Overview

1. Introduction
   1.1. Fault Localization
   1.2. Test Case Purification
   1.3. Motivation
2. Test Case Purification for Fault Localization
   2.1. Framework
   2.2. Test Case Atomization
   2.3. Test Case Slicing
   2.4. Rank Refinement
3. Testing Test Case Purification for Fault Localization
   3.1. Implementation
   3.2. Results
4. Conclusion

Introduction

- Bugs in software are often times inevitable.
- Reported bugs require a developer’s time and effort in identifying its exact location in source code.
- Localizing bugs in large pieces of software can be a daunting task. Thus, methods for alleviating the work have been created.

Fault Localization - The act of localizing a bug or fault.
Introduction (Fault Localization)

- Fault localization
  - Consists of automatically ranking program entities based on an oracle of the bug, usually a failing test case.
  - Simply, how likely is a given function or statement associated with the given bug?
- Spectrum-based fault localization (A.K.A. coverage based fault localization)
  - Family of methods that use the execution trace of test cases (the coverage data) to measure the faultiness probabilities of program entities.
  - With these rankings, developers then manually examine the program to find the location of the bug.
  - We ask:
    - what parts of the code were covered?
    - Which part of the covered code has a higher chance of being faulty?

Introduction (Test Case Purification)

- Test Case Purification
  - The act of separating existing test cases into smaller fractions called purified cases.
- Goal
  - We would like to generate purified versions of test cases that
    - consist of only one assertion per test.
    - exclude unrelated statements of this assertion.
- Why?
  - Purified tests remove the complexity of a test by isolating assertions.
  - Makes it easier to understand what assertions are failing.

Terminology

- Test Oracle
  - determines the correctness of the software w.r.t. Its test input
- Assertion
  - a predicate that indicates the expected behavior of the program
- Program Entity
  - represents an analysis granularity (ex: Class, Method, Statement)
- Spectrum
  - set of program entities paired with execution flags.
  - These flags indicate whether the test case covers the entity.
Motivation

- Assertion fails. Abort.
- Lines after 21 not tested.
- Assertion at line 9 not tested.
- Tarantula ranks all 11 executed statements with the same suspiciousness.

- Split t1 into a1,a2,a3 (one assertion each)
  - a1 passes. a2,a3 fail.
  - Remove irrelevant statements from a2,a3 to create p2,p3.
  - p2,p3 both fail but both execute lines 20 and 21 twice. We can rank these as possible faulty statements.

Test Case Purification for Fault Localization

- Goal
  - Employ purified test cases to improve existing fault localization techniques.
- Framework
  - Test Case Atomization
    - Generates a set of single-assertion test cases for each failed test case.
  - Test Case Slicing
    - Removes the unrelated statements in all the failing single-assertion test cases
  - Rank Refinement
    - Combines the set of re-ranked purified test cases with an existing fault localization technique and sorts the statements according to rank as the final result.

Test Case Purification for Improving Fault Localization

- Test Case Atomization
  - Goal
    - Generate a set of purified test cases for each failing test.
  - Each original failing test case with k assertions is replaced with k purified test cases with single assertions.
  - No exceptions from the test cases should reach the framework if the assertion fails.
    - Exception handling.
      - Java: try { /* assertion */ } catch (Exception ..) { /* pass */ }
Test Case Atomization (continued)

1. Compile and execute all our single-assertion test cases.
2. Collect failing test cases.
   - For each, record its position that aborts the execution (A.K.A. broken statement)
3. Finally, each failing single-assertion test case along with its broken position are collected.

\[ \text{failed\_SA\_tests} = \{(\text{SA\_test}\_n : \text{broken\_statement}\_n) , (___ : ___), \ldots\} \]

Test Case Slicing

- **Goal**
  - To generate purified test cases, execute them, and collect their spectra.
  - Given a failing single-assertion test case from Test Case Atomization, slice this test case by removing irrelevant statements.
- **Static Slicing**
  - Keeps all the possible statements based on static data and control dependencies.
- **Dynamic Slicing**
  - Keeps the actually executed statements in the dynamic execution with dynamic data and control dependencies.
  - Leads to more removal of statements so we use this in our test case slicing.
  - Slicing Criterion = \( < b, V > \), where \( b \) is a statement in the object program and \( V \) is a set of variables to be observed at \( b \).

