Automatically Locating software Errors using Interesting Value Mapping Pair (IVMP)

Ajai Kumar[1], Anil Kumar[2], Deepti Tak[3], Sonam Pal[4],

[1,2]Sr. Lecturer, Krishna Institute of Management & Technology, Moradabad.INDIA
ajai4u87@gmail.com, chauhananil01@gmail.com

[3]Lecturer, Vivekananda Institute of Technology, Jaipur, INDIA
deeptitak10@gmail.com

4Lecturer, Jaipur National University, Jaipur, INDIA
sonam.pal@gmail.com

Abstract
Software does not always behave as expected due to errors. These errors can potentially lead to disastrous consequences. Unfortunately, debugging software errors can be difficult and time-consuming. Many techniques to automatically locate errors have been developed, but the results are far from ideal. Unlike other techniques that analyze existing state information from program executions, dynamic state alteration techniques modify the state of program executions to gain deeper insight into the potential locations of errors. However, prior state alteration techniques are generally no more effective than other techniques, and come at the expense of increased computation time. The Value Replacement technique performs aggressive state alterations to locate software errors by replacing the set of values used in different statement instances in failing program executions. In a set of benchmarks, interesting value mapping pair (IVMP) precisely identifies a faulty statement; interesting value mapping pair (IVMP) can be generalized to iteratively locate multiple errors.
Key word: IVMP, Value Replacement, Erroneously-Omitted Statements, Extraneous Statements

1. Introduction

An automated, dynamic state alteration technique called Value Replacement is developed for locating software errors. This technique analyzes program executions that fail due to incorrect output being produced. In such a failing execution, Value Replacement alters the execution state at a single statement instance, one after the other, by replacing the set of values involved at that statement instance with an alternate (did errant) set of values. Execution then proceeds from that point under the altered state. At the end of execution, the output is examined to determine whether or not it has changed to become correct. If the output has become correct, then there is a chance that the statement instance, at which the value replacement was performed, is faulty. The Value Replacement technique performs these value replacements at different statement instances in a failing execution, one at a time, to rank program statements according to how likely they are to be faulty. This is the essence of the Value Replacement technique.

1.1 Definition 1(Value Replacement)

Given a statement instance in the execution of a failing run, a value replacement involves replacing the set of values involved at that statement instance with an alternate (different) set of values. Execution then proceeds from that point under the altered state until the execution terminates. Program state that is altered under a value replacement can consist of global, local (stack), and heap values. Address values are not considered when performing value replacements.

1.1.1 Computing Interesting Value Mapping Pairs

If a value replacement causes the execution of a failing run to become correct, this fact is represented by an interesting value mapping pair (IVMP). An IVMP is associated with a statement instance in a failing execution, and is composed of two sets of values: the original set of values used at that statement instance, and the alternate set of values that can be substituted in place of the original values in order to cause the execution to produce correct output. An IVMP is a value mapping pair" because it is composed of a pair of value mappings (sets of values). An IVMP is interesting" because it represents how the state of a failing execution can be modified in order to cause the execution to become passing (i.e., produce correct output).

1.2 Definition 2(Interesting Value Mapping Pair)

An interesting value mapping pair (IVMP) is a pair of value mappings (original", alternate") associated with a particular statement instance in a failing run, such that: (1) original" is the original set of values used by the failing run at that instance; and (2) alternate" is an alternate (different) set of values such that if the values in original" are replaced by the values in alternate" at that instance during execution of the failing run, then the incorrect output of the failing run becomes correct.
To illustrate, Figure 1 shows three possible IVMPs for a given statement instance. The statement in this case is an if condition in which the < operator is mistakenly used instead of <=. The effect of this error is that whenever the operand values x and y are identical, then the condition will erroneously evaluate to false when it should have evaluated to true. As a result, all original sets of values in the IVMPs have identical values for x and y, and the condition evaluating to false. However, all alternate sets of values have different values for x and y that instead cause the condition to evaluate to the expected outcome of true. These alternate values can cause a failing run to pass (assuming, for instance, that neither x nor y are subsequently referenced and that there are no other errors in the program).
The Value Replacement technique involves searching for IVMPs that can be associated with a failing run. If a program statement is associated with at least one IVMP, then this statement can be shown to a act the output of a failing run such that the incorrect output becomes correct. The intuition is that these statements are more likely to be faulty, as compared to other statements that are not associated with any IVMPs. As will be discussed in detail later, this IVMP information is used to rank program statements according to how likely they are to be faulty. Given a failing run, the task of searching for IVMPs is straightforward: simply consider statement instances in the failing run one at a time, replacing the value mapping used at each one with a different value mapping, then checking to see if the output of the

