Seeding Bugs to Find Bugs: Mutation Testing Revisited

How do you know your test suite is "good enough"? One of the best ways to tell is mutation testing. Mutation testing seeds artificial defects (mutations) into a program and checks whether your test suite finds them. If it does not, this means your test suite is not adequate yet.

Despite its effectiveness, mutation testing has two issues. First, it requires large computing resources to re-run the tests again and again. Second, it can be time-consuming and resource-intensive.

Hans-Peter is moving into this building – actually, he built it, too. He’s worried that everything might be okay. But he’s not that worried.
If you’re building not a building, but a piece of software, you have many more reasons to be worried.

It’s not like this is the ultimate horror…

…but still, this question causes fear, uncertainty and doubt in managers.
Testing

• You want your program to be well tested
• How do we measure “well tested”?

While the Program is executed, one statement (or basic block) after the other is covered — i.e., executed at least once — but not all of them. Here, the input is “test”; checkmarks indicate executed blocks.
We’d like to test every statement, so we come up with more test cases.

This is an interesting boundary test case, as it may cause non-deterministic behavior. Can you see why?
Coverage Criteria

Statement testing

Branch testing

MC/DC testing

Compound condition testing

Path testing

Boundary interior testing

LCSAJ testing

Loop boundary testing

Basic condition testing

And this is the summary of structural testing techniques.

Established by a number of studies done by E. Weyuker at AT&T. “Any explicit relationship between coverage and error detection would mean that we have a fixed distribution of errors over all statements and paths, which is clearly not the case”.

Weyuker’s Hypothesis

The adequacy of a coverage criterion can only be intuitively defined.

Mutation Testing

DeMillo, Lipton, Sayward 1978
A Mutation

class Checker {
    public int compareTo(Object other) {
        return 1;
    }
}

not found by AspectJ test suite

Mutation Operators

<table>
<thead>
<tr>
<th>op</th>
<th>description</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>cpr</td>
<td>constant for constant replacement</td>
<td>C \ne C'</td>
</tr>
<tr>
<td>scr</td>
<td>scalar for constant replacement</td>
<td>C \ne X</td>
</tr>
<tr>
<td>acr</td>
<td>array for constant replacement</td>
<td>C \ne A[I]</td>
</tr>
<tr>
<td>scr</td>
<td>struct for constant replacement</td>
<td>C \ne S</td>
</tr>
<tr>
<td>csr</td>
<td>constant for scalar variable replacement</td>
<td>X \ne C</td>
</tr>
<tr>
<td>svr</td>
<td>scalar variable replacement</td>
<td>X \ne Y</td>
</tr>
<tr>
<td>asr</td>
<td>array for scalar variable replacement</td>
<td>A[I] \ne C</td>
</tr>
<tr>
<td>ssr</td>
<td>struct for scalar replacement</td>
<td>X \ne S</td>
</tr>
</tbody>
</table>

Expression Modifications

<table>
<thead>
<tr>
<th>op</th>
<th>description</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>absolute value insertion</td>
<td>e &lt; 0</td>
</tr>
<tr>
<td>aor</td>
<td>arithmetic operator replacement</td>
<td>C \ne C'</td>
</tr>
<tr>
<td>lcr</td>
<td>logical connector replacement</td>
<td>C \ne X</td>
</tr>
<tr>
<td>ror</td>
<td>relational operator replacement</td>
<td>C \ne X</td>
</tr>
<tr>
<td>uoi</td>
<td>unary operator insertion</td>
<td>C \ne X</td>
</tr>
</tbody>
</table>

Statement Modifications

<table>
<thead>
<tr>
<th>op</th>
<th>description</th>
<th>constraint</th>
</tr>
</thead>
<tbody>
<tr>
<td>sdl</td>
<td>statement deletion</td>
<td></td>
</tr>
<tr>
<td>sca</td>
<td>switch case replacement</td>
<td></td>
</tr>
<tr>
<td>ses</td>
<td>end block shift</td>
<td></td>
</tr>
</tbody>
</table>

from Pezze + Young, “Software Testing and Analysis”, Chapter 16
If one ever needed a proof that testing is a destructive process — here it is

Does it work?

