MTAT.03.159: Software Testing

Lecture 04: White-Box Testing (advanced) Part 1

Spring 2018

Dietmar Pfahl
e-mail: dietmar.pfahl@ut.ee
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews
White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews
Data Flow Testing – Motivation

• Middle ground in structural testing
  – Node (=statement) and edge (=branch) coverage don’t test interactions between statements
  – All-path testing is infeasible
  – Need a coverage criterion that is stronger than branch testing but feasible

• Intuition: Statements interact through data flow
  – Value computed in one statement, used in another
  – Bad value computation revealed only when it is used
Data Flow Testing

• Identifies paths in the program that go
  – from the **assignment** of a value to a variable to
  – the **use** of such variable,
  to make sure that the variable is properly used.

\[ X \leftarrow 14; \ldots \ Y \leftarrow X-3; \]
Data Flow Testing – Definitions

- **Def** – assigned or changed
- **Uses** – utilized (not changed)
  - **C-use** (Computation) e.g. right-hand side of an assignment, an index of an array, parameter of a function.
  - **P-use** (Predicate) branching the execution flow, e.g. in an if statement, while statement, for statement.

```plaintext
[0] bool AccClient(int age;
  gtype gender)
[1] bool accept = false
[2] if (gender==female & age<85)
[3]    accept = true;
[4] if (gender==male & age<80)
[5]    accept = true;
[6] return accept
```

\[\text{Def}(0/1) = \{\text{age, gender, accept}\}\]
\[\text{P-use}(2) = \{\text{age, gender}\}\]
\[\text{P-use}(4) = \{\text{age, gender}\}\]
\[\text{C-use}(6) = \{\text{accept}\}\]

\[\text{Def}(3) = \{\text{accept}\}\]
\[\text{Def}(5) = \{\text{accept}\}\]
Data Flow Testing – Criteria

• All **def-use paths**
  – requires that each simple (i.e., traversing a loop at most once) definition-clear path from a definition of a variable to its use is executed

• All **uses paths**
  – requires that for each definition-use pair of a variable at least one simple definition-clear path is executed

• All **definitions paths**
  – requires that at least one path from the definition of a variable to one of its uses is executed

• ...
Data Flow Testing – Example

Considering age, there are two DU pairs:

(a) [0]-[2]

(b) [0]-[4]

Test case for ‘all-uses’:
AccClient(*, *)-> *

```c
[0] bool AccClient(int age;
                  gtype gender)
[1] bool accept = false
[2] if (gender==female & age<85)
    accept = true;
[3] if (gender==male & age<80)
    accept = true;
[4] return accept
```

Test cases needed to satisfy all-uses-paths criterion:
AccClient() is executed
Data Flow Testing – Example

Considering gender, there are two DU pairs:

(a) [0]-[2]
(b) [0]-[4]

Test case for ‘all-uses’:
AccClient(*, *)-> *
Data Flow Testing – Example

Considering gender, there are two DU pairs with three def-use paths:

(a) [0]-[2]: 0-1-2
(b) [0]-[4]: 0-1-2-4, 0-1-2-3-4

Test cases for ‘all-def-uses’:
AccClient(f, 83) -> true
AccClient(m, *) -> *

```java
[0] bool AccClient(int age;
    gtype gender)
[1] bool accept = false
[2] if (gender==female & age<85)
[3] accept = true;
[4] if (gender==male & age<80)
[5] accept = true;
[6] return accept
```
Data Flow Testing – Example

Considering `accept`, there are three DU pairs:
(a) [1]-[6] (b) [3]-[6] (c) [5]-[6]

Test cases for ‘all-uses’:
(a) `AccClient(*, 85)` -> false
(b) `AccClient(f, 80)` -> true
(c) `AccClient(m, 79)` -> true
Data Flow Criteria

All c-uses  All defs  All p-uses

All c-uses, some p-uses  All p-uses, some c-uses

All uses

All def-use paths

Weaker  Stronger

# tests
Data Flow Criteria

- All c-uses
- All defs
- All p-uses

All uses
- All def-use paths
- All paths

Weak
er

Stronger

# tests

All branches
Effectiveness of Control-Flow & Data-Flow Test Criteria

**TABLE I. SYSTEMS UNDER TEST**

<table>
<thead>
<tr>
<th>Program</th>
<th>KLOC</th>
<th>Test KLOC</th>
<th>Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>96</td>
<td>50</td>
<td>2,205</td>
</tr>
<tr>
<td>Closure</td>
<td>90</td>
<td>83</td>
<td>7,927</td>
</tr>
<tr>
<td>Math</td>
<td>85</td>
<td>19</td>
<td>3,602</td>
</tr>
<tr>
<td>Time</td>
<td>28</td>
<td>53</td>
<td>4,130</td>
</tr>
<tr>
<td>Lang</td>
<td>22</td>
<td>6</td>
<td>2,245</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>321</strong></td>
<td><strong>211</strong></td>
<td><strong>20,109</strong></td>
</tr>
</tbody>
</table>

