Visual Mapping to Assist Fault Localization Using Output Influences.

Team Members: Aditi Noata, Balaji Ganesan

1. Introduction

The software debugging process consumes 50%-80% of the software's development and maintenance effort. Many tools have been developed to assist the programmer in the fast development and debugging of his program. The Visual mapping tool developed by James A Jones, Mary Jean Harrold and John Stasko in their paper "*Visualization of test information to assist fault localization*" helps the developer in identifying the suspicious statements in a program by coloring the statements according to their participation in testing.

The visualization technique discussed in the above paper considers a statement as covered merely by its presence in the execution trace for a particular test case. This increases the number of suspicious statements that need to be analyzed to find the buggy statements. This is because many statements may be spuriously marked as suspicious simply because they were exercised by many failed test cases while they may not have actually contributed to the failure.

Our project is aimed at refining this coverage criterion by selecting only those statements, which are influencing the program's output. This is as discussed in the paper "*Rigorous Data Flow Testing through Output Influences*" by Evelyn Duesterwald, Rajiv Gupta and Mary Lou Soffa. This paper proposes the use of output slices to determine whether the statement contributed to the program's output. Our refinement to the coverage criterion reduces the number of statements that are marked suspicious.

Formal Problem Statement

"To incorporate the notion of output influences in the Visualization technique in order to reduce the number of statements marked suspicious so that the developer can locate the faulty statement faster."

2. Steps in our approach:

1. We use the Aristotle tool to obtain the code coverage information for the given Siemens test suite and the program.

2. We initially use the information obtained above to color the statements in the program based on the coloring criteria discussed in Visualization paper. We are writing a tool similar to TARANTULA for this purpose.

3. We use the dependence graph obtained from the Aristotle tool to refine the coverage information obtained in step 1 so that only those statements, which actually influenced the output, are considered as covered.

4. Now the coloring tool will color the statements according to the new coverage information obtained in step 3.

5. We will then conduct a series of experiments to verify the effectiveness of our approach in reducing the percentage of suspicious statements in comparison to the existing technique.

Color Mapping Equation:

All three inputs: source code, test suite and the refined code coverage result are provided to the coloring module which colors the program statements according the following equation:

$$\text{Color(s)} = \text{low color (red)} + \frac{\%\ \text{passed(s)}}{(\%\ \text{passed(s)} + \%\ \text{failed(s)})} * \text{color range}$$

- Color(s) - Color for the statement s in the source code S.
- Low color - Value assigned to the red color (lowest) in the spectrum as 0.
- % Passed(s) – Number of passed test cases for the statement(s) / total number of the passed test cases in the entire test suite T.
- % Failed(s) – Number of failed test cases for the statement(s) / total number of the failed test cases in the entire test suite T.
- Color range - the difference in the value of the highest color (green/120) and the lowest color (red/0).

Example program:

Consider the following example program, which computes the sum of the elements and the minimum element in the given input array. Three errors have been introduced in this program, which marked, by the comments. This program for a given test input n=4 and a=(0,0,0,4), produces a correct output despite the presence of the three bugs.

```
(1)     input (n,a);
(2)     i := 2;
(3)     p := 1;
(4)     m = a[p];
(5)     while i<n do begin      /* should correctly be i<=n */
(6)             if a[p] <= a[1] then     /* should correctly be a[p] <= a[i] */
```

```
(7)                     p := i;
(8)             a[i] := a[i] + a[i -1];
(9)             i := i + 1;
        EndWhile
                                        /* Omitted m := a[p]; */
(10)    output('min is' m);
(11)    output('sum is' a[n]);
```

Using the existing approach for the above program, we will have all the statements marked as covered. However if we incorporate the notion of output influence by considering only those statements in the dynamic output slice as below, we will only mark statements 1,2,3,4,10 and 11 as covered.

```
(1)     input (n,a);
(2)     i := 2;
(3)     p := 1;
(4)     m = a[p];
(10)    output ('min is' m);
(11)    output ('sum is' a[n]);
```

3. Current Status of the Project.

We have obtained the code coverage information from the Aristotle tool and have visually mapped that information using the coloring tool developed by us. Now we are in the process of using the dependence graph information obtained from Aristotle to refine the coverage criteria.