Suspicious Statement:
// assume that '<' should actually be '<=' if (x < y)
Three Possible IVMPs:

**(1) ORIGINAL: {x=1, y=1, branch=FALSE}
ALTERNATE: {x=3, y=5, branch=TRUE}**

**(2) ORIGINAL: {x=8, y=8, branch=FALSE}
ALTERNATE: {x=1, y=2, branch=TRUE}**

**(3) ORIGINAL: {x=3, y=3, branch=FALSE}
ALTERNATE: {x=12, y=82, branch=TRUE}**

Figure .1: Example IVMPs at a statement.

run becomes corrected. If so, an IVMP has been found. Searching for IVMPs requires only a failing test case execution with the corresponding incorrect and correct outputs, and some set of alternate value mappings that can be applied at different statement instances in the failing execution.
In general, the set of all possible alternate value mappings at a statement instance can be theoretically in finite. For example, suppose the value 1 is used at a statement instance. Then the set of all possible alternate values is the set of \all values except 1," which is, in theory, an in finite set (in practice, the size of the set would be limited by the maximum number of values that can be stored in the associated storage location). It is impractical to perform a value replacement for every possible set of alternate values. A method is required to select a finite set of alternate mappings that can be applied at each statement. This is accomplished by extracting the (finite) set of alternate mappings for each statement from the execution traces of all test cases in an available test suite. This includes the test case associated with the failing execution being analyzed, since different instances of the current statement in the same execution, may involve different values. The extracted set of alternate mappings is called the value parole.

1.3 Definition 3 (Value Problem)

A value problem for a program with respect to a test suite is a mapping of each program statement to the set of all unique sets of values occurring at that statement during execution of test cases in the test suite. It is reasonable to assume the existence of a test suite for computing the value pro le since a failing test case is usually part of a larger suite of test cases. It has been observed that rich value pro les can result from only a few test cases, and yet the sizes of value pro les increase logarithmically in general as the number of test cases in the suites increase. This is because as information from more test cases is added to a value pro le, the sets of values used by a test case tend to match those already added to the value pro le from previous test cases. In the value pro le, the alternate sets of values between passing and failing executions are not distinguished. This is because alternate sets of values that may result in IVMPs can potentially come from any test case executions, regardless of whether the executions pass or fail.
input:

Faulty program P, and failing test case f (with actual and expected output) from test suite T.

```
output:
Set of identified IVMPs for f . algorithm SearchForIVMPs
begin
Step 1:  [Compute value proble for P  with respect to T ]
1:          valProf  := fg;

2:          for each test case t in T  do

3:          trace := trace of value mappings from execution
of t;

4:          augment valProf  using the data in trace;

end for
Step 2:  [Search for IVMPs in f ]
5:          trace := trace of value mappings from execution
of f ;

6:          for each statement instance i in trace do

7:          origMap := value mapping from trace  at i;

9:                   for each altMap in valProf  at s do
10:         execute f  while replacing origMap with altMap at
i;

11:         if  output of f  becomes correct then

12:         output IVMP (origMap, altMap) at i;

end for end for
end SearchForIVMPs
```

Figure 2: General algorithm for computing IVMPs in a failing run.

1.3.1. Examples of IVMPs Linked to Faulty Statements

It has been seen that IVMPs occur precisely at faulty statements in many
cases. In cases where this is not possible, IVMPs can occur at statements that are just one
static dependence edge away from faulty statements. Because of this, IVMPs can be useful for locating errors. Several examples are now presented that show different ways in which IVMPs can be closely linked to faulty statements. These examples are based on situations that are encountered using the Siemens benchmark programs.

