An Empirical Comparison of Automated Generation and Classification Techniques for Object-Oriented Unit Testing

Marcelo d’Amorim (UIUC)
Carlos Pacheco (MIT)
Tao Xie (NCSU)
Darko Marinov (UIUC)
Michael D. Ernst (MIT)

Automated Software Engineering 2006
Motivation

• Unit testing validates individual program units
  – Hard to build correct systems from broken units
• Unit testing is used in practice
  – 79% of Microsoft developers use unit testing
    [Venolia et al., MSR TR 2005]
  – Code for testing often larger than project code
    • Microsoft [Tillmann and Schulte, FSE 2005]
    • Eclipse [Danny Dig, Eclipse project contributor]
Focus: Object-oriented unit testing

- **Unit** is one class or a set of classes
- **Example** [Stotts et al. 2002, Csallner and Smaragdakis 2004, …]

```java
// class under test
class UBStack {
    public UBStack() {...}
    public void push(int k) {...}
    public void pop() {...}
    public int top() {...}
    public boolean equals(UBStack s) {
        ...
    }
}
```

```java
// example unit test case
void test_push_equals() {
    UBStack s1 = new UBStack();
    s1.push(1);
    UBStack s2 = new UBStack();
    s2.push(1);
    assert(s1.equals(s2));
}
```
Unit test case = Test input + Oracle

- **Test Input**
  - Sequence of method calls on the unit
  - Example: sequence of `push`, `pop`

- **Oracle**
  - Procedure to compare actual and expected results
  - Example: `assert`
Creating test cases

- **Automation** requires addressing both:
  - Test input generation
  - Test classification
  - Oracle from user: rarely provided in practice
  - No oracle from user: users manually inspect generated test inputs
    - Tool uses an approximate oracle to reduce manual inspection

- Manual creation is tedious and error-prone
  - Delivers incomplete test suites
Problem statement

• Compare *automated* unit testing techniques by effectiveness in finding faults
Outline

• Motivation, background and problem
• Framework and existing techniques
• New technique
• Evaluation
• Conclusions
A general framework for automation

Formal specification
A general framework for automation

Program

class UBStack{
... push(int k){...}
pop(){...}
equals(UBStack s){...}
}

Test suite

daios [Ernst et al., 2001]

test_push.equals(
) {
...
}

test0() {
pop(); push(0);
}
test1() {
push(1);
pop();
}

Unit testing tool

Daikon [Ernst et al., 2001]

Test suite generator

Candidate inputs

test0() {
pop(); push(0);
}
test1() {
push(1);
pop();
}

But formal specifications are rarely available

Model of correct operation

class UBStack
//@ invariant
//@ size >= 0

Fault-revealing test inputs

False alarm

True fault
Reduction to improve quality of output

Model of correct operation

Fault-revealing test inputs

Classifier

Reducer

Candidate inputs

(subset of) Fault-revealing test inputs

True fault

False alarm
Combining generation and classification

<table>
<thead>
<tr>
<th>generation</th>
<th>classification</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Random (RanGen)</td>
<td>Uncaught exceptions (UncEx)</td>
<td>Operational models (OpMod)</td>
</tr>
<tr>
<td>Symbolic (SymGen)</td>
<td>[ Csallner and Smaragdakis, SPE 2004 ], …</td>
<td>[ Pacheco and Ernst, ECOOP 2005 ]</td>
</tr>
<tr>
<td></td>
<td>[ Xie et al., TACAS 2005 ]</td>
<td>?</td>
</tr>
</tbody>
</table>
Random Generation

• Chooses sequence of methods at random
• Chooses arguments for methods at random
Instantiation 1: \textbf{RanGen + UncEx}

Program \rightarrow Random generation \rightarrow Uncaught exceptions

Uncaught exceptions \rightarrow Candidate inputs

Candidate inputs \rightarrow Fault-revealing test inputs

Fault-revealing test inputs \rightarrow \text{True fault} \rightarrow \text{False alarm}
Instantiation 2: **RanGen** + **OpMod**

