Visual Mapping to Assist Fault Localization Using Output Influences.

Aditi Noata and Balaji Ganesan
Department of Computer Science,
University of Arizona, Tucson.
aditi, balaji @ cs.Arizona.edu

Abstract:

This project presents a refinement of the existing Visualization Technique for assisting Fault Localization by incorporating the idea of Output Influences. We have studied the prior work done in these fields and have combined the concepts introduced in two different papers. The existing Visualization technique uses color mapping to distinguish statements in a faulty program according to the likelihood of their correctness. We find that the present approach marks many statements as suspicious even though they may not be contributing to the program failure. We believe this is because, the present approach considers a statement as covered merely by its presence in the execution trace. We refine this technique using the Output Influence concept wherein a statement is considered as covered only if contributes to at least one successful run of the program.

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."

The organization of the report is as follows. In the next section, we discuss the previous work done in this field based on which we have developed our approach. In section 3, we discuss in detail the approach we have used to solve the aforementioned problem. In sections 4 and 5, we present our experiments and analyze the results. Finally we present the conclusions of our work in section 6 and also discuss some future work.

2. Related work:

Our work is based on prior work done in this field. In two papers [1, 2], the concept of using Visualization technique to assist the developer in fault location and the use of Output Influences for rigorous Data flow testing have been discussed.

Visualization Technique:

One of the most expensive and time-consuming components of the debugging process is locating the errors or faults. To locate faults, developers must identify statements involved in failures and select suspicious statements that might contain faults. Several tools have been developed to assist a programmer in fast debugging process. Similar work was implemented by James A. Jones et al in their paper “Visualization of Test Information to assist fault localization”. The technique proposed by them used color to visually map the participation of each program statement in the outcome of the execution of the program with a test suite, consisting of both passed and failed test cases. The statements are colored within a spectrum from green, yellow to red interpreted as non-faulty, suspicious and faulty statements respectively. Based on this visual mapping, a user can inspect the statements in the program, identify statements involved in failures, and locate potentially faulty statements. The paper also describes a prototype tool called TARANTULA that implements their technique along with a set of empirical studies to verify the effectiveness of their approach.

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:

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

- 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).

Note: The color range is from red to green with the middle of the spectrum containing yellow.

Using Output Influences:

The paper [2] by Evelyn Duesterwald et al, presented a refinement of existing data flow testing criteria through the notion of output influences in a program. Prior to this paper, existing data flow testing criteria considered exercising a definition-use pair in a successful test case as sufficient evidence of its correctness. They argue that the correctness is not demonstrated unless exercising the def-use pair has an influence on the computation of at least one correctly produced output value. i.e they refined the existing data flow criteria by requiring that an exercised def-use pair must be output-influencing to be considered tested by a test case.

The paper discussed several methods based on static and dynamic program slicing to compute the output influencing def-use pairs in a test case. Their method essentially involves generating the output slices from the dependence graph of the given program. They use a combination of static and dynamic dependence graphs for their purpose. The idea is to move as much work to compile time as possible. Thus the static dependence graph is generated partially at compile time and the dynamic information is added at run time.

From the dependence graph, they generate the output slices. Again they use a combination of static and dynamic output slices. They define the static and dynamic output slices as follows.

For a given program P, and a set of output nodes S,

- Static output-slice of P with respect to S is defined as the subprogram P' of P that when executed computes the same values in S as P does.
- Dynamic output slice of P with respect to S is defined as the subprogram P'' that when executed on Input I, computes the same values in S as P does.

Given dependence graph G = (N,E) for program P and a set of output nodes S in G, the Output slice is the sub graph G/S = (N',E') where

$$N' = \{ n \in N \mid \exists\, s \in S: s \Longrightarrow n \} \text{ and}$$
$$E' = \{ e \in E \mid \exists\, s \in S: s \Longrightarrow e \}$$

The output set thus generated can be used to determine the Cover set of the program. A cover set, Cover (t) is a subset of the potential def-use pairs in P. It contains only pairs that had influence on the computation of at least one correct output value produced by t.

```
(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]);
```

(i)

```
(1)     input (n,a);
(2)     i := 2;
(3)     p := 1;
(4)     m = a[p];
(5)     while i<n do begin
(8)             a[i] := a[i] + a[i -1];
(9)             i := i + 1;
        EndWhile
(10)    output ('min is' m);
(11)    output ('sum is' a[n]);
```

(ii)

```
(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]);
```

(iii)

Figure 1: Original program fragment (i), the static slice on the two output values in statements 10 and 11 (ii), and the dynamic slice for the two output slices on input n=4, a[0,0,0,4] (iii).