- Generated mutants are similar to real faults
  Andrews, Briand, Labiche, ICSE 2005
- Mutation testing is more powerful than statement or branch coverage
  Walsh, PhD thesis, State University of NY at Binghampton, 1985
- Mutation testing is superior to data flow coverage criteria
Efficiency

- Test suite must be re-run for every single mutation
- Expensive

How do we make mutation testing efficient?

Efficiency

- Manipulate byte code directly rather than recompiling every single mutant
- Focus on few mutation operators
  - replace numerical constant \( C \) by \( C + 1 \), or 0
  - negate branch condition
  - replace arithmetic operator (+ by –, \( \ast \) by \( / \), etc.)
- Use mutant schemata
  individual mutants are guarded by run-time conditions
- Use coverage data
  only run those tests that actually execute mutated code
A Mutation

class Checker {
    public int compareTo(Object other) {
        return 1;
    }
}

not found by AspectJ test suite because it is not executed

– and since we know it’s not executed, we don’t even apply this mutation.

Efficiency

Inspection

Inspection

• A mutation may leave program semantics unchanged
• These equivalent mutants must be determined manually
• This task is tedious.
An Equivalent Mutant

```java
public int compareTo(Object other) {
    if (!(other instanceof BcelAdvice))
        return 0;
    BcelAdvice o = (BcelAdvice)other;
    if (kind.getPrecedence() != o.kind.getPrecedence()) {
        if (kind.getPrecedence() > o.kind.getPrecedence())
            return +1;
        else
            return -1;
        }
    // More comparisons...
}
```

Frankl’s Observation

We also observed that [...] mutation testing was costly. Even for these small subject programs, the human effort needed to check a large number of mutants for equivalence was almost prohibitive.


Inspection

How do we determine equivalent mutants?
Aiming for Impact

Measuring Impact

• How do we characterize “impact” on program execution?
• Idea: Look for changes in pre- and postconditions
• Use dynamic invariants to learn these

Dynamic Invariants
pioneered by Mike Ernst’s Daikon

Invariant
At f(), x is odd

Property
At f(), x = 2
Example

```java
public int ex1511(int[] b, int n)
{
    int s = 0;
    int i = 0;
    while (i != n) {
        s = s + b[i];
        i = i + 1;
    }
    return s;
}
```

- Run with 100 randomly generated arrays of length 7–13

Postcondition

```java
b[] = orig(b[])
return == sum(b)
```

Precondition

```java
n == size(b[])
b != null
n <= 13
n >= 7
```

Obtaining Invariants

- Run
- Get trace
- Filter invariants
- Report results

Impact on Invariants

```java
public ResultHolder signatureToStringInternal(String signature) {
    switch(signature.charAt(0)) {
        ...
        case 'L': { // Full class name
            int index = signature.indexOf(';');
            // Jump to the correct ';
            if (index != -1 && signature.length() > index + 1 &&
                signature.charAt(index + 1) == '>
            )
                index = index + 2;
            ...
            return new ResultHolder (signature.substring(1, index));
        }
    }
```
Impact on Invariants

Impact on Invariants

public ResultHolder signatureToStringInternal(String signature) {
switch (signature.charAt(0)) {
... case 'L': { // Full class name
    int index = signature.indexOf(';');
    // Jump to the correct ';
    if (index != -1 && signature.length() > index + 1 &&
        signature.charAt(index + 1) == '>
        index = index + 2;
    ...
    return new ResultHolder (signature.substring(1, index));
    }
}

impacts 40 invariants
but undetected by AspectJ unit tests

Javalanche
• Mutation Testing Framework for Java
  12 man-months of implementation effort
• Efficient Mutation Testing
  Manipulate byte code directly • Focus on few mutation operators • Use mutant schemata • Use coverage data
• Ranks Mutations by Impact
  Checks impact on dynamic invariants • Uses efficient invariant learner and checker
Mutation Testing
with Javalanche

1. Learn invariants from test suite
2. Insert invariant checkers into code
3. Detect impact of mutations
4. Select mutations with the most invariants violated (= the highest impact)

But does it work?
1. Are mutations that violate invariants useful?

