Automatically Allocating Multiple Simultaneous Errors

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

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

Abstract

This technique uses multiple failing runs in order to rank program statements according to likelihood of being faulty. However, the technique implicitly assumes that all considered failing runs fail due to the same error. In the event that multiple simultaneous errors are present in software, different failing runs may fail due to different errors,. This can decrease the perceived suspiciousness of the faulty statements as compared to the non-faulty statements, thereby making it more difficult to isolate the faulty statements and decreasing the effectiveness of the technique for locating errors. Three variations of this idea are developed that involve different techniques for computing a new ranked list of statements on each iteration. Each ranked list is computed using the updated dynamic information that results from fixing the previously-located error.

Keywords: *multiple simultaneous errors, faulty statements, Minimal-Computation Technique Partial-Recomputation Technique, Full-Recomputation Technique.*

1. Introduction

In the previous chapter, the Value Replacement state alteration technique for locating software errors was developed. This technique uses multiple failing runs in order to rank program statements according to likelihood of being faulty. However, the technique implicitly assumes that all considered failing runs fail due to the same error. In the event that multiple simultaneous errors are present in software, different failing runs may fail due to different errors, or different combinations of errors. This can decrease the perceived suspiciousness of the faulty statements as compared to the non-faulty statements, thereby making it more difficult to isolate the faulty statements and decreasing the effectiveness of the technique for locating errors. This chapter shows how to generalize the Value Replacement technique into an iterative technique that can effectively handle the situation when multiple errors are present in software. The goal is to present a ranked list of program statements to the developer to isolate each individual error; each time a located error is fixed, then another ranked list is presented to the developer as long as at least one test case still fails. Three variations of this idea are developed that involve different techniques for computing a new ranked list of statements on each iteration. First, a Minimal-Computation technique is developed in which Value Replacement is applied only once to rank program statements, and the search for all errors is performed within the single ranked list (i.e., the same ranked list is reported to the developer on each iteration). This simple technique has relatively low cost as compared to the other techniques that will be described, because it performs Value Replacement only once to locate all errors. However, the effectiveness of this technique is relatively low because only a single ranked list is used to locate all errors; the ranked list is never updated to account for revised dynamic information that results from fixing an error. Second, a Full-Recomputation technique is developed in which Value Replacement is iteratively invoked to find and x one error at a time. This technique is highly effective as compared to the other techniques, because it computes a new ranked list to locate each error. Each ranked list is computed using the updated dynamic information that results from fixing the previously-located error.

2. Techniques to Locate Multiple Errors

Figure 1 depicts an overview of the core Value Replacement technique (A) that is used to locate single errors, and three variations of the technique (B{D) that can be used to iteratively locate multiple errors. In the core technique (Figure 1 (A)), a faulty program and associated test suite are passed as input to the Value Replacement algorithm, which computes a ranked list of program statements that can be examined by a developer in order to find and the (single) error. To handle multiple errors, Figure 1(B) illustrates the Minimal-Computation technique in which Value Replacement is performed only once, and the developer uses the single ranked list of statements to search for faulty statements as needed. Figure 1 (C) shows the opposite extreme: the Full-Recomputation technique in which Value Replacement is invoked to allow a developer to find and an error, then this process is fully repeated on the new version of the program as necessary until all errors are fixed. Figure 1 (D) illustrates the Partial-Recomputation technique that performs only partial Value Replacement computation on each iteration. The three techniques (B {D) for handling multiple simultaneous errors are now described in detail.

