Static and Dynamic Analysis of Test Suites

Patrick Lam
University of Waterloo

October 3, 2015
Goal

Convince you to analyze tests (with their programs).
Why do we analyze programs?
Motivations for Program Analysis

- enable program understanding & transformation;
- eliminate bugs;
- ensure software quality.
Observation

Programs come with gobs of tests.
Pervasiveness of Test Suites

\[ m = 0.3002 \]

\[ m = 0.03514 \]
Tests and Modern Software

Extensive library support:

JUnit

NUnit

PHPUnit
What Tests Do

Tests encode:
- how to invoke the system under test; and,
- what it should do.
Opportunity

Leverage test suites in program analysis.
Related Work
Static Analysis Limitations
Dynamic Analysis Limitations
Concolic analysis

Run the program with arbitrary inputs, (some symbolic); use solver to find new paths/inputs.
Choose program inputs and run the program.
Observe the configuration loading phase.
Use configuration information to do type analysis on the remainder of the program.
TamiFlex (Bodden et al)

- Choose program inputs and run the program.
- Observe classes loaded through reflection and custom classloaders.
Goal: learn quantified array invariants.

Approach: observe from test runs; namely:
- guess coarse templates;
- run with simple random tests;
- generate constraints;
- validate types.
DSD-Crasher (Csallner and Smaragdakis)

Existing tests:
\[ m(1); m(2); \ldots \]

**Dynamic**
- Run tests: Infer spec
  - Testee: \( m(\text{int } p) \)
  - Inferred spec: \( p > 0 \)

**Static**
- Search for bug
  - New test: \( \{m(7);\} \)

**Dynamic**
- Run test: Confirm
  - New result: \( m \text{ crash} \)
Empirical Studies
Complexity of Test Suites

- apache-cc: Longest bar
- google-vis
- hsqldb
- jdom
- jfreechart
- jgrapht
- jmeter
- joda-time
- poi
- weka

Bar chart showing the percentage of tests with >= 5 asserts for various projects.
% Tests with control-flow

- Apache-cc
- Google-Vis
- HsqlDB
- JDOM
- JFreeChart
- JGraphT
- JMeter
- Joda-Time
- POI
- Weka

% tests

- Branches
- Loops
Similar Test Methods
(a case study in static analysis of tests)

(with Felix Fang)
By Joonspoon (Own work) [CC BY-SA 4.0 (http://creativecommons.org/licenses/by-sa/4.0)], via Wikimedia Commons
Story: Writing Widget class

```
1 class FooWidget extends Widget {
    /*...*/
2 }
3
4
5 class BarWidget extends Widget {
    /*...*/
6 }
7
1 @Test
2
class FooWidgetTest {
3    /*...*/
4 }
5
6 @Test
class BarWidgetTest {
7    /*...*/
8 }
9 }
```
Story: Writing New Code

```java
class BazWidget extends Widget {
}

@Test
class BazWidgetTest {
    /* New Code */
    /* ? */
}
```
class BazWidget extends Widget {
    /* New Code */
}

@正因为 BazWidgetTest {
    /* Ctrl-C, Ctrl-V */
}
Hypothesis

- Developers often copy-paste tests (and probably enjoy doing it).
- Why? JUnit test cases tend to be self-contained.
- Test clones can become difficult to comprehend or maintain later.
Benefits of Refactoring Tests

- xUnit Test Patterns: Refactoring Test Code (Meszaros)

- Reduce long term maintenance cost (Saff)

- Can detect new defects and increase branch coverage if tests are parametrized (Thummalapenta et al)

- Reduce brittleness and improve ease of understanding
Refactoring Techniques

- Language features such as inheritance or generics
- Parametrized Unit Tests and Theories
Refactoring Example

```java
public void testNominalFiltering() {
    m_Filter = getFilter(Attribute.NOMINAL);
    Instances result = useFilter();
    for (int i = 0; i < result.numAttributes(); i++)
        assertTrue(result.attribute(i).type() != Attribute.NOMINAL);
}

public void testStringFiltering() {
    m_Filter = getFilter(Attribute.STRING);
    Instances result = useFilter();
    for (int i = 0; i < result.numAttributes(); i++)
        assertTrue(result.attribute(i).type() != Attribute.STRING);
}
```
Refactored Example

```java
static final int[] filteringTypes = {
    Attribute.NOMINAL, Attribute.STRING,
    Attribute.NUMERIC, Attribute.DATE
};

public void testFiltering() {
    for (int type : filteringTypes)
        testFiltering(type);
}

public void testFiltering(final int type) {
    m_Filter = getFilter(type);
    Instances result = useFilter();
    for (int i = 0; i < result.numAttributes(); i++)
        assertTrue(result.attribute(i).type() != type);
}
```
Our Contributions

- Test refactoring candidate detection technique using *Assertion Fingerprints*
- Empirical and qualitative analyses of results on 10 Java benchmarks
What is a test case made of?