1.3.2. IVMPs at a Faulty Statement

IVMPs can be found precisely at a faulty statement when applying an alternate set of values causes the faulty statement to define the correct value. Figure 2.3 shows a code fragment and a test suite based on Siemens program schedule, faulty version v9. This fragment of code involves a check on the number of input arguments (argc), so that the program terminates with an error message if there are too few input arguments specified. There is an o -by-1 error in this condition. The effect of this o -by-1 error is that when argc is equal to 3, the program will erroneously proceed as normal when it should have terminated early due to too few input arguments. Thus, executing test case B in results in a failure. However, for test case B, changing the value of argc at line 1 from 3 to 2 (which is the value used by test case A) causes the output of the failing run to become correct. Therefore, this represents an IVMP providing an important clue that at line 1 in the code fragment, the value of variable argc should be decremented by 1 (or equivalently, the value of constant 3 should be incremented by 1). In this case, the IVMP is located at precisely the faulty statement.

1.3.3 IVMPs Directly Linked to a Faulty Statement

In some cases, IVMPs may not be found precisely at the faulty statements. One situation where this can happen is when there is an error in a constant assignment statement.

```
argc := ...;
1:          if  (argc < 3) /* 3 should actually be 4 */

2:          print (\Too few");

3:          else

4:          print (\Okay");
```

Figure 3: Code fragment based on schedule, faulty version v9.

A constant assignment will never be associated with an IVMP because there are no alternate values at the assignment; every executed instance of a constant assignment wills de ne the same constant value. Instead, IVMPs can be found at the statements in which the defined constant values are used. Figure 2.4 shows a code fragment and test suite based on Siemens program, faulty version v7. The code fragment shows an erroneously-defined constant value at line 2, which is larger than it should be. The effect is that when the array index AltLayV al is 1, the condition at line 5 will erroneously evaluate to false instead of true due to the incorrect constant value defined at that position. Thus, executing test case B in Figure 4 results in a failing run. However, for test case B, changing the value of AltLayV al at line 5 from 1 to 0 (which is the value used by test case A) causes the output of the failing run to become correct. Assuming the value of index variable AltLayV al is correct, this IVMP provides the important clue that the value stored at array index 1 is incorrect. Further, since accessing array index 0 (with value 400) corrects the output of the failing run, this provides the hint that the value 550 at array index 1 should be changed to another value that is smaller.

