Test-based Fault Localization in the Context of Formal Verification: Implementation and Evaluation of the Tarantula Algorithm in CPAchecker

Schindar Ali

Software and Computational Systems Lab
LMU Munich

02.09.2020
Agenda

1. Motivation

2. Background

3. Implementation

4. Evaluation

5. Future Work

6. Conclusion
Motivation
1.1 Motivation

1. Debugging software is an expensive and mostly manual process.

2. Of all debugging activities, locating the fault is the most challenging one.
1.1 Motivation

1. Debugging software is an expensive and mostly manual process.

2. Of all debugging activities, locating the fault is the most challenging one.
1.1 Motivation

There are two important concepts of checking whether the software contains bugs.

<table>
<thead>
<tr>
<th>Testing</th>
<th>Formal Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where we make sure our code - as written - actually works the way it’s supposed to work</td>
<td>Checking whether the software design satisfies some requirements (properties)</td>
</tr>
</tbody>
</table>
# 1.1 Motivation

There are two important concepts of checking whether the software contains bugs:

<table>
<thead>
<tr>
<th>Testing</th>
<th>Formal Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Where we make sure our code - as written - actually works the way it’s supposed to work</td>
<td>Checking whether the software design satisfies some requirements (properties)</td>
</tr>
</tbody>
</table>
1.2 Motivation - Goal

We want to check whether Test-Based Tarantula works better with Abstract reachability graph (ARG) than with test suites.
Background
2.1 Background - Tarantula

**Insight**
– Program elements that are executed by failed test cases/Counterexample are more likely to be faulty than those that are executed by **passed** test cases/Safe paths.

**Solution**
– Make ranking for the program by giving probability for each code line based on **suspiciousness**.

\[
\text{Suspicious}(s) = \frac{\text{fail}(s)}{\text{totalfail}} + \frac{\text{pass}(s)}{\text{totalpass}}
\]

\{ We need at least one fail(s) and one pass(s) to prevent divided by 0 \}
2.1.2 Background - Tarantula Example Process of using Tarantula

\[
\text{Suspicious(s)} = \frac{\text{fail(s)}}{\text{total fail}} - \frac{\text{fail(s)}}{\text{total fail} + \text{total pass}}
\]

<table>
<thead>
<tr>
<th>mid() {</th>
</tr>
</thead>
<tbody>
<tr>
<td>int x,y,z,m;</td>
</tr>
<tr>
<td>1: read(&quot;Enter 3 numbers:&quot;,x,y,z);</td>
</tr>
<tr>
<td>2: m = z;</td>
</tr>
<tr>
<td>3: if (y&lt;z)</td>
</tr>
<tr>
<td>4: if (x&lt;y)</td>
</tr>
<tr>
<td>5: m = y;</td>
</tr>
<tr>
<td>6: else if (x&lt;z)</td>
</tr>
<tr>
<td>7: m = y; // *** bug ***</td>
</tr>
<tr>
<td>8: else</td>
</tr>
<tr>
<td>9: if (x&gt;y)</td>
</tr>
<tr>
<td>10: m = y;</td>
</tr>
<tr>
<td>11: else if (x&gt;z)</td>
</tr>
<tr>
<td>12: m = x;</td>
</tr>
<tr>
<td>13: print(&quot;Middle number is:&quot;,m);</td>
</tr>
<tr>
<td>}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Test Cases</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>Pass/Fail Status</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>P</td>
<td>F</td>
</tr>
</tbody>
</table>

(1/1)/((1/1) + (5/5)) = 0.5

(1/1)/((1/1)+(1/5)) = 0.83
2.3 Background - DStar and Ochiai

In our Evaluation we compared Tarantula against:

1. **DStar Metric**

\[
\text{Suspicious}(s) = \frac{\text{Failed}(s)^{\delta}}{\text{Passed}(s) \times (\text{TotalFailed} - \text{Failed}(s))}
\]

We used \((\delta = 2)\), the most efficient value

2. **Ochiai Metric**

\[
\text{Suspicious}(s) = \frac{\text{Failed}(s)}{\sqrt{\text{TotalFailed} \times (\text{Failed}(s) + \text{Passed}(s))}}
\]

Needs at least only one failed(s)
Implementation
2.4.1 Implementation - Test-based Tarantula

How did we run Tarantula on test suites?

- Buggy Program
- Klee or VeriFuzz
- Test Suites
- TestCov
- Coverage Statistics
- Tarantula Ranking
How did we run Tarantula in CPAchecker?
2.4.3 Implementation - Formal-based Tarantula

How did we run Tarantula in CPAchecker?