The approach followed by Evelyn Duesterwald et al is illustrated by the above example. The program fragment computes the minimum and the sum over an input array. The authors have introduced three errors that are documented in the comments in Figure 1(i). For a test case t_0: n = 4 and a[0,0,0,4], the program fragment accidentally produces the correct output values. (i.e the output is 0,4). When executing the fragment on this input, all statements are executed and in particular all def-use pairs inside the loop computation are exercised. Therefore, using the traditional data flow-testing criterion leads to premature conclusions about the correctness of the def-use pairs inside the loop computation that actually contains errors.

However using the Output Influence criterion, the authors have generated the static and dynamic output slices as shown in Figure 1(ii) and 1(iii). As can be seen from the combination of both the static and dynamic output slices, the statements 6 and 7 are not exercised by this run of the program. Hence the authors argue that this run will not be considered as successful using their approach.

3. Our approach:

We have incorporated the idea of output influences introduced by Evelyn Duesterwald et al, in the Visualization technique presented in the paper by James A Jones et al. This we believe will address the problem of spurious suspicious statements reported by the existing Visualization techinique.

Our approach involves the following steps.

1. To begin with we use the Aristotle tool [3] to obtain the code coverage information for the given program and a set of test cases.

2. We then use the Unravel tool [4] to obtain the output slice of the program.

3. Using a coloring tool designed by us, we color the statements in the program based on the coloring criteria discussed in the Visualization paper. i.e For each test case if a particular statement is present in the execution trace, the statement will be marked as covered. This coverage information is obtained for a reasonable number of test cases and finally the statements are colored based on the Coloring equation similar to the one used in the paper by James A Jones et al.

4. We then generate another color mapping based on the one above, except this time we use the Slicing Information together with the coverage information. i.e. A statement considered as covered in Step 3 will be considered as covered in this step only if that statement is present in the output slice of the program.

5. Finally we compare the color mapping generated by both approaches. We consider a statement as marked suspicious if it is in the color range from yellow to red.

Coloring Tool:

For our experiments, we generated a coloring tool similar to TARANTULA developed by James A Jones et al for their Visualization experiments. Our tool is written in java and takes the given program and two text files containing the two sets of statement numbers. The first set is the statements present in the execution trace of each program. The other set contains those statements, which are present in both the execution trace as well as the Output slice of the program. The statement numbers in the text file are padded with end markers to distinguish each test run.

The tool then displays two versions of the program side by side in a java applet, each bearing the color mapping based on the earlier approach and our approach. Thus we can compare the number of statements marked suspicious by both approaches. A statement marked in the color range from Yellow to red is considered a potential faulty statement. Rest of the spectrum (i.e) towards Green is considered to be potentially error free. A statement in white is either not tested by any test case or doesn’t affect the output of the program either positively or negatively.

Example program:

Consider the following example program, which computes the sum of the elements and the minimum element in the given input array. This is a simplified version of the example discussed in [2]. This program for a given test input n=4 and a=(0,0,0,4), produces a correct output despite the presence of the bug in the While statement where ‘Less than’ operator is used instead of ‘Less than equal to’ as it should have been.

```
// Program to find the minimum value and sum of
// elements in the array.
// Statement in error is commented.
main ()
{
        int i,j,m,p,n,a[100],res;
        i = 2;
        p = 1;
        printf("Enter n\n");
        scanf("%d",&n);

        for (j=1;j<=n;j++)
        {       printf("Enter number %d\n",j);
                scanf("%d",&a[j]);
        }
        m = a[p];

        while (i<n) /* should have been (I<=n) */
        {       if (a[p] <= a[i])
                        p = i;
                a[i] = a[i] + a[i -1];
                i = i + 1;
        }

        m = a[p];
        printf("Minimum is %d\n",m);
        res = a[n];
        printf("Sum is %d\n",res);
        exit(0);
}
```

Figure 2 (i): - Example Program.

Using the existing approach for the above program, we will have all the statements marked as covered. However using our approach, only those statements that are in the execution trace as well as in the Output slice will be marked as covered. In Figure 2(ii), all the statements in the output slice as obtained from the Unravel tool are marked in bold text.

```
main()
{
        int i,j,m,p,n,a[100],res;
        i = 2;
        p = 1;
        printf("Enter n\n");
        scanf("%d",&n);

        for(j=1;j<=n;j++)
        {
                printf("Enter number %d\n",j);
                scanf("%d",&a[j]);
        }

        m = a[p];

        while (i<n) /* should have been (I<=n) */
        {
                if (a[p] <= a[i])
                        p = i;
                a[i] = a[i] + a[i -1];
                i = i + 1;
        }

        m = a[p];

        printf("Minimum is %d\n",m);
        res = a[n];
        printf("Sum is %d\n",res);
        exit(0);
}
```

Figure 2 (ii): - Example Program. Statements in the output slice are in bold text.

