
<td>12665</td>
<td>11103</td>
<td>143</td>
<td>1126</td>
</tr>
<tr>
<td># Events</td>
<td>N/A</td>
<td>181</td>
<td>219</td>
<td>110</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td># Executable Lines</td>
<td>No</td>
<td>11803</td>
<td>9917</td>
<td>5381</td>
<td>6218</td>
<td>3012</td>
</tr>
<tr>
<td># Classes</td>
<td>No</td>
<td>330</td>
<td>197</td>
<td>135</td>
<td>N/A</td>
<td>25</td>
</tr>
<tr>
<td># Methods</td>
<td>No</td>
<td>1253</td>
<td>1380</td>
<td>746</td>
<td>123</td>
<td>232</td>
</tr>
</tbody>
</table>
Standard Approaches
- Call Stack (CS)
- Line (L)
- Method (M)
- Random (RAND)
- Event (E1)
- Event-Interaction (E2)

“Additional” Approaches (adds random cases to match CS size)
- Line-Additional (LA)
- Method-Additional (MA)
- Event-Additional (E1A)

“Short” Approaches (excludes library methods)
- Short Call Stack (SCS)
- Short Method (SM)
Size Reduction (GUI Application)
Size Reduction (Conventional Application)

![Graph showing nanoxml - % Size Reduction vs. Original Suite Size]
GUI Applications

- E2 displays very little size reduction (expected because test case generation was E2-based).
- Other non-CS techniques perform similarly.
- CS strikes a middle ground (38-50% reduction for largest suite size).

Conventional Applications

- CS still yields less reduction than comparison techniques.
- But closer than in the GUI subjects.
Fault Detection Reduction (GUI Applications)
Fault Detection Reduction (Conventional Applications)

- **nanoxml - % Fault Detection Reduction**
  - Graph showing the average reduction in fault detection over 25 suites as a function of the original suite size.
  - Different algorithms (CS, RAND, M, L, LA, MA, SCS, SM) are compared.

- **Space - % Fault Detection Reduction**
  - Graph showing the average reduction in fault detection over 50 suites as a function of the original suite size.
  - Different algorithms (CS, RAND, M) are compared.
GUI Applications

- Call-Stack-based reduction (CS) loses only 0-5% of detectable faults.
  - Comparable to E2, even though E2 displays almost no size reduction.
- Other techniques perform comparably to one another.

Conventional Applications

- CS performs well for space, not for Nanoxml.
  - Nanoxml has only 9 faults, and 7 are very easy to find (allowing techniques with random selection to perform well).
## Coverage Requirements and Fault-Revealing Test Cases

- Which coverage criterion’s requirements are best correlated with fault-revealing test cases?
- Use the *average probability of detecting each fault* metric against the full universe of test cases.

<table>
<thead>
<tr>
<th></th>
<th>TP</th>
<th>TS</th>
<th>TW</th>
<th>nanoxml</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>0.51</td>
<td>0.52</td>
<td>0.47</td>
<td>--</td>
</tr>
<tr>
<td>E2</td>
<td>0.92</td>
<td>0.88</td>
<td>0.96</td>
<td>--</td>
</tr>
<tr>
<td>L</td>
<td>0.84</td>
<td>0.69</td>
<td>0.77</td>
<td>1.00</td>
</tr>
<tr>
<td>M</td>
<td>0.80</td>
<td>0.69</td>
<td>0.72</td>
<td>0.81</td>
</tr>
<tr>
<td>CS</td>
<td>1.00</td>
<td>0.97</td>
<td>0.97</td>
<td>0.997</td>
</tr>
<tr>
<td>SM</td>
<td>0.70</td>
<td>0.68</td>
<td>0.61</td>
<td>0.81</td>
</tr>
<tr>
<td>SCS</td>
<td>0.73</td>
<td>0.85</td>
<td>0.77</td>
<td>0.94</td>
</tr>
</tbody>
</table>
Individual Fault Probabilities

![Graph showing individual fault probabilities for various categories: E1, E2, L, M, CS, SM, SCS. The graph includes data points for Q1, MIN, MEDIAN, MAX, and Q3.](image)
Dissertation Bibliography


Other Advances and Future Directions

- Automated GUI Test Case Maintenance
- Using Annotations in GUI Testing
  - Test Oracles
  - Test Case Generation
Automated GUI Test Case Maintenance

- Test case replayers need to find the right elements to act upon when GUIs are modified.
- Automated approach is based on *heuristics* (same-label, same-position, etc.).