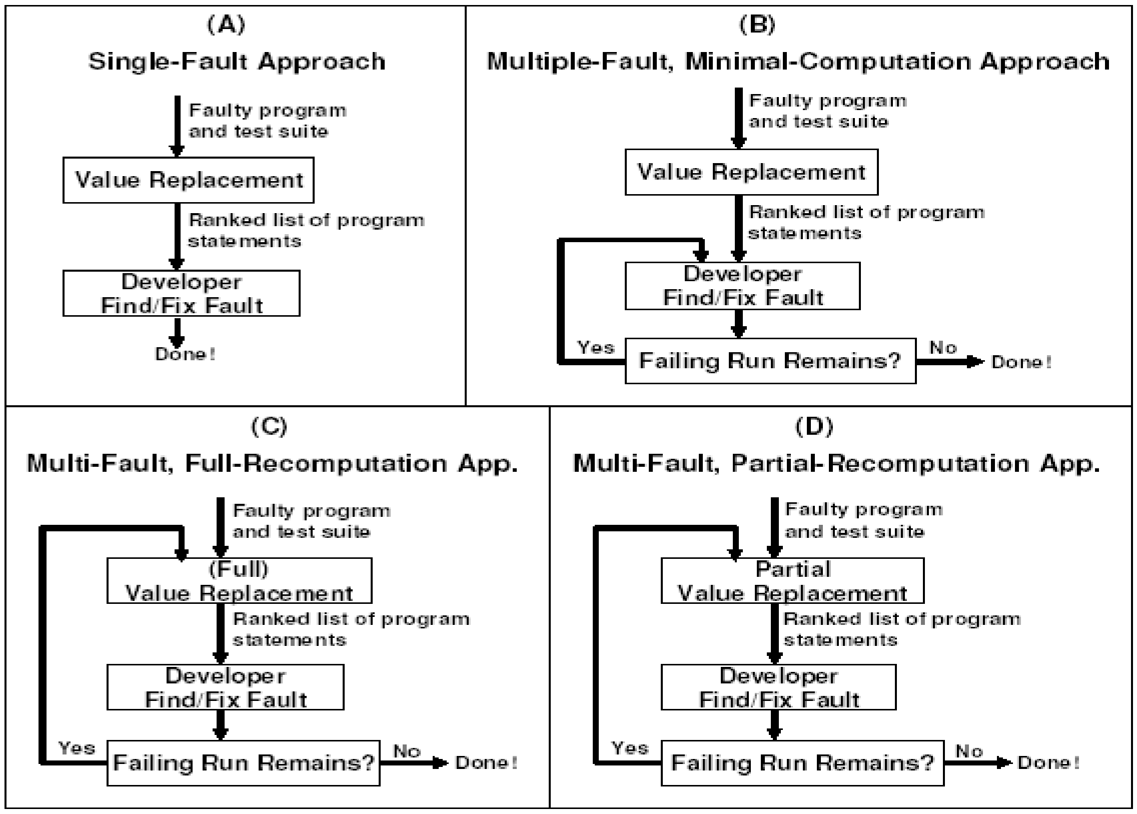

(A)
Single-Fault Approach
Faulty program and test suite
Value Replacement
Ranked list of program statements
Developer Find/Fix Fault
Done!
(B)
Multiple-Fault, Minimal-Computation Approach
Faulty program and test suite
Value Replacement
Ranked list of program statements
Developer Find/Fix Fault
Yes
Failing Run Remains?
No
Done!
(C)
Multi-Fault, Full-Recomputation App.
Faulty program and test suite
(Full) Value Replacement
Ranked list of program statements
Developer Find/Fix Fault
Yes
Failing Run Remains?
No
Done!
(D)
Multi-Fault, Partial-Recomputation App.
Faulty program and test suite
Partial Value Replacement
Ranked list of program statements
Developer Find/Fix Fault
Yes
Failing Run Remains?
No
Done!

Figure 1: Single-fault core technique (A) and Multi-fault generalized technique (B-D)

3. Minimal-Computation Technique

The algorithm for the Minimal-Computation technique is presented. The technique takes as input a faulty program and a corresponding set of test cases containing at least one failing run. First, the Value Replacement technique is executed to obtain a ranked list of program statements (line 1). Then, as long as a failing run exists in the test suite with respect to the current version of the program (line 3), the ranked list is used to locate an error in the program (line 4), which is then fixed (line 5), resulting in a new version of the program with the error removed. The actual identification and fixing of a faulty statement is done manually by a developer. If at least one run still fails on the new version of the program, then the (same) ranked list is consulted again to find and x another error (back to line 3). Under this technique, the computation time is expected to be comparable to that of the single-fault core Value Replacement technique.

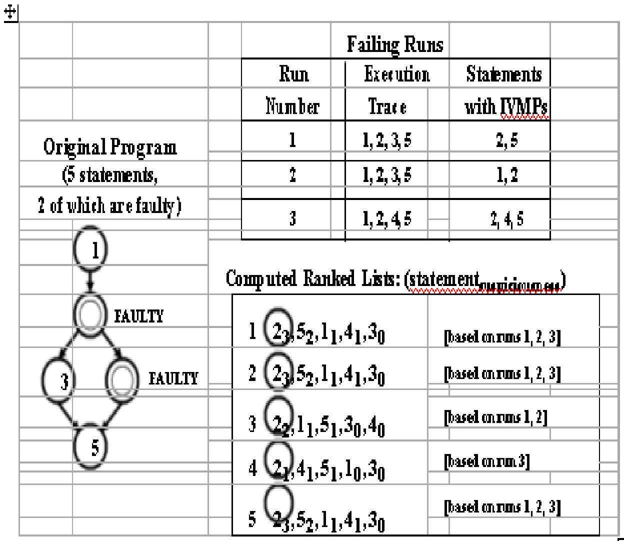
First, the Value Replacement technique is executed to obtain a ranked list of program statements (line 1). When, as long as a failing run exists in the test suite with respect to the current version of the program (line), the ranked list is used to locate an error in the program (line 4), which is then fixed (line 5), exulting in a new version of the program with the error removed. The actual identification and fixing of a faulty statement is done manually by a developer. If at least one run still fails on the new version of the program, then the (same) ranked list is consulted again to find and another error (back to line 3). Under his technique, the computation time is expected to be comparable to that of the single-fault core Value replacement technique, since a ranked list of program statements need be computed only once.