Thus using the Output Influence in the above example makes the coverage criteria more rigorous than in the existing approach. When the coverage information for many test runs is collected as above, the final color mapping obtained will be different than using the existing approach. We find that our approach marks less number of statements as suspicious thereby reducing the amount of effort required by the developer to locate the fault.

4. Experiments:

For our experiments we used five c programs of varying complexity each of which had one or more known bugs. These programs were taken from the test cases provided with the Aristotle Tool and the Unravel Tool as well as one program from the book 'Find the bug'.

Coverage Information:

We started by using the Aristotle tool [3] to obtain the Statement coverage Information. We collected the statement numbers covered in each execution of a program by generating its instrumented code and running this instrumented code for each test case. The steps involved in generating the instrumented code and obtaining the statement numbers covered are as explained in [5].

It involves running the cfe analyzer provided in Aristotle to generate the control flow and the dependence graph information. We used the branch trace instrumenter on the input program to obtain the Instrumented code. We then ran this instrumented code and generate its trace. The trace information thus generated for all the test runs is collected to generate the trace history. This trace history will then be read by the 'Visual Mapping tool' and the statement coverage will be displayed using the color mapping discussed above.

Output Slice:

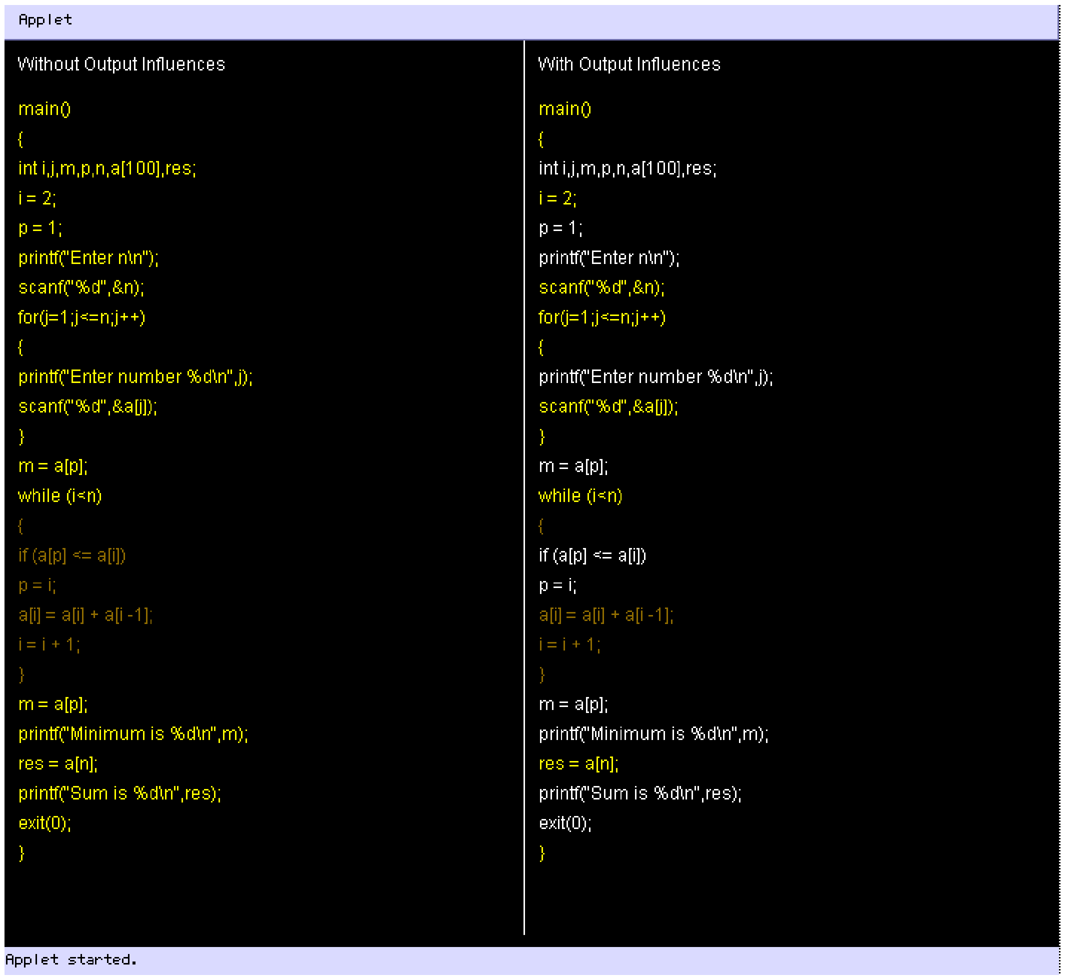
Unravel tool is used to obtain the program output slice for a selected variable. We used this tool to obtain the output slice for select output variables in the sample programs used by us.

