Fault Localization

- Debugging software is an expensive and mostly manual process
- Of all debugging activities, locating the faults, or fault localization, is the most challenging one
- Approaches have been investigated to help automate fault localization

Typical Fault Localization Techniques

- Tarantula
- Set Union & Set Intersection
- Nearest Neighbor
- Cause Transitions

What Is the Fault in the Following Buggy Program?

```c
int mid(int x, int y, int z) {
    int m;
    m = z;
    if (y < z) {
        if (x < y) m = y;
        else if (x < z) m = y;  // should be m = x;
    } else {
        if (x > y) m = y;
        else if (x > z) m = x;
    }
    return m;
}
```

Tarantula: Coverage-based Fault Localization

<table>
<thead>
<tr>
<th>Statements</th>
<th>3,3,5</th>
<th>1,2,3</th>
<th>3,2,1</th>
<th>5,5,5</th>
<th>5,3,4</th>
<th>2,1,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>int m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = z;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (y &lt; z){</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (x &lt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (x &lt; z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>// should be x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>} else {</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>if (x &gt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>else if (x &gt; z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>m = x;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>return m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Approach

- Insight
  - Entities in a program that are primarily executed by failed test cases are more likely to be faulty than those that are primarily executed by passed test cases
- Solution
  - Ranking based on suspiciousness
    \[
    \text{Suspicious(s)} = \frac{\text{fail(s)}}{\text{fail(s)} + \text{totalfail} + \text{pass(s)} / \text{totalpass}}
    \]
Tarantula

<table>
<thead>
<tr>
<th>Statements</th>
<th>3.3.5</th>
<th>1.2.3</th>
<th>3.2.1</th>
<th>5.5.5</th>
<th>5.3.4</th>
<th>2.1.3</th>
<th>Susp</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int m;</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.63</td>
<td>0.5</td>
<td>0.5</td>
<td>0.71</td>
</tr>
<tr>
<td>m = x;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.03</td>
<td>0.03</td>
<td>0.03</td>
<td>0</td>
</tr>
<tr>
<td>if (x &lt; y)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m = y;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>else if (x &lt; y)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>) else {</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>if (x &gt; y)</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m = x;</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>return m;</td>
<td>0.5</td>
<td>0.5</td>
<td>0.5</td>
<td>0.63</td>
<td>0.63</td>
<td>0.63</td>
<td>1</td>
</tr>
</tbody>
</table>

Tarantula Display

Continuous Coloring

- Color of a statement is:
  \[
  \text{color(s)} = \text{low color (red)} + \frac{\% \text{ passed(s)}}{\% \text{ passed(s)} + \% \text{ failed(s)}} \times \text{color range}
  \]
- Brightness of a statement is:
  \[
  \text{bright(s)} = \max(\% \text{ passed(s)}, \% \text{ failed(s)})
  \]

Evaluation

- RQ1: How often does Tarantula color the faulty statements in a program red or in a reddish color?
- RQ2: How often does Tarantula color nonfaulty statements in a program red or in a reddish color?

Data Set

- Space program written in C
- 6218 LOC (executable)
- 13585 test cases
  - Generated test cases until the set contains at least 30 test cases that exercise nearly every statement and edge
  - Extracted 1000 randomly sized generated, near-decision-adequate test suites from this test pool

Single-fault Versions (20 versions)
Multiple fault (40 versions)

Slice-based Fault Localization

- A dynamic slice is the set of statements which do affect the value of the output.
- Dice: the set difference of two slices.
- dice (A - B) is effective to isolate bug b.

Formulas

- Set Union
  \[ E_{\text{initial}} = E_I - \bigcup_{p \in P} E_p \]
- Set Intersection
  \[ E_{\text{initial}} = \bigcap_{p \in P} E_p - E_I \]
- What is the insight behind each formula?

Set Union & Set Intersection 

Statements: 1, 1.5, 2, 2.1, 2.5, 3, 3.4, 2.1.3

Set Union & Set Intersection

- Nothing is found!
- y; //should be x
Nearest Neighbor [4]

- Spectra-based Fault Localization
  - Spectrum: profiling data that shows the number of times each program line is executed
  - Given a set of passing tests and a failing test F, find the passing test P, which has the most similar spectrum as F
  - Calculate the distance metric

Two Variants

- NN/perm
  - Frequency-marked statements
  - Sort statements based on frequency
  - Ulam edit distance
    - E.g., Dist([a, b, c, d], [a, c, d, b]) = 1 (move)
- NN/binary
  - 0-or-1 mark for each statement
  - No frequency is considered
  - Set subtraction is used to calculate distance

Nearest Neighbor

<table>
<thead>
<tr>
<th>Statements</th>
<th>1.3,5</th>
<th>3.5,5</th>
<th>5.3,4</th>
<th>5.3,5</th>
<th>5.3,3</th>
<th>2.2,5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>int m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>m = z;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>if (y + z) {</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>if (x &lt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>else if (x &lt; z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>m = y; //should be x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>} else {</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>if (x &gt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>else if (x &gt; z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>m = x;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>return m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

cause-transitions [2]

- Leverage delta debugging to isolate failure-inducing variable values at specific program locations
- Identify the transition points between different failure-inducing variable values
- Consider the transition points as bug locations

Definition

- Problem Statement
  - Yesterday, my program worked. Today, it does not. Why?
- Configuration: the set of all applied changes \( C = \{ \Delta_1, \Delta_2, ..., \Delta_n \} \)
  - \( c \subseteq C \) represents a subset of changes
- Test: the function \( c \rightarrow \{ X, V, ? \} \) to determine whether a configuration \( c \) leads to failure, success, or unresolved outcome of regression testing
How to Find the Minimum Failing-Inducing Changes?

