Software Mutational Robustness

A presentation by Rebecca Sousa
Presentation Overview

Introduction

Background

Technical Approach

Experimental Results

Applications

Discussion
Introduction

Terms and Definitions

- **Neutral mutation**: a random change to a program that still passes the test suite
- **Software mutational robustness** measures the fraction of neutral mutations
- Infinite number of ways to implement an algorithm in code
- Quicksort example:

```java
if (right > left) {
    // code elided ...
    quick(left, r);
    quick(l, right);
    quick(l, right);
}
```
Background

Biology

- Environmental and mutational robustness
- Neutral neighbors and neutral spaces

Evolutionary Computation

- Genetic programming (GP)

Software Engineering

- Mutation Testing
- N-Version Programming
Mutation Testing

Program Syntactic Space

- Specification (acceptable)
- Killed Mutants
- Neutral Mutants
- Equivalent Mutant
- Original Program
- Test Suite
Mutation Testing

 /*************************************************************************/
 * Spec (S):
 * Pre: parameter P is an array of three integer elements
 * Post: returns the smallest of the three input elements
 */

 int a(int p[]) {
     if (p[0] <= p[1] && p[0] <= p[2]) return p[0];
     else return p[2];
 }

 int b(int p[]) {
     sort(p, "ascending");
     return p[0];
 }
Technical Approach

- Program P
- Variant P'
- Mutation operators M (copy, swap, delete)
- Test suite T
- Finding: MutRB does not depend strongly on P or T

\[
MutRB(P, T, M) = \frac{\left| \{P' \mid m \in M. P' = m(P) \land T(P') = \text{true}\} \right|}{\left| \{P' \mid m \in M. P' = m(P)\} \right|}
\]
Representation and Operators

Representation

- Abstract syntax trees (AST)
- Low-level assembly code (ASM)

Operators

- Copy
- Delete
- Swap
Figure 3: Mutation operators: Copy, Delete, Swap.
Experimental Results

- 22 benchmark programs with test suites
  - Sorters
  - Siemens
  - Off-the-shelf

- **First order mutation**: apply a single random mutation to copy of program
- Want to rule out trivial mutations that produce equivalent assembly code
- 36.8% of variants continue to pass all test cases (?!)
- What is the cause of this?
  - Inadequate test suites (bad)
  - Semantically equivalent mutations (good)
Does robustness depend on test suite?

(a) Sorters

(b) Siemens

(c) Systems Programs

C  ASM 
# Taxonomy of Neutral Variants

<table>
<thead>
<tr>
<th>#</th>
<th>Functional Category</th>
<th>Frequency/35</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Different whitespace in output</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>Inconsequential change of internal variables</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Extra or redundant computation</td>
<td>6</td>
</tr>
<tr>
<td>4</td>
<td>Equivalent or redundant conditional guard</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>Switched to non-explicit return</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>Changed code is unreachable</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Removed optimization</td>
<td>1</td>
</tr>
</tbody>
</table>
Cumulative Robustness

(a) Program size not controlled.
(b) Program size controlled.
Multiple Languages

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>C++</th>
<th>Haskell</th>
<th>OCaml</th>
<th>Avg.</th>
<th>Std.Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble</td>
<td>25.7</td>
<td>28.2</td>
<td>27.6</td>
<td>16.7</td>
<td>24.6</td>
<td>5.3</td>
</tr>
<tr>
<td>insertion</td>
<td>26.0</td>
<td>42.0</td>
<td>35.6</td>
<td>23.7</td>
<td>31.8</td>
<td>8.5</td>
</tr>
<tr>
<td>merge</td>
<td>21.2</td>
<td>46.0</td>
<td>24.9</td>
<td>22.7</td>
<td>28.7</td>
<td>11.6</td>
</tr>
<tr>
<td>quick</td>
<td>25.5</td>
<td>42.0</td>
<td>26.3</td>
<td>11.4</td>
<td>26.3</td>
<td>12.5</td>
</tr>
</tbody>
</table>

| Avg.  | 24.6 | 39.5 | 28.6    | 18.6  | **27.9** | 3.1  |
| Std.Dev. | 2.3  | 7.8  | 4.8     | 5.7   | 3.1     |      |

Table 3: Mutational robustness of sorting algorithms at the assembly instruction level with 100% test suite coverage, for different algorithms and source language.
Application: Proactive Bug Repair

<table>
<thead>
<tr>
<th>Program</th>
<th>Fraction of Bugs Fixed</th>
<th>Bug Fixes</th>
</tr>
</thead>
<tbody>
<tr>
<td>bzip</td>
<td>2/5</td>
<td>63</td>
</tr>
<tr>
<td>imagemagick</td>
<td>2/5</td>
<td>8</td>
</tr>
<tr>
<td>jansson</td>
<td>2/5</td>
<td>40</td>
</tr>
<tr>
<td>leukocyte</td>
<td>1/5</td>
<td>1</td>
</tr>
<tr>
<td>lighttpd</td>
<td>1/5</td>
<td>73</td>
</tr>
<tr>
<td>nullhttpd</td>
<td>1/5</td>
<td>7</td>
</tr>
<tr>
<td>oggenc</td>
<td>0/5</td>
<td>0</td>
</tr>
<tr>
<td>potion</td>
<td>2/5</td>
<td>14</td>
</tr>
<tr>
<td>redis</td>
<td>0/5</td>
<td>0</td>
</tr>
<tr>
<td>tiff</td>
<td>0/5</td>
<td>0</td>
</tr>
<tr>
<td>vyquon</td>
<td>1/5</td>
<td>1</td>
</tr>
<tr>
<td>average</td>
<td>1.0/5</td>
<td>18.8</td>
</tr>
</tbody>
</table>
Application: N-Version Programming

- Want to develop N independent software instances
- Separate teams of human programmers likely to create similar programs - creating independence is hard!
- Solution: generate independent programs through neutral mutations
Discussion

Threats to Validity

● Choice of mutation operators
● Insufficient test suites

Further Investigation

● Landscape of neutral variants
● Markov Chain Monte Carlo

Applications to Software Engineering

● Optimization
● Automated program repair
● Mutation testing

Comparison to Biology

● Role of neutrality in evolution
Thank You!

Any questions?