Source:
# Effectiveness of Control-Flow & Data-Flow Test Criteria

<table>
<thead>
<tr>
<th>% of detected faults</th>
<th>Statement</th>
<th>Branch</th>
<th>MC/DC</th>
<th>Loop</th>
<th>All Control-flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>32%</td>
<td>5%</td>
<td>10%</td>
<td>0%</td>
<td>14%</td>
<td>10%</td>
</tr>
<tr>
<td>32%</td>
<td>18%</td>
<td>18%</td>
<td>11%</td>
<td>18%</td>
<td>19%</td>
</tr>
<tr>
<td>24%</td>
<td>18%</td>
<td>18%</td>
<td>11%</td>
<td>25%</td>
<td>19%</td>
</tr>
<tr>
<td>12%</td>
<td>5%</td>
<td>18%</td>
<td>0%</td>
<td>8%</td>
<td>8%</td>
</tr>
<tr>
<td><strong>44%</strong></td>
<td><strong>24%</strong></td>
<td><strong>33%</strong></td>
<td><strong>15%</strong></td>
<td><strong>29%</strong></td>
<td><strong>28%</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># of undetected faults by control flow criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>% of detected faults</th>
<th>def-use (DUA)</th>
<th>Data &amp; control-flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>86%</td>
<td>87%</td>
<td>92%</td>
</tr>
<tr>
<td>87%</td>
<td>80%</td>
<td>90%</td>
</tr>
<tr>
<td>80%</td>
<td>91%</td>
<td>87%</td>
</tr>
<tr>
<td>91%</td>
<td>50%</td>
<td>92%</td>
</tr>
<tr>
<td>50%</td>
<td>79%</td>
<td>65%</td>
</tr>
<tr>
<td><strong>86%</strong></td>
<td><strong>87%</strong></td>
<td><strong>92%</strong></td>
</tr>
</tbody>
</table>

White-Box Testing Techniques

- Control-Flow Testing
- Data-Flow Testing
- Mutation Testing
- Symbolic Execution
- Static Code Analysis
- Reviews
Mutation Testing
(Fault-Based Testing)

Original Program

Fault Introduction

Mutant Program

Test Cases Applied to Both Original & Mutant Program

Output is compared. If results for original and mutant program are same, mutant is KILLED.
Assessing Test Suite Quality

• Idea
  – I make $n$ copies of my program, each copy with a known number $m_n$ of (unique) faults
  – Assume introduced faults are exactly like real faults in every way
  – I run my test suite on the programs with seeded faults ...
    • ... and the tests reveal 20% of the introduced faults

• What can I infer about my test suite?
Mutation Testing Procedure

1. Take a program and test data generated for that program
2. Create a number of *similar* programs (mutants), each differing from the original in a small way
3. The original test data are then run through the *mutants*
4. If tests detect all changes in mutants, then the mutants are dead and the test suite adequate
   Otherwise: Create more test cases and iterate 2-4 until a sufficiently high number of mutants is killed
Mutation Testing – Terminology

- **Mutant** – new version of the program with a small deviation (=fault) from the original version
- **Killed** mutant – new version detected by the test suite
- **Live** mutant – new version *not* detected by the test suite
Examples of Mutation Operations

- Change relational operator (\(<\), \(\geq\), …)
- Change logical operator (\(\lor\), \&\, …)
- Change arithmetic operator (\(*\), \(+\), \(-\), …)
- Change constant name / value
- Change variable name / initialisation
- Change (or even delete) statement
- …

more examples

http://pitest.org/quickstart/mutators/
Example Mutants

\[
\begin{align*}
\text{if} \ (a \ || \ b) & \quad \text{if} \ (a \ && b) \\
\quad & \quad \\
\quad c = a + b; & \quad c = a + b; \\
\text{else} & \quad \text{else} \\
\quad & \quad \\
\quad c = 0; & \quad c = 0;
\end{align*}
\]
Types of Mutants

Not interesting:

- **Stillborn mutants**: Syntactically incorrect – killed by compiler, e.g., \( x = a ++ b \)
- **Trivial mutants**: Killed by almost any test case
- **Equivalent mutant**: Always acts in the same behaviour as the original program, e.g., \( x = a + b \) and \( x = a - (-b) \)