2. Are mutations with the highest impact most useful?

3. Are mutants that violate invariants less likely to be equivalent?

### Evaluation Subjects

<table>
<thead>
<tr>
<th>Name</th>
<th>Lines of Code</th>
<th>#Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ Core</td>
<td>94,902</td>
<td>321</td>
</tr>
<tr>
<td>Barbecue</td>
<td>4,837</td>
<td>137</td>
</tr>
<tr>
<td>Commons</td>
<td>18,782</td>
<td>1,590</td>
</tr>
<tr>
<td>Jaxen</td>
<td>12,449</td>
<td>680</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>25,861</td>
<td>3,447</td>
</tr>
<tr>
<td>JT opas</td>
<td>2,031</td>
<td>128</td>
</tr>
<tr>
<td>XStream</td>
<td>14,480</td>
<td>838</td>
</tr>
</tbody>
</table>

### Mutations

<table>
<thead>
<tr>
<th>Name</th>
<th>#Mutations</th>
<th>%detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>AspectJ Core</td>
<td>47,146</td>
<td>53</td>
</tr>
<tr>
<td>Barbecue</td>
<td>17,178</td>
<td>67</td>
</tr>
<tr>
<td>Commons</td>
<td>15,125</td>
<td>83</td>
</tr>
<tr>
<td>Jaxen</td>
<td>6,712</td>
<td>61</td>
</tr>
<tr>
<td>Joda-Time</td>
<td>13,859</td>
<td>79</td>
</tr>
<tr>
<td>JT opas</td>
<td>1,533</td>
<td>72</td>
</tr>
<tr>
<td>XStream</td>
<td>5,186</td>
<td>92</td>
</tr>
</tbody>
</table>

% detected means covered mutations
Performance

- Learning invariants is very expensive
  22 CPU hours for AspectJ – one-time effort
- Creating checkers is somewhat expensive
  10 CPU hours for AspectJ – one-time effort
- Mutation testing is feasible in practice
  14 CPU hours for AspectJ, 6 CPU hours for XStream

Are mutations that violate invariants useful?

Results

Evaluation

1. Are mutations that violate invariants useful?

What is a “useful” mutation?

2. Are mutations with the highest impact most useful?

3. Are mutants that violate invariants less likely to be equivalent?

What is a “useful” mutation?
Useful Mutations

A technique for generating mutants is useful if most of the generated mutants are detected:

- less likely to be equivalent because detectable mutants = non-equivalent mutants
- close to real defects because the test suite is designed to catch real defects

Mutations we look for

<table>
<thead>
<tr>
<th></th>
<th>not violating invariants</th>
<th>violating invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>not detected by test suite</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>detected by test suite</td>
<td>?</td>
<td>!</td>
</tr>
</tbody>
</table>

Are mutations that violate invariants useful?

Mutations that violate invariants are more likely to be detected by actual tests – and thus likely to be useful.

All differences are statistically significant according to $\chi^2$ test
Are mutations with the highest impact *most useful?*

Mutations that violate several invariants are *most likely* to be detected by actual tests – and thus the *most useful.*

All differences are statistically significant according to \( \chi^2 \) test.

Detection Rates

Are mutants that violate invariants less likely to be equivalent?

- Randomly selected non-detected *Jaxen* mutants – 12 violating, 12 non-violating
- Manual inspection: *Are mutations equivalent?*
- Mutation was proven non-equivalent iff we could create a detecting test case
- Assessment took 30 minutes per mutation
Are mutants that violate invariants less likely to be equivalent?

In our sample, mutants that violated several invariants were significantly less likely to be equivalent.

Evaluation

1. Are mutations that violate invariants useful? ✔
2. Are mutations with the highest impact most useful? ✔
3. Are mutants that violate invariants less likely to be equivalent? ✔
Future Work

- How effective is mutation testing? on a large scale – compared to traditional coverage
- Predicting defects
  How does test quality impact product quality?
- Alternative impact measures
  Coverage • Program spectra • Method sequences
- Adaptive mutation testing
  Evolve mutations to have the fittest survive

Conclusion

Factor 6,666 – plus full automation due to lack of inspection