Renieris and Reiss’ Localization

Basic idea (over-simplified)

- We have lots of test cases
  - Some fail
  - A much larger number pass
- Pick a failure
- Find most similar successful test case
- Report differences as our fault localization

“nearest neighbor”
Collect *spectra* of executions, rather than the full executions

- For example, just count the number of times each source statement executed
- Previous work on using spectra for localization basically amounted to set difference/union – for example, find features unique to (or lacking in) the failing run(s)
  - Problem: many failing runs have *no such features* – many successful test cases have $R$ (and maybe $I$) but not $P$!
    - Otherwise, localization would be very easy
Renieris and Reiss’ Localization

- Some obvious and not so obvious points to think about
  - Technique makes intuitive sense
  - But what if there are no successful runs that are very similar?
    - Random testing might produce runs that all differ in various accidental ways
    - Is this approach over-dependent on test suite quality?
Renieris and Reiss’ Localization

Some obvious and not so obvious points to think about

• What if we minimize the failing run using delta-debugging?
  • Now lots of differences with original successful runs just due to length!
  • We could produce a very similar run by using delta-debugging to get a 1-change run that succeeds (there will actually be many of these)
    • Can still use Renieris and Reiss’ approach – because delta-debugging works over the inputs, not the program behavior, spectra for these runs will be more or less similar to the failing test case
Renieris and Reiss’ Localization

Many details (see the paper):
- Choice of spectra
- Choice of distance metric
- How to handle equal spectra for failing/passing tests?

Basic idea is nonetheless straightforward
The Tarantula Approach

- Jones, Harrold (and Stasko): Tarantula
- Not based on distance metrics or a Lewis-like assumption
- A “statistical” approach to fault localization
- Originally conceived of as a visualization approach: produces a picture of all source in program, colored according to how “suspicious” it is
  - **Green**: not likely to be faulty
  - **Yellow**: hrm, a little suspicious
  - **Red**: very suspicious, likely fault
The Tarantula Approach
The Tarantula Approach

- How do we score a statement in this approach? (where do all those colors come from?)

- Again, assume we have a large set of tests, some passing, some failing

- “Coverage entity” e (e.g., statement)
  - $\text{failed}(e) = \# \text{ tests covering } e \text{ that fail}$
  - $\text{passed}(e) = \# \text{ tests covering } e \text{ that pass}$
  - $\text{totalfailed, totalpassed} = \text{what you’d expect}$
The Tarantula Approach

How do we score a statement in this approach? (where do all those colors come from?)

\[
suspiciousness(e) = \frac{\text{failed}(e)}{\text{totalfailed}} + \frac{\text{failed}(e)}{\text{totalpassed}}
\]
The Tarantula Approach

\[
suspiciousness(e) = \frac{\text{failed}(e)}{\text{totalfailed}} + \frac{\text{failed}(e)}{\text{totalpassed}}
\]

- Not very suspicious: appears in almost every passing test and almost every failing test
- Highly suspicious: appears much more frequently in failing than passing tests
The Tarantula Approach

\[
\text{suspiciousness}(e) = \frac{\text{failed}(e)}{\text{totalfaile}} = \frac{\text{passed}(e)}{\text{totalpassed}} + \frac{\text{failed}(e)}{\text{totalfaile}}
\]

\text{mid()}

\begin{verbatim}
int x, y, z, m;
1  read (x, y, z);
2  m = z;
3  if (y < z)
4    if (x < y)
5      m = y;
6  else if (x < z)
7      m = y;
8  else
9    if (x > y)
10      m = y;
11  else if (x > z)
12      m = x;
13  print (m);
\end{verbatim}

Simple program to compute the middle of three inputs, with a fault.
The Tarantula Approach

Run some tests...
Look at whether they pass or fail
Look at coverage of entities

mid()

```
int x, y, z, m;
1  read (x, y, z);
2  m = z;
3  if (y < z)
4    if (x < y)
5      m = y;
6    else if (x < z)
7      m = y;
8    else
9      if (x > y)
10     m = y;
11    else if (x > z)
12     m = x;
13    print (m);
```

(3,3,5) (1,2,3) (3,2,1) (5,5,5) (5,3,4) (2,1,3)

<table>
<thead>
<tr>
<th></th>
<th>0.5</th>
<th>0.5</th>
<th>0.5</th>
<th>0.63</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.0</th>
<th>0.5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>read (x, y, z);</td>
<td></td>
<td></td>
<td></td>
<td></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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<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>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>print (m);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compute suspiciousness using the formula

```
suspiciousness(e) = \frac{\text{failed}(e)}{\text{totalfailed}}
```

Fault is indeed most suspicious!
The Tarantula Approach

- Obvious benefits:
  - No problem if the fault is reached in some successful test cases
  - Doesn’t depend on having any successful tests that are similar to the failing test(s)
  - Provides a ranking of every statement, instead of just a set of nodes – directions on where to look next
    - Numerical, even – how much more suspicious is X than Y?
  - The pretty visualization may be quite helpful in seeing relationships between suspicious statements
  - Is it less sensitive to accidental features of random tests, and to test suite quality in general?
  - What about minimized failing tests here?
Evaluating Fault Localization Approaches

• So, how do the techniques stack up?
• Tarantula seems to be the best of the test suite based techniques
  • Next best is the Cause Transitions approach of Cleve and Zeller (see their paper), but it sometimes uses programmer knowledge
  • Two different Nearest-Neighbor approaches are next best
  • Set-intersection and set-union are worst
• For details, see the Tarantula paper