- C-Program
- CFA
- ARG
- Algorithm
- CPA
- Tarantula
  - no bugs found
  - Candidate of suspicious program elements

ReachedSet
2.4.4.1 Tarantula on ARG - Example

1. Generating ARG

Example of ARG using Predicate Abstraction without merging the paths together

```c
int main() {
    char a = __VERIFIER_nondet_char();
    char b = __VERIFIER_nondet_char();
    char c = __VERIFIER_nondet_char();

    if (a == 'a' && b == 5 && c == 16) {
        ERROR:__VERIFIER_error();
    }
}
```
2.4.4.2 Tarantula on ARG - Example

2. Determine of Safe/fail paths

Example of ARG using Predicate Abstraction without meging the paths together
2.4.4.3 Tarantula on ARG - Example

3. Determine of Coverage for each CFAEdge

<table>
<thead>
<tr>
<th>CFAEdge</th>
<th>Coverage</th>
<th>Suspicious</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(a==<code>a</code>)]</td>
<td>((S,2),(E,1))</td>
<td></td>
</tr>
<tr>
<td>![a==<code>a</code>]</td>
<td>((S,1),(E,0))</td>
<td></td>
</tr>
<tr>
<td>[(b==5)]</td>
<td>((S,1),(E,1))</td>
<td></td>
</tr>
<tr>
<td>![b==5]</td>
<td>((S,1),(E,0))</td>
<td></td>
</tr>
<tr>
<td>[(c==16)]</td>
<td>((S,0),(E,1))</td>
<td></td>
</tr>
<tr>
<td>![c==16]</td>
<td>((S,1),(E,0))</td>
<td></td>
</tr>
<tr>
<td>ERROR</td>
<td>((S,0),(E,1))</td>
<td></td>
</tr>
</tbody>
</table>

S: means Coverage of Safe paths
E: means Coverage of Error paths
### 2.4.4.4 Tarantula on ARG - Example

#### 4. Calculate the Suspicious

$$\text{Suspicious}(s) = \frac{\text{failPath}(s)}{\text{total failPaths}} + \frac{\text{safePaths}(s)}{\text{totalsafePaths}}$$

<table>
<thead>
<tr>
<th>CFAEdge</th>
<th>Coverage</th>
<th>Suspicious</th>
</tr>
</thead>
<tbody>
<tr>
<td>[(a==<code>a</code>)]</td>
<td>((S,2),(E,1))</td>
<td>0.75</td>
</tr>
<tr>
<td>![a==<code>a</code>)]</td>
<td>((S,1),(E,0))</td>
<td>0.0</td>
</tr>
<tr>
<td>[(b==5)]</td>
<td>((S,1),(E,1))</td>
<td>0.60</td>
</tr>
<tr>
<td>![b==5)]</td>
<td>((S,1),(E,0))</td>
<td>0.0</td>
</tr>
<tr>
<td>[(c==16)]</td>
<td>((S,0),(E,1))</td>
<td>1.0</td>
</tr>
<tr>
<td>![c==16)]</td>
<td>((S,1),(E,0))</td>
<td>0.0</td>
</tr>
<tr>
<td>ERROR</td>
<td>((S,0),(E,1))</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**S**: means Coverage of Safe paths  
**E**: means Coverage of Error paths
Evaluation
3.2 Evaluation - Setup

1. Sv-Benchmarks and Bekkouche Benchmarks
2. Omega evaluation metric
3. Predicate Abstraction with $merge^{sep}$
4. Symbolic Execution with CEGAR
5. Test Generators with Branch Coverage
6. BenchExec for Time Measurement
7. Time limit: 900 seconds
8. Memory limit: 4869 MB
### 3.2.1 Evaluation - Setup - Benchmarks

Benchmark-set consists of 35 programs. An overview of error type is as following:

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Exploitation of the Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>assign</td>
<td>Wrong assignment expression</td>
</tr>
<tr>
<td>op</td>
<td>Wrong operator usage e.g. : &lt;= instead of &lt;</td>
</tr>
<tr>
<td>init</td>
<td>Wrong value initialization of a variable</td>
</tr>
<tr>
<td>branch</td>
<td>Error in branching due to negation of branching condition</td>
</tr>
<tr>
<td>assign-for-loop</td>
<td>Wrong assignment inside loop</td>
</tr>
<tr>
<td>if-for-loop</td>
<td>Wrong check inside loop</td>
</tr>
<tr>
<td>index-for-loop</td>
<td>Use of wrong array index</td>
</tr>
<tr>
<td>index-while</td>
<td>Use of wrong array index inside while loop</td>
</tr>
</tbody>
</table>

This type of bug is taken from BugAssist’s evaluation
3.2.2 Evaluation - Evaluation Metric

Worst-Case step

\[
\text{worst-case-step} = | \{ \text{codeline}; \text{rank(codeline)} \\
\leq \text{rank(faulty codeline)} \&\& \text{codeline} \neq \text{faulty codeline} \} |
\]

cardinality of a set of code lines, whose rank is less than or equal to the rank of the actual error code line and this set should not contain any faulty code line.