Example GUI Test Case

1. {FindTextBox, setText('GUI')}
2. {CaseSensitiveCheckBox, click}
3. {FindButton, click}
4. {CancelButton, click}
GUI Modification

Version 1

Version 2
What About the Test Case?

1. {FindTextBox, setText('GUI')}

2. {CaseSensitiveCheckBox, click}

3. {FindButton, click}

4. {CancelButton, click}

=> Test Case is BROKEN!!!
The Fix

1. {FindTextBox, setText('GUI')}

2. {MatchCaseCheckBox, click}

3. {FindButton, click}

4. {CancelButton, click}

Can the fix be automated?
Classify each GUI element into one of three sets:

1. **Created** - elements which are new in the new version of the GUI.
2. **Deleted** - elements from the old version of the GUI which do not appear in the new version.
3. **Maintained** – elements which have been kept and possibly modified between versions.

Calculating these sets requires heuristic approaches.

- Cannot work on arbitrary GUI modifications.
- Focus is on building an accurate *Maintained* set for relatively small modifications.
GUIAnalyzer

- Automated framework for GUI element identification.
- Builds GUI models from windows/dialogs in Java Swing applications.
- Performs GUI element identification using customizable, extensible heuristic sets.
  - Heuristics are applied in order of definition.
  - Multiple passes are made until the process converges.
Applying heuristics, pass 1
javax.swing.JLabel:Find: identified by SameTextHeuristic as javax.swing.JLabel:Find:
javax.swing.JCheckBox:Whole Words Only identified by SameTextHeuristic as
javax.swing.JCheckBox:Whole Words Only
javax.swing.JButton:Find Next identified by SameTextHeuristic as javax.swing.JButton:Find Next
javax.swing.JButton:Cancel identified by SameTextHeuristic as javax.swing.JButton:Cancel
javax.swing.JTextField:null identified by SamePreviousSiblingHeuristic as
javax.swing.JTextField:null
javax.swing.JCheckBox:Match Case identified by SamePreviousSiblingHeuristic as
javax.swing.JCheckBox:Case-Sensitive
Applying heuristics, pass 2
Done

1. “Whole Words Only” checkbox is identified by its label.
2. “Case-Sensitive” checkbox is presumed to be the same as the old
   “Match Case” checkbox by its position in the element hierarchy.
3. Heuristics identify no further elements → termination.
Research Agenda for Automated GUI Test Case Maintenance

- Evaluate the effectiveness of different heuristics, heuristic sets and priorities.
  - Metrics
    1. False Positives (misidentified elements from original version).
    2. False Negatives (unidentified elements from original version).
- Empirical studies using a variety of GUI windows/dialogs with multiple versions and different-sized modifications.
- New techniques
  - Evaluate test case executability with a proposed Maintained set.
  - Apply multiple heuristic sets simultaneously.
Oracles for GUI testing have been rather limited.
- “Crash-testing”

Researchers and practitioners are leveraging annotations (source-code-based metadata) for program analysis and bug detection.
- JSR 305, JSR 308
- @NonNull, @NullFeasible, @NonNegative, etc.

→ Idea: Define annotations for GUI state invariants, and a framework that test case replayers can use to verify them.
CrosswordSage

- Open-source application.
- Has several menu items that should be disabled but aren’t (leads to unhandled exceptions).

MainScreen.java (annotated)

```java
private CrosswordCompiler cc;

@Enable("cc != null")
JMenuItem mFile_Print = new JMenuItem();

@Enable("cc != null")
JMenuItem mAction_Publish = new JMenuItem();
```
JUnit/Jemmy test case that checks CrosswordSage MainScreen:

```java
private JFrameOperator mainFrame;

@Before
public void setUp() throws Exception {
    new ClassReference("crosswordsage.MainScreen").startApplication();
    mainFrame = new JFrameOperator("Crossword Sage");
}

private void checkGUI() throws Exception {
    GUIAnnotationChecker checker = new GUIAnnotationChecker();
    List<GUIInvariantViolation> result = checker.check(mainFrame.getSource());
    for( GUIInvariantViolation violation : result ) {
        System.err.println(violation);
    }
    assertTrue("Got GUI invariant violations", result.isEmpty());  // FAILS
}
```

Prints:  mFile_Print was enabled but shouldn't be
         mAction_Publish was enabled but shouldn't be
Annotiations for GUI Test Case Generation

- Idea: If we have GUI element invariants defined in annotations, we should be able to use them to generate test cases that cover the invariant conditions.
Questions

Advances in Coverage-Based Test Suite Reduction

Scott McMaster
University of Maryland – College Park
mailto:scottmcm@cs.umd.edu
mailto:smcmaster@acm.org