4. Full-Recomputation Technique

The algorithm for the Full-Recomputation technique is presented in Figure 2.
This technique is identical to the Minimal-Computation technique, except the invocation of the Value Replacement technique has been moved to inside the main loop. The effect is that a revised ranked list is computed on each iteration when an error is found and fixed. This ensures that the technique has up-to-date data that can be used to compute a more effective ranking on each iteration. However, the timing requirements increase significantly because Value Replacement must be invoked on every iteration to search for new IVMPs.Example The Partial-Recomputation technique is demonstrated with an example. Figure 2 shows an example control- own graph of a program containing 5 statements, two of which happen to be faulty. Suppose a test suite is available that contains 3 failing runs as depicted in the figure, with associated execution traces and sets of statements containing IVMPs as shown. In this case, two of the runs fail due to faulty statement 2, and one of them fails due to faulty statement 4. In the first step, the set of statements exercised by failing runs is identified (all statements in this case). Next, a ranked list of program statements is computed and associated with each one of these statements, by ordering statements according to suspiciousness value. Recall that the suspiciousness value is the number of (considered) failing runs in which the associated statement has an IVMP.

5. Partial-Recomputation Technique

The algorithm for the Partial-Recomputation technique is presented in this technique consists of two main steps. In the rest step, a set of ranked lists is computed, and these lists are used to find and x a rest error in the program. In



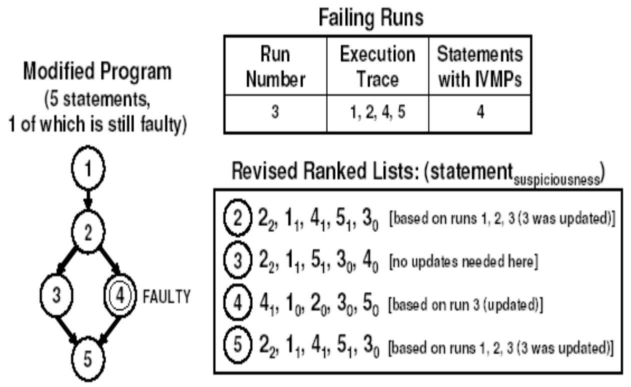
	Failing Runs	
Run Number	Execution Trace	Statements with IVMPs
1	1, 2, 3, 5	2, 5
2	1, 2, 3, 5	1, 2
3	1, 2, 4, 5	2, 4, 5

Computed Ranked Lists: (statement$_{suspiciousness}$)

1	$2_3, 5_2, 1_1, 4_1, 3_0$	[based on runs 1, 2, 3]
2	$2_3, 5_2, 1_1, 4_1, 3_0$	[based on runs 1, 2, 3]
3	$2_2, 1_1, 5_1, 3_0, 4_0$	[based on runs 1, 2]
4	$2_1, 4_1, 5_1, 1_0, 3_0$	[based on run 3]
5	$2_3, 5_2, 1_1, 4_1, 3_0$	[based on runs 1, 2, 3]

Figure 2 Abstract example for the partial recomputation technique

the second step, the technique iteratively performs partial Value Replacement re-computation, updates any acted ranked lists, and then uses the revised ranked lists to find and _ a next error. Step 1: Initialize ranked lists and locate the _rest error .In this step, the approach rest collects together all statements exercised by failing runs to consider for ranking purposes .Next, for each of these statements s, a ranked list of program statements is computed using Value Replacement by searching for IVMPs in only those failing runs exercising ,The intuition for this step is as follows: it is known that at least one of the statements s is faulty, and maximum suspiciousness is most likely to be achieved for such a statement if statements are ranked based on the IVMP information of only those failing runs which exercise s. Since the faulty statements are not known,Next, the technique identifies the ruts ranked list (from