Test Case Slicing (continued)

1. Execute dynamic slicing on the single-assertion test cases. Collect statements that will be removed.
   - Slicing criterion = \( < b, V > \), where
     - \( b \) is our broken statements.
     - \( V \) is all of the variables at this statement.
2. Update each failing single-assertion test from atomization with a purified test case.
3. Execute these purified test cases. Record the spectra for the next phase.

Rank Refinement

- **Goal**
  - Re-rank the statements by an existing fault localization technique with the spectra produced in the phase of test case slicing.
  - Let \( S \) be a set of candidate statements.

\[
\text{ratio}(s) = \frac{|P_f(s)|}{|P_e(s)| + |P_u(s)|}
\]

where \( |P_f(s)| = \text{number of test cases covering } s \)
\( |P_e(s)| = \text{number of test cases not covering } s \)
\( |P_u(s)| = \text{number of test cases not covering } s \)

\[ \text{ratio}(s) = 0 \text{, for any } s \text{ not covered by a purified test case} \]
Rank Refinement (continued)

- The output of an existing fault localization method is the suspiciousness or faultiness values for all the candidate statements.
- Let $susp(s)$ be the suspiciousness value for a statement $s$.
- Normalize $susp(s)$ as 0 to 1 for all statements in $S$.

\[
norm(s) = \frac{susp(s) - \min(S)}{\max(S) - \min(S)}
\]

where $\max(S)$ denotes maximum score for all statements $\in S$
where $\min(S)$ denotes minimum score for all statements $\in S$

- Both $\text{ratio}(s)$ and $\text{norm}(s)$ is between 0 and 1.
- We Combine $\text{ratio}(s)$ and $\text{norm}(s)$ to refine ranking for each statement.
- The final score is between 0 and 1.
- Based on final scores of all statements, we re-rank the statements as the result of fault localization by test case purification.

The final score is:

\[
score(s) = \text{norm}(s) \times \left(1 + \frac{1}{\text{ratio}(s)}\right)
\]

Testing Test Case Purification in Fault Localization

- Xuan and Monperrus:
  - Tested with six existing Fault Localization techniques.
  - Created faulty versions of six open-source Java programs using mutation testing tools.
  - Selected 300 mutants from all the seeded faulty versions for each subject program as the final data set of faulty programs.
  - Executed original techniques, computed the suspiciousness values, collected the program spectra.
  - Compared original techniques with Test Case Purification technique.

Results

<table>
<thead>
<tr>
<th>Technique in comparison</th>
<th>Positive</th>
<th>Negative</th>
<th>Neutral</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td># Faults Percent</td>
<td># Faults Percent</td>
<td># Faults Percent</td>
</tr>
<tr>
<td>Tarantula</td>
<td>779</td>
<td>43.28</td>
<td>44</td>
</tr>
<tr>
<td>SBI</td>
<td>722</td>
<td>40.11</td>
<td>24</td>
</tr>
<tr>
<td>Ochiai</td>
<td>373</td>
<td>20.72</td>
<td>28</td>
</tr>
<tr>
<td>Jaccard</td>
<td>360</td>
<td>20.00</td>
<td>28</td>
</tr>
<tr>
<td>Ochiai2</td>
<td>330</td>
<td>18.33</td>
<td>28</td>
</tr>
<tr>
<td>Kuleczynski2</td>
<td>666</td>
<td>37.00</td>
<td>24</td>
</tr>
</tbody>
</table>

- Positive gives number of faults improved in localization using this technique.
- Negative gives number of faults when original techniques worked better.
- Neutral gives number of faults with no change.
Conclusion

- Applying Test Case Purification to existing fault localization techniques facilitates the act of finding bugs.
- Often times, results are up to 43% positive with the cost of about 2.4% worsened results.

Summary

- Collecting failed test cases from existing fault localization techniques.
- (Atomization) reducing to single-assertion test cases.
- (Slicing) creating purified tests and collect resulting spectra.
- (rank refinement) re-ranking statements for faultiness.
- Finding bugs more easily!

Questions?