1. **Program** → **Model generator**
2. **Test suite** → **Model generator**
3. **Model of correct operation** → **Fault-revealing test inputs**
4. **Operational models** → **Random generation**
5. **Random generation** → **Candidate inputs**
6. **Fault-revealing test inputs** → **True fault**
7. **Fault-revealing test inputs** → **False alarm**
Symbolic Generation

• Symbolic execution
  – Executes methods with symbolic arguments
  – Collects constraints on these arguments
  – Solves constraints to produce concrete test inputs

• Previous work for OO unit testing
  [Xie et al., TACAS 2005]
  – Basics of symbolic execution for OO programs
  – Exploration of method sequences
Instantiation 3: SymGen + UncEx

- Program → Symbolic generation → Uncaught exceptions → Fault-revealing test inputs
- Candidate inputs
- True fault, False alarm
Outline

• Motivation, background and problem
• Framework and existing techniques
• New technique
• Evaluation
• Conclusions
Proposed new technique

• Model-based Symbolic Testing
  \textit{(SymGen+OpMod)}
  – Symbolic generation
  – Operational model classification

• Brief comparison with existing techniques
  – May explore failing method sequences that \textit{RanGen+OpMod} misses
  – May find semantic faults that \textit{SymGen+UncEx} misses
Contributions

• Extended symbolic execution
  – Operational models
  – Non-primitive arguments

• Implementation (Symclat)
  – Modified explicit-state model-checker
    Java Pathfinder [Visser et al., ASE 2000]
**Instantiation 4: SymGen + OpMod**

1. **Program** → **Symmetric generation**
2. **Model generator**
3. **Model of correct operation**
4. **Fault-revealing test inputs**
5. **Candidate inputs**
6. **True fault** → **False alarm**
Outline

• Motivation, background and problem
• Framework and existing techniques
• New technique
• Evaluation
• Conclusions
## Evaluation

- **Comparison of four techniques**

<table>
<thead>
<tr>
<th>generation</th>
<th>classification</th>
<th>Implementation tool</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>RanGen+ UncEx</td>
<td>RanGen+ OpMod</td>
</tr>
<tr>
<td>Symbolic</td>
<td>SymGen+ UncEx</td>
<td>SymGen+ OpMod</td>
</tr>
</tbody>
</table>

- Eclat [Pacheco and Ernst, 2005]
- Symclat
# Subjects

<table>
<thead>
<tr>
<th>Source</th>
<th>Subject</th>
<th>NCNB LOC</th>
<th>#methods</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>UBStack 12</td>
<td>88</td>
<td>11</td>
</tr>
<tr>
<td>Daikon [Ernst et al. 2001]</td>
<td>UtilMDE</td>
<td>1832</td>
<td>69</td>
</tr>
<tr>
<td>DataStructures [Weiss 99]</td>
<td>BinarySearchTree</td>
<td>186</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>StackAr</td>
<td>90</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>StackLi</td>
<td>88</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>IntegerSetAsHashSet</td>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Meter</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>DLLList</td>
<td>286</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>E_OneWayList</td>
<td>171</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>E_SLLList</td>
<td>175</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>OneWayList</td>
<td>88</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>OneWayNode</td>
<td>65</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>SLLList</td>
<td>92</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td>TwoWayList</td>
<td>175</td>
<td>9</td>
</tr>
<tr>
<td>JML samples [Cheon et al.2002]</td>
<td>RatPoly (46 versions)</td>
<td>582.51</td>
<td>17.20</td>
</tr>
<tr>
<td>MIT 6.170 problem set [Pacheco and Ernst, 2005]</td>
<td>RatPoly (46 versions)</td>
<td>582.51</td>
<td>17.20</td>
</tr>
</tbody>
</table>
Experimental setup

- Eclat (RanGen) and Symclat (SymGen) tools
  - With UncEx and OpMod classifications
  - With and without reduction
- Each tool run for about the same time (2 min. on Intel Xeon 2.8GHz, 2GB RAM)
- For RanGen, Eclat runs each experiment with 10 different seeds
Comparison metrics