**Omega Percentage** = \( \text{Worst-Case step} / \text{Total Code-Lines} \)
3.2.2.1 Evaluation - Evaluation Metric - Example

<table>
<thead>
<tr>
<th>codeLine</th>
<th>suspicious</th>
<th>rank</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1.0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>16</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>11</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>14</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>0.5</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>0.5</td>
<td>2</td>
</tr>
</tbody>
</table>

Worst-Case step = |\{5, 6, 1, 2, 11, 14, 4, 12\}| = 8
Omega percentage = 8/20 = 0.400

The lower the omega result the better the technique

\[ \text{worst-case-step} = \| \text{codeline; rank(codeline)} \leq \text{rank(faulty codeline)} \&\& \text{codeline} \neq \text{faulty codeline} \| \]
3.2.3 Evaluation - Setup - merge operator

With default merge we get for all merged paths as suspicious value 0.5, therefore we use $\text{merge}_{\text{sep}}$. 

Figure 6.3: $\text{merge}_{\text{sep}}$

Figure 6.4: Default Merge

Figure 6.5: Comparison between ARGs created by PredicateCPA with $\text{merge}_{\text{sep}}$ (left) and with default merge (right) operators generated by CPAchecker
3.3.1 Evaluation - Overview

1. Tarantula SymExec vs Predicate Abstraction
2. Tarantula vs DStar and Ochiai
3. Test-Based Tarantula vs Formal-based Tarantula
3.4.1 Evaluation - Discussions - Symbolic vs Predicate

Symbolic Execution is better than predicate analysis with mergeSEP

Reasons:

1. ARG of Predicate is merged together so we need to apply mergSEP which is very expensive and slows down the analysis, and even runs the analysis for certain large programs infinitely.
3.4.1 Evaluation - Discussions - Symbolic vs Predicate

Symbolic Execution is better than predicate analysis with mergeSEP

**Reasons:**

2. ARG Graph from Predicate analysis is constructed in such a way that the bug location is more often on the safe path than on the failed path which lowers the suspicious.
3.4.2 Evaluation - Discussions - Tarantula vs DStar and Ochiai

DStar is better than Ochiai and Tarantula

Reasons:
DStar does not take TotalSafePaths into account in its suspicious form and with the help of the delta exponential variable, the suspicion of the fault position was increased.
3.4.2 Evaluation - Discussions - Tarantula vs DStar and Ochiai

Ochiai is better than Tarantula

Reasons:
Ochiai’s $\Omega$ percentage was almost the same as Tarantula’s, but Ochiai analyzed more test programs than Tarantula. The reason for this is that Ochiai does not need at least one failure path and at least one safe path in contrast to Tarantula.
3.4.3 Evaluation - Discussions - Formal-based vs Test-based Tarantula

Formal-based is better than test-based Tarantula

Reasons:
Klee and VeriFuzz very often generated bad analyse through the whole program, so the bug sometimes suspected 0.0. Quite often both techniques delivered only counterexamples but no safe cases, so Tarantula can work perfectly well, thus the suspicious is 1.
Future Work
4.1 Future Work

Future work should include:

1. The use of more advanced fault localization analysis on CPAchecker to choose the best fault localization technique or to design a new ranking method and use it as a default feature in CPAchecker.

2. The work on more ranking metrics, such as Barinel and Op2 is still open and can be analysed.

3. Improving CPAchecker to be able to not only analyse C-programs but also java and Java Script programs.
Conclusion
4.2 Conclusion

- DStar and Ochiai are improvements and work better than Tarantula
- Symbolic execution was able to identify potential faults, 88.57% of the chosen benchmarks with a very good percentage of $\Omega$, while predicate-merge-set found 60% of the total benchmarks with very good results from $\Omega$
- Klee was only successful in 17.14% of all benchmarks used
- VeriFuzz was better than Klee but not CPAchecker in 37.14%

⇒ In our experimental Evaluation:
Techniques such as model checking and data flow analysis can find subtle and more bugs in programs as test generators.
References


References

- M. Jose and R. Majumdar. “Cause Clue Clauses: Error Localization using Maximum Satisfiability” In: (2002), pp. 49, 76
Available Sources

The used benchmark-set, evaluation data and python script of test-based tarantula algorithm are available under:

https://gitlab.com/Schindar/fault_localization_tarantula
Thank you for your Attention Questions?