Unravel has two phases – the first phase is called the analyzer which analyzes a C program written in ANSI C. The second phase is the slicer, which computes the output slice. Its GUI shows all the variables (local/global) and the functions a user can select for obtaining an output slice. After the variable has been selected, its output slice can be obtained from unravel by clicking on any statement using that variable.

We analyzed and obtained the output slice for all our test programs as explained above through unravel. We collected the statement numbers in each case and provided this as a text file to the Coloring tool. This information from the output slice was used to refine the coverage information obtained by the Aristotle tool in the previous step.

Visual Mapping:

Using the coverage information and the output slice obtained above, we generated the color mapping. The two files containing the statement numbers were read by the coloring tool and the source program mapped using both approaches was displayed. The applet shows the previous approach on the left and our approach on the right. The color mapping obtained for the example program discussed in the last section is as shown in the figure below.


Applet

Without Output Influences

```
main()
{
int i,j,m,p,n,a[100],res;
i = 2;
p = 1;
printf("Enter n\n");
scanf("%d",&n);
for(j=1;j<=n;j++)
{
printf("Enter number %d\n",j);
scanf("%d",&a[j]);
}
m = a[p];
while (i<n)
{
if (a[p] <= a[i])
p = i;
a[i] = a[i] + a[i -1];
i = i + 1;
}
m = a[p];
printf("Minimum is %d\n",m);
res = a[n];
printf("Sum is %d\n",res);
exit(0);
}
```

With Output Influences

```
main()
{
int i,j,m,p,n,a[100],res;
i = 2;
p = 1;
printf("Enter n\n");
scanf("%d",&n);
for(j=1;j<=n;j++)
{
printf("Enter number %d\n",j);
scanf("%d",&a[j]);
}
m = a[p];
while (i<n)
{
if (a[p] <= a[i])
p = i;
a[i] = a[i] + a[i -1];
i = i + 1;
}
m = a[p];
printf("Minimum is %d\n",m);
res = a[n];
printf("Sum is %d\n",res);
exit(0);
}
```

Applet started.

Figure 3: - The output of the Visual Mapping Tool.

The color mapping on the left is based entirely on the coverage information, whereas the one on the right uses both the coverage information and the output slice. As explained in the last section, this example program has a bug on the while statement and hence that statement and the statements inside the while loop are marked suspicious in the color mapping on the left. However our approach finds that the first two lines in the While statement do not affect the output. i.e Finding the sum of the elements.

5. Results:

The results obtained for the test programs are as shown in the Table 1 and the two charts below. We calculated the number of statements marked as suspicious in both approaches. We find that our method produces significant reduction in the number of statements marked as suspicious. We obtained an average reduction of 45% using our approach.

Program	Previous Approach	Our Approach	% reduction
sum.c	26	15	42.30
mid.c	12	9	25.00
select.c	17	12	29.41
flavors.c	28	12	57.14
amoeba.c	21	6	71.42

Table 1: Comparison of number of suspicious statements reported.