- Setup
- Run/Exercise
- Verify
- Teardown
What is a test case made of?

- Setup
- Run/Exercise
- Verify
- Teardown
Similar tests often have similar sets of asserts.
Similar Sets of Asserts: Example

1 public void test1() {
2     /* ... */
3     assertEquals(int, int);
4     /* ... */
5     assertEquals(float, float);
6     /* ... */
7     assertTrue(boolean);
8 }

1 public void test2() {
2     /* ... */
3     assertEquals(int, int);
4     /* ... */
5     assertEquals(float, float);
6     /* ... */
7     assertTrue(boolean);
8 }
Our Approach

Assertion Fingerprints
Assertion Fingerprints

Augment set of assertions.

For each assertion call, collect:

- Parameter types
- Control flow components
Using Assertion Fingerprints

Group tests with similar assert structures.
Assertion Fingerprints: Control Flow Components

- Branch Count (Branch Vertices)
- Merge Count (Merge Vertices)
- In Loop Flag (Depth First Traversal)
- Exceptional Successor Count (Exceptional CFG)
- In Catch Block Flag (Dominator Analysis)
public void test() {
    int i;
    for (i = 0; i < 10; ++i) {
        if (i == 2)
            assertEquals(i, 2);
        assertTrue(i != 10);
    }
    try {
        throw new Exception();
        fail("Should have thrown exception");
    } catch (final Exception e) {
        assertEquals(i, 10);
    }
}
An assertion inside an if statement is likely to be different from one that is not.
Minimal number of branches needed to reach that vertex from the start of a method, excluding n.
An assertion with some prior operations inside an `if` statement is likely to be different from one that is not with any.
Example: CFG

```
/* line 3 */
{i < 10}

try:
/* line 10 */
{¬{i < 10}}
fail(String)

catch:
/* line 12 */
{¬{i < 10}}
assertEquals(int, int)

catch: Exception
```

```
/* line 4 */
* bc:1 {{i < 10}}
inLoop:true
*i == 2

/* line 5 */
* bc:2 {{i < 10}, {i == 2}}
inLoop:true
assertEquals(int, int)
```

```
/* line 6 */
* bc:1 {{i < 10}}
mc:1 {{i == 2}}
inLoop:true
*i != 10
```

```
/* line 6 */
* bc:1 {{i < 10}}
mc:2 {{i == 2}, {i != 10}}
inLoop:true
assertTrue(boolean)
```
Merge Count

- Minimal number of merge vertices needed to reach vertex \( n \) from the start of the method, including \( n \).
In-Loop Flag: Intuition

- An assertion inside a loop is likely to be different from one that is not.
Example: CFG

```plaintext
/* line 3 */
{i < 10}

try:
/* line 10 */
* bc:1 (¬{i<10})
* es:1
*/
fail(String)

catch:
/* line 12 */
* bc:1 (¬{i<10})
* inCatch:true
*/
assertEquals(int, int)

/* line 4 */
* bc:1 ({i<10})
* inLoop:true
*/
i == 2

/* line 5 */
* bc:2 ({i<10}, {i==2})
* inLoop:true
*/
assertEquals(int, int)

/* line 6 */
* bc:1 ({i<10})
* mc:1 ({i==2})
* inLoop:true
*/
i != 10

/* line 6 */
* bc:1 ({i<10})
* mc:2 ({i==2}, {i!=10})
* inLoop:true
*/
assertTrue(boolean)
```
An assertion inside a try block with corresponding catch block(s) is likely to be different from one that is not.
Example: CFG

```
/* line 3 */
{i < 10}

try:
/* line 10 */
{¬{i < 10}}
fail(String)

catch:
/* line 12 */
{¬{i < 10}}
assertEquals(int, int)
```

```
/* line 4 */
{bc:1 {{i < 10}}} inLoop:true
{i == 2}

/* line 5 */
{bc:2 {{i < 10}, {i == 2}}} inLoop:true
assertEquals(int, int)
```

```
/* line 6 */
{bc:1 {{i < 10}}} mc:1 {{i == 2}} inLoop:true
{i != 10}
assertEquals(int, int)
```
In-Catch-Block Flag: Intuition

- An assertion inside a catch block is likely to be different from one that is not.
Filtering: Must satisfy one of the following

- Contain some control flow
- Contain more than 4 assertions
- Heterogeneous in signature (invoke different assertion types)
Evaluation: Implementation

@Test A()

@Test B()

@Test C()

Soot

CFG(A) → [fingerprint(A)]

CFG(B) → [fingerprint(B)]