- Compare effectiveness of various techniques in finding faults
- Each run gives to user a set of test inputs
  - Tests: Number of test inputs given to user
- Metrics
  - Faults: Number of actually fault-revealing test inputs
  - $\text{DistinctF}$: Number of distinct faults found
  - $\text{Prec} = \text{Faults}/\text{Tests}$: Precision, ratio of generated test inputs revealing actual faults
Evaluation procedure

Unit testing tool → Tests → JML formal spec

Faults

DistinctF

Prec = Faults/Tests
Summary of results

• All techniques miss faults and report false positives
• Techniques are complementary
• \textit{RanGen} is sensitive to seeds
• Reduction can increase precision but decreases number of distinct faults
False positives and negatives

• Generation techniques can miss faults
  – RanGen can miss important sequences or input values
  – SymGen can miss important sequences or be unable to solve constraints

• Classification techniques can miss faults and report false alarms due to imprecise models
  – Misclassify test inputs (normal as fault-revealing or fault-revealing as normal)
# Results without reduction

<table>
<thead>
<tr>
<th></th>
<th>RanGen+ UncEx</th>
<th>RanGen+ OpMod</th>
<th>SymGen+ UncEx</th>
<th>SymGen+ OpMod</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>4,367.5</td>
<td>1,666.6</td>
<td>6,676</td>
<td>4,828</td>
</tr>
<tr>
<td>Faults</td>
<td>256.0</td>
<td>181.2</td>
<td>515</td>
<td>164</td>
</tr>
<tr>
<td>DistinctF</td>
<td>17.7</td>
<td>13.1</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Prec</td>
<td>0.20</td>
<td>0.42</td>
<td>0.15</td>
<td>0.14</td>
</tr>
</tbody>
</table>

# of test inputs given to the user
# of actual fault-revealing tests generated

# distinct actual faults

precision = Faults / Tests
Results with reduction

<table>
<thead>
<tr>
<th></th>
<th>RanGen+ UncEx</th>
<th>RanGen+ OpMod</th>
<th>SymGen+ UncEx</th>
<th>SymGen+ OpMod</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tests</td>
<td>124.4</td>
<td>56.2</td>
<td>106</td>
<td>46</td>
</tr>
<tr>
<td>Faults</td>
<td>22.8</td>
<td>13.4</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>DistinctF</td>
<td>15.3</td>
<td>11.6</td>
<td>11</td>
<td>7</td>
</tr>
<tr>
<td>Prec</td>
<td>0.31</td>
<td>0.51</td>
<td>0.17</td>
<td>0.20</td>
</tr>
</tbody>
</table>

- DistinctF ↓ and Prec ↑
  - Reduction misses faults: may remove a true fault and keep false alarm
  - Redundancy of tests decreases precision
Sensitivity to random seeds

• For one RatPoly implementation

\[
\begin{array}{|c|c|c|}
\hline
& \text{RanGen+ UncEx} & \text{RanGen+ OpMod} \\
\hline
\text{Tests} & 17.1 & 20 \\
\hline
\text{Faults} & 0.2 & 0.8 \\
\hline
\text{DistinctF} & 0.2 & 0.5 \\
\hline
\text{Prec} & 0.01 & 0.04 \\
\hline
\end{array}
\]

• \textbf{RanGen+OpMod (with reduction)}
  – 200 tests for 10 seeds 8 revealing faults
  – For only 5 seeds there is (at least) one test that reveals fault
Outline

• Motivation, background and problem
• Framework and existing techniques
• New technique
• Evaluation
• Conclusions
Key: Complementary techniques

• Each technique finds some fault that other techniques miss

• Suggestions
  – Try several techniques on the same subject
    • Evaluate how merging independently generated sets of test inputs affects Faults, DistinctF, and Prec
    • Evaluate other techniques (e.g., RanGen+SymGen [Godefroid et al. 2005, Cadar and Engler 2005, Sen et al. 2005])
  – Improve RanGen
    • Bias selection (What methods and values to favor?)
    • Run with multiple seeds (Merging of test inputs?)
Conclusions

• Proposed a new technique: Model-based Symbolic Testing
• Compared four techniques that combine
  – Random vs. symbolic generation
  – Uncaught exception vs. operational models classification
• Techniques are complementary
• Proposed improvements for techniques