1.3.4 IVMPs in the Presence of Erroneously-Omitted Statements

Another situation in which IVMPs cannot be found at an erroneous statement is when the error involves one or more missing statements. In these cases, IVMPs can still be found at nearby statements that can compensate for the effects of the missing code. Figure 4 shows an erroneous function and accompanying test suite inspired by schedule2, faulty version v1. The purpose of this function is to return the inputted value of x incremented by one, only when the value of y is positive (in bounds). If y is equal to 0, the function returns 0.The missing code at line 1 is meant to check whether y is negative (out of bounds), and if so, to return the original value of x without having incremented it. Since test case A

```
int foo(int x, int y)
1:      /* if  (y < 0) return x; */
2:      if  (y == 0) return 0;
3:      return x + 1;
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	(x,y) = (1,-1)	2	1	FAIL
B	(x,y) = (2,2)	3	3	PASS
C	(x,y) = (0,1)	1	1	PASS

Figure 4: Code fragment inspired by schedule2, faulty version v1.

has y with out-of-bounds value -1, then the function erroneously increments the value of x in this case when it should not have done so. When the value of x at line 3 is changed from 1 to 0 (which is the value used by test case C), then the output becomes 1 and is correct. This IVMP at line 3 provides the important clue that for the failing run corresponding to test case A, the value for x actually should not have been incremented. This suggests that a statement (the one at line 1) is missing in the above function that will prevent test case A from incrementing the value of x.

1.3.5 .IVMPs in the Presence of Extraneous Statements

Some program errors may involve extraneous statements. It turns out that IVMPs often occur precisely at extraneous statements where they have the effect of \canceling out" the effects of the extra code. For instance, an extraneous assignment statement to variable x can have an IVMP that forces the original value of x to be defined, rather than the new value for x that resulted from the extra code.

1.3.6 .The Need to Consider Multiple Failing Runs

Although IVMPs often occur at or near faulty statements, a significant challenge to using IVMPs for locating errors is that IVMPs can be found at other statements besides those that are faulty. This is possible because there are often multiple statements exercised during a failing execution whose values can be changed to cause the output to become correct. There are two main causes for this, referred to as the dependence cause and the compensation cause for IVMPs at multiple statements.

Dependence Cause. IVMPs may be found at different statements that are all part of the same de definition-use chain in a program. This is because if a statement S_1 defining a variable x has an IVMP associated with it, there's a chance that another statement S_2 that uses x will also have an IVMP associated with it. In such cases, changing the value of x at either S_1 or S_2 can correct the program output, even though only one of the two statements may contain an error.

Compensation Cause. This occurs when IVMPs are found at two different statements that do not appear to be related to each other at all, yet they both in hence the output such that applying an alternate set of values at either statement can compensate for the effects of the error on the program output, thereby making the output correct. To address the challenge posed by the dependence and compensation causes for IVMPs at multiple statements, the technique considers IVMPs computed from multiple failing runs. A dependence chain with IVMPs in one failing run may not exist in another failing run that may involve different dependence chains. Also, IVMPs that happen to compensate for an error in one failing run are unlikely to compensate for the error in the same way in another failing run. Considering multiple failing runs is particularly effective when the failing runs exercise very different paths in the program. Since all failing runs must traverse the error (assuming a single error exists), the statements that are associated with IVMPs in more failing runs have a greater likelihood of being faulty. Therefore, IVMP statements are ranked using the intuition that statements associated with IVMPs in more failing runs are more likely to be faulty, than statements that are associated with IVMPs in fewer failing runs. Consider the example program with accompanying test suite.

In this example program, there is an error at line 2 in which

the addition operator is mistakenly used instead of the subtraction operator. In cases where inputted value y is 0, the defined value of a at line 2 will be correct regardless of the error. As a result, only test cases A and B pass, while test cases C and D fail.
Consider failing test case C. An IVMP is identified at line 2 because changing the values of x and y respectively from 1 and 1, to 0 and 0 (which are used by test case A), will correct the program output. Also, an IVMP is identified at line 6 because changing the used value of a from 2 to 0 (which is the value of a used by test case A), will correct the output as well. Although IVMPs are found at lines 2 and 6, only one of these lines contains the actual error. The IVMP at the other line is present due to the dependence cause for IVMPs at multiple statements. To help distinguish between these two statements, another failing run is considered.
When considering failing test case D, an IVMP is identified at line 2 because changing x and y from 0 and 1, to -1 and 0 (used by test case B), will correct the output. Also, an IVMP is identified at line 4 because changing the value of a here from 1 to -1 (the value of a in test case B) will correct the output. Here, IVMPs are found at lines 2 and 4.