CFG(C) → [fingerprint(C)]
## Evaluation: Benchmarks

<table>
<thead>
<tr>
<th></th>
<th>Test LOC</th>
<th>Total LOC</th>
<th>% Test LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache POI</td>
<td>86 113</td>
<td>247 799</td>
<td>35%</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>46 129</td>
<td>110 394</td>
<td>42%</td>
</tr>
<tr>
<td>Google Visualization</td>
<td>13 440</td>
<td>31 416</td>
<td>43%</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>30 481</td>
<td>32 208</td>
<td>95%</td>
</tr>
<tr>
<td>JDOM</td>
<td>25 618</td>
<td>76 734</td>
<td>33%</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>93 404</td>
<td>317 404</td>
<td>29%</td>
</tr>
<tr>
<td>JGraphT</td>
<td>12 142</td>
<td>41 801</td>
<td>29%</td>
</tr>
<tr>
<td>JMeter</td>
<td>20 260</td>
<td>182 293</td>
<td>11%</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>67 978</td>
<td>134 758</td>
<td>50%</td>
</tr>
<tr>
<td>Weka</td>
<td>26 270</td>
<td>495 198</td>
<td>5%</td>
</tr>
</tbody>
</table>
### Evaluation: Analysis Run Time (seconds)

<table>
<thead>
<tr>
<th>Library</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Apache POI</td>
<td>113</td>
</tr>
<tr>
<td>Commons Collections</td>
<td>50</td>
</tr>
<tr>
<td>Google Visualization</td>
<td>240</td>
</tr>
<tr>
<td>HSQLDB</td>
<td>233</td>
</tr>
<tr>
<td>JDOM</td>
<td>25</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>70</td>
</tr>
<tr>
<td>JGraphT</td>
<td>43</td>
</tr>
<tr>
<td>JMeter</td>
<td>70</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>45</td>
</tr>
<tr>
<td>Weka</td>
<td>91</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>994</strong></td>
</tr>
</tbody>
</table>
Results: Detected Refactoring Candidates

- Apache POI: 21%
- Commons Collections: 42%
- Google Visualization: 37%
- HSQLDB: 28%
- JDOM: 23%
- JFreeChart: 48%
- JGraphT: 16%
- JMeter: 25%
- Joda-Time: 55%
- Weka: 53%
Results: Asserts vs Methods distribution

![Chart showing the distribution of asserts vs methods]
Results: Sampling, 75% True Positive rate

- Apache POI: 30% True Positives, 15% Fragmented True Positives
- Commons Collections: 85% True Positives, 10% Fragmented True Positives
- Google Visualization: 60% True Positives, 35% Fragmented True Positives
- HSQLDB: 75% True Positives, 15% Fragmented True Positives
- JDOM: 85% True Positives, 15% Fragmented True Positives
- JFreeChart: 70% True Positives, 15% Fragmented True Positives
- JGraphT: 100% True Positives, 0% Fragmented True Positives
- JMeter: 85% True Positives, 15% Fragmented True Positives
- Joda-Time: 95% True Positives, 5% Fragmented True Positives
- Weka: 80% True Positives, 15% Fragmented True Positives
Qualitative Analysis: JGraphT

- Small and heterogenous tests that are unlikely to be false positives
Qualitative Analysis: Joda-Time

- Wide hierarchy of tests with identical structures and straight-line assertions.
Qualitative Analysis: Weka, Apache Commons Collections

- Textually identical clones of methods with similar data types but with different environment setups.
public void test_TCC___String() {
    // [... 4x assertTrue(String, boolean)]
    try {
        // ...
        fail("allowed creation of an element with no name");
    } catch (IllegalArgumentException e) {
        // Test passed!
    }
}
Complex query-related statements and helper methods reduce the roles of assertions in a test method, resulting in a below-average true positive rate.
Qualitative Analysis: Refactorability

- Test methods that show structural similarities are most likely amenable to refactoring, however;
- Non-parametrized and small methods are difficult to refactor.
Guided test refactoring.
Future Perspectives
What Do We Do With Tests?

Traditionally:

run the test, get yes/no answer.

(Also, can combine with DSD/concolic analysis.)
Our usual interaction with tests (static)

Statically, we usually just ignore tests.
Our usual interaction with tests (dynamic)

Tests → Program → Analysis Tool → Generated Tests

Tests are write-only with respect to the tool.
A Better Way

Tests → Program

Analysis Tool

Generated Tests
Why is it hard to write tests?

Need to:
1. get system under test in appropriate state;
2. decide what the right answer is.

Useful hints for static analysis!
Unit tests also illustrate. . .

- interesting points in execution space, with
- complete execution environments for program fragments.
Challenges

How to combine information from test runs?
What can we learn from failing tests?
Conclusions
We can go beyond test generation.

Tests are a valuable source of information about their associated programs.