- Naïve approach
  - Brute-force search: too expensive
- Efficient approach
  - Delta debugging: Binary search

Insight

- By finding the minimum set of changes whose application fails the test, Delta Debugging identifies bug-inducing changes

Search for Single Failure-Inducing Change

- Suppose there are 8 changes with the 7th is the cause. How do you use binary search to find it?

Conceptual Solution

<table>
<thead>
<tr>
<th>Step</th>
<th>Configuration</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 2 3 4</td>
<td>✔</td>
</tr>
<tr>
<td>2</td>
<td>5 6 7 8</td>
<td>✗</td>
</tr>
<tr>
<td>3</td>
<td>5 6</td>
<td>✔</td>
</tr>
<tr>
<td>4</td>
<td>7 8</td>
<td>✗</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
<td>✗</td>
</tr>
</tbody>
</table>

How Does DD Localize Failure-Inducing Variable Values?

Figure 2: Narrowing down state differences. By assessing whether a mixed state results in a passing (✓), a failing (✗), or an unresolved (?) outcome, Delta Debugging isolates a relevant difference.

Cause-Transitions

<table>
<thead>
<tr>
<th>Statements</th>
<th>1,2,3</th>
<th>2,3,1</th>
<th>5,5,5</th>
<th>5,3,4</th>
<th>2,1,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 int m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 if (y &lt; z) {</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 if (x &lt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 else if (x &lt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 m = y; //should be x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 ) else {</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9 if (x &gt; y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10 m = y;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11 else if (x &lt; z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 m = x;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 return m;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
How to do Fault Localization?

Statements 3,3,5 1,2,3 3,2,1 5,5,5 5,3,4 2,1,3

```c
int m;
if (y < z) {
    if (x < y) (y)
        m = y;
    else if (x < z) m = x;
} else {
    if (x > y) m = y;
    else if (x > z) m = x;
}
return m;
```

Pass Pass Pass Pass Pass
Fail

Step 5: Line 7: 3,3,3,1✗ Line 7 is the fault location!

Evaluation

- Siemens suite
  - 7 programs, 132 fault versions, 21,631 test suites designed to expose the faults
  - 122 versions are usable by the authors
  - Each version contains exactly one fault
  - Each fault may span multiple statements or even functions

Evaluation Method

- Basic Idea
  - Imagine an “ideal” debugger or a perfect programmer examines the ranked list of bug locations
  - The fewer locations/statements examined before the actual location, the higher score the report/tool gets
- Tarantula: go through the ranked list
- Other tools: PDG-based location examination

PDG-Based Evaluation Method [6]

- Given a reported location, do breadth-first search of Program Dependency Graph (PDG)
  - Terminate the search when a real fault is found
  - Score is proportional to the unexplored part of the PDG
  - Score near 1.0 means the No. 1 reported location is the correct one.

An Example [6]

12 total nodes in PDG

An Example [6]

Fault
An Example [6]

12 total nodes in PDG

Report + 1 Layer BFS

Real fault discovered

8 out of 12 nodes not covered by BFS: score = 8/12 ~ 0.67

12 total nodes in PDG

Report + 2 layers BFS
An Example [6]

Report + 3 layers BFS
12 total nodes in PDG

Fault

An Example [6]

Report + 4 layers BFS
STOP: Real fault discovered
12 total nodes in PDG

Fault

An Example [6]

Report + 4 layers BFS
12 total nodes in PDG
0 of 12 nodes not covered by BFS: score = 0/12 ~ 0.00.

Limitations [6]

• Isn’t a misleading report worse than an empty report?
• Nobody really searches a PDG like that!

Evaluation Results

Table 3: Average time expressed in seconds.

<table>
<thead>
<tr>
<th>Program</th>
<th>Tarantula (computation only)</th>
<th>Tarantula (including I/O)</th>
<th>Cleanup Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>print.tokens</td>
<td>0.0002</td>
<td>0.0337</td>
<td>250.3</td>
</tr>
<tr>
<td>print.tokens2</td>
<td>0.0037</td>
<td>0.0035</td>
<td>398.9</td>
</tr>
<tr>
<td>replace</td>
<td>0.0035</td>
<td>0.0032</td>
<td>189.3</td>
</tr>
<tr>
<td>schedule</td>
<td>0.0032</td>
<td>0.0030</td>
<td>7741.2</td>
</tr>
<tr>
<td>schedule2</td>
<td>0.0025</td>
<td>0.0023</td>
<td>186.8</td>
</tr>
<tr>
<td>trim</td>
<td>0.0023</td>
<td>0.0032</td>
<td>321.4</td>
</tr>
<tr>
<td>tot. info</td>
<td>0.0032</td>
<td>8.31</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2: Comparison of the effectiveness of each technique.
Reference