Those mutants are interesting which behave differently than the original program, and we do not (yet) have test cases to identify them.
Equivalent Mutants

\[
\text{if} \ (a == 2 \ \&\& \ b == 2) \\
\quad c = a + b; \\
\text{else} \\
\quad c = 0;
\]

\[
\text{if} \ (a == 2 \ \&\& \ b == 2) \\
\quad c = a \ast b; \\
\text{else} \\
\quad c = 0;
\]

\[
\text{int} \ \text{index}=0; \\
\text{while} \ (\ldots) \\
\{ \\
\quad \ldots; \\
\quad \text{index}++; \\
\quad \text{if} \ (\text{index}==10) \\
\quad \quad \text{break;}
\}
\]

\[
\text{int} \ \text{index}=0; \\
\text{while} \ (\ldots) \\
\{ \\
\quad \ldots; \\
\quad \text{index}++; \\
\quad \text{if} \ (\text{index}>=10) \\
\quad \quad \text{break;}
\}
\]
Program Example

nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Program returns the index of the array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Program Example

```java
nbrs = new int[range]
public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] > a[imax]
            imax:= i;
    return imax;
}
```

Program returns the index of the array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Variable Name Mutant

nbrs = new int[range]

public int max(int[] a) {
    intimax := 0;
    for (int i = 1; i <= range; i++)
        if i > a[imax]
            imax:= i;
    return imax;
}

Program returns the index of the (first) array element with the maximum value.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>
Relational Operator Mutant

```java
nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 1; i <= range; i++)
        if a[i] >= a[imax]
            imax := i;
    return imax;
}
```

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TC1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>TC2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>TC3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

Need a test case with two identical max entries in a[.], e.g., (1, 3, 3)
Variable Operator Mutant

nbrs = new int[range]

public int max(int[] a) {
    int imax := 0;
    for (int i = 0; i < range; i++)
        if a[i] > a[imax]
            imax := i;
    return imax;
}

Need a test case detecting wrong loop counting
Mutation Testing
Assumptions

• Competent programmer hypothesis:
  – Programs are nearly correct
    • Real faults are small variations from the correct program
    • => Mutants are reasonable models of real faulty programs

• Coupling effect hypothesis:
  – Tests that find simple faults also find more complex faults
    • Even if mutants are not perfect representatives of real faults, a test suite that kills mutants is good at finding real faults too
Mutation Testing Tool: PIT

Real world mutation testing

PIT is a state of the art mutation testing system, providing gold standard test coverage for Java and the jvm. It's fast, scalable and integrates with modern test and build tooling.

Get Started

User Group  Issues  Source  Maven Central
## Default Mutation Operators in PIT

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conditionals Boundary</td>
<td>Replaces relational operators with their boundary counterpart (e.g., <code>&lt;</code> becomes <code>&lt;=</code>, <code>&gt;=</code> becomes <code>&gt;</code>, etc.).</td>
</tr>
<tr>
<td>Negate Conditionals</td>
<td>Replaces all conditionals with their negated counterpart (e.g., <code>==</code> becomes <code>!=</code>, <code>&lt;</code> becomes <code>&gt;=</code>, etc.).</td>
</tr>
<tr>
<td>Math</td>
<td>Replaces binary arithmetic operations from either integer or floating-point arithmetic with another operation (e.g., <code>+</code> becomes <code>−</code>, <code>*</code> becomes <code>/</code>, etc.).</td>
</tr>
<tr>
<td>Increments</td>
<td>Replaces increments of local variables with decrements and vice versa.</td>
</tr>
<tr>
<td>Invert Negatives</td>
<td>Inverts the negation of integer and floating point numbers.</td>
</tr>
<tr>
<td>Return Values</td>
<td>Changes the return value of a method depending on the return type (e.g., non-null return values are replaced with <code>null</code>, integer return values are replaced with <code>0</code>, etc.).</td>
</tr>
<tr>
<td>Void Method Call</td>
<td>Removes method calls to void methods.</td>
</tr>
</tbody>
</table>
Lab 4 – Mutation Testing

• Part 1 – Code Defenders Game (during lab)

• Part 2 – Lab 4 Assignment (started in lab and completed at home)

http://code-defenders.org
Lab 4 – Mutation Testing

Lab 4 (week 31: Apr 03 - Apr 04) - Mutation Testing (10%)

Lab 4 Instructions & Tools

Submission Deadlines:
• Tuesday Labs: Monday, 09 Apr, 23:59
• Wednesday Labs: Tuesday, 10 Apr, 23:59

• Penalties apply for late delivery: 50% penalty, if submitted up to 24 hours late; 100 penalty, if submitted more than 24 hours late
Lab 4 – Mutation Testing (cont’d)

Mutation Testing: Run tests, kill mutants
Add tests, kill more mutants, detect faults

Report:
Detected faults
Mutation coverage
Code coverage

PIT

SUT: Minimum Binary Heap (incl. Test code)

Improved Test Suite

Mutants
Next Weeks

• Lab 4:
  – Mutation Testing

• Lecture 5:
  – Test Lifecycle, Test Levels, Test Tools