```
1:    /* let (x,y) be input values */
2:    a := x + y;    /* should be x y */
3:    if  (x < y)
4:        write(a);
5:    else
6:        write(a + 1);
```

Test Case	Input Values	Actual Output	Expected Output	Result
A	(x,y) = (0,0)	1	1	PASS
B	(x,y) = (-1,0)	-1	-1	PASS
C	(x,y) = (1,1)	3	1	FAIL
D	(x,y) = (0,1)	1	-1	FAIL

Figure 5 Example to motivate the need to consider multiple failing runs.

Consider the statements with IVMPs in both failing runs C and D. Line 2 is associated with IVMPs in both failing runs, whereas lines 4 and 6 are associated with IVMPs in only one failing run each. Therefore, line 2 is more likely to be faulty than either lines 4 or 6.The example from Figure 2.6 shows the benefit of considering IVMPs from multiple failing runs when ranking program statements using IVMPs

1.3.7 .Ranking Statements using IVMPs

Given a faulty program and a test suite containing multiple failing runs, the statements exercised by the failing runs are ranked in decreasing order of suspiciousness value (likelihood of being faulty). Let F be the set of all failing runs in an available test suite, and let ST $MT_{IV\ MP}$ (f) refer to the set of all program statements associated with at least one IVMP identified from failing run f . Then the suspiciousness of a statement s, suspiciousness(s), can be defined as the number of failing runs in which at least one IVMP was identified for that statement.

2. Effectiveness of Value Replacement

2.1. Setup for Experiments

2.1.1 Implementation Details

The implementation uses [1] the Val grind infrastructure for dynamic binary translation. This system provides a synthetic CPU in software and allows for dynamic bi-nary instrumentation of an executing program. Val grind comes with a set of tools to perform tasks such as debugging and pro ling, but new tools were created to record definition use tracing information and to perform value replacements. Val grind allows for instrumentation at the granularity of machine code instructions, so the implementation records traces in terms of instruction instances, and performs value replacements at the binary instruction level. Instructions are then mapped back to their corresponding statements (source code line numbers) when necessary to compute a ranked list of program statements. Note that when an alternate set of values is applied at an instruction instance in the implementation, the original values are actually overwritten in their respective memory or register locations. As a result, any subsequent uses of these locations in subsequent instructions will involve the new values. In other words, the implementation ensures that the state of an executing program is properly modified at the point of a value replacement during execution. Also, in the experiments, several techniques are developed (described later in Chapter 4) that significantly reduce the search space for identifying IVMPs, while still allowing for highly effective results for locating errors in the benchmark programs. The experiments were run on a Dell Power Edge 1900 server with two Intel Xeon quad-core processors at 3.00 GHz, and 16 GB of RA.

2.1.2 Subject Programs and Test Suites

The Siemens suite programs listed in Table 1 are used for the experiments. From left to right, the columns in this table show the program name, the number of lines of code, the number of provided faulty versions (each containing a seeded error), the average number of test cases in each created test suite (in parentheses, the total number of test cases available in the provided test case pools), and a brief description of the program functionality. The Siemens suite programs, along with their corresponding faulty versions and test case pools, were obtained from the Software-artifact Infrastructure Repository, organized by researchers at the University of Nebraska Lincoln.

All faulty versions contain seeded errors. These errors are related to computation of non-address values, including errors such as operator and operand mutations, missing and extraneous code, and constant value mutations. These types of computation-related errors are distinct from memory errors, such as those that involve accessing incorrect memory locations and assigning incorrect pointer values. As a result, the implementation ignores address values that are involved in each program execution.

Most faulty versions are seeded with a single error in a single statement, but some faulty versions involve modifications to several statements. A few faulty versions were excluded because they did not yield any failing test cases from the provided test case pools. One of the faulty versions from program ptok2 was also excluded because the error in this case caused execution to loop for a very long time, causing traces to be very long and the Val grind-based implementation to run out of memory.

2.2. Effectiveness Results and Discussion

Experimental results are shown for each of the statement ranking techniques of faulty versions with associated ranked lists of statements in each specified score range, for both the basic IVMP and Tarantula ranking techniques. The results for each of the three techniques that are variations of the basic techniques. a graphical view of this data. In the graph, the x-axis represents the lower bound of each score range, and the y-axis represents the percentage of faulty versions achieving a score greater than or equal to that lower bound. In the results, percentages are computed with respect to faulty versions from among the Siemens programs. This presentation of data follows the convention of Jones et al. [2]. However, whereas [2] computes scores with respect to the total number of pro-gram statements, scores are computed here with respect to the total number of statements exercised by failing test cases in the suite. This is because statements that are not exercised by any failing test cases can be safely ignored when trying to locate the corresponding errors.

3. Conclusion

Interesting value mapping pair (IVMP) performs aggressive state alteration by replacing the set of values involved in each statement instance in a failing execution with alternate sets of values. If any value replacement causes the output of the execution to change and become correct, then the statement associated with the value replacement is likely to be associated with an error. It was shown how Value Replacement can be generalized into an iterative technique for locating multiple simultaneous errors in software. Also, several techniques to improve the efficiency of interesting value mapping pair (IVMP) were presented.

References

[1] D. Jerey, N. Gupta, and R. Gupta. Fault localization using value replacement. International Symposium on Software Testing and Analysis, July 2008.

[2] L. Jiang and Z. Su. Context-aware statistical debugging: From bug predictors to faulty control on paths. Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, November 2007.