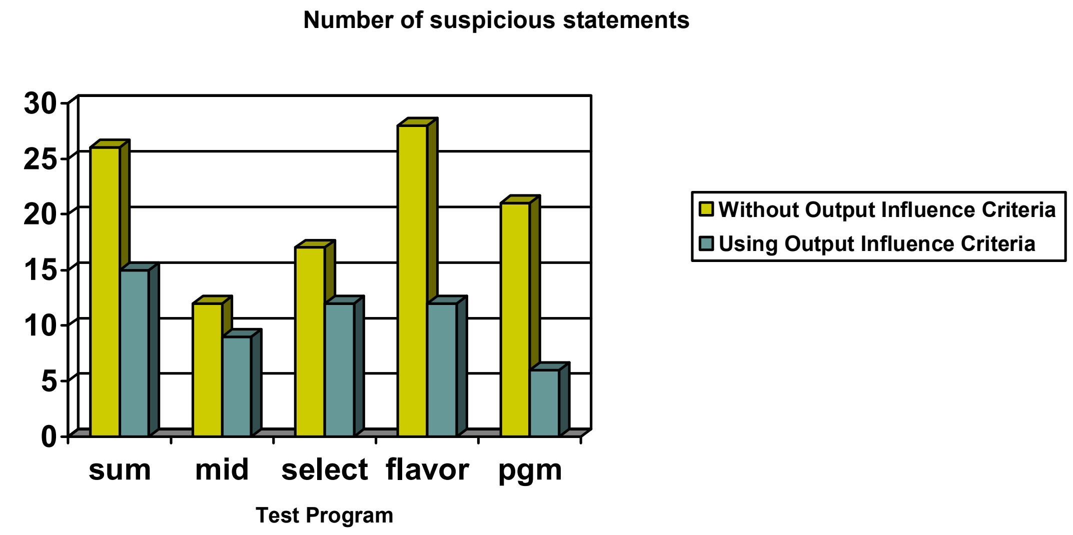


Figure 4: - Chart comparing the number of suspicious statements

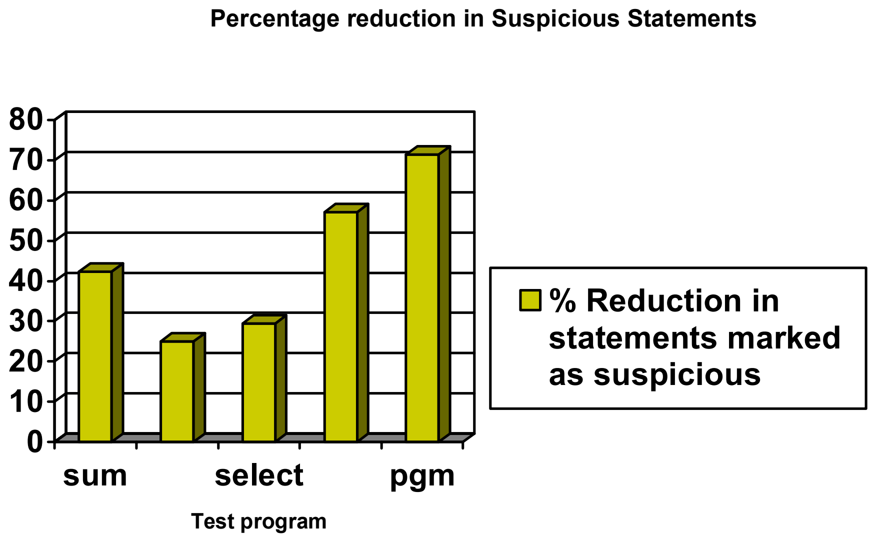


Figure 5: - Chart comparing the percentage reduction in the number of suspicious statements

6. Conclusions and Future work:

In this paper, we have tried to refine the existing Visualization techinique by incorporating the idea of Output Influence. Our method has shown significant reduction in the number of statements reported as spurious by the Visualization techinique. We have obtained an average reduction of 45% with the reduction as high as 71.42% for one program. These results are definite indicators that if the concept of output influences is incorporated to refine the coverage criteria, only the relevant suspicious statements would get marked thus reducing the number of statements a programmer has to look at while debugging his program.

Presently our color-mapping tool requires the input of the coverage information in the form of two files. These files are composed manually using the output obtained from the statement coverage tool – Aristotle and program slicing tool – Unravel. As future work we would like to automate this visualization tool to obtain the information directly from the other interfacing tools. We also plan to conduct further experiments in this regard and find practical applications for our approach.

Acknowledgements:

We wish to acknowledge the immense support we received from Prof. Neelam Gupta, who guided us all along the project.

References:

[1] James A. Jones, Mary Jean Harrold, John Stasko. *Visualization of Test Information to Assist Fault Localization.* In *Proceedings of the 24th International Conference on Software Engineering*. Pages: 467 – 477, 2002

[2] Evelyn Duesterwald, Rajiv Gupta, Mary Lou Soffa. *Rigorous Data Flow Testing through Output Influences.* In 2 nd Irvine Software Symposium, Irvine CA, Pages 131-145, Mar. 1992.

[3] Aristotle Analysis System, a system that provides program analysis information, and supports the development of software engineering tools. http://www.cc.gatech.edu/aristotle/Tools/aas.html

[4] Unravel, a prototype program slicing tool that can be used to statically evaluate ANSI C source code http://hissa.nist.gov/project/unravel.html

[5] Subject Infrastructure Repository. Site for information on how to use Aristotle tool for coverage information. http://csce.unl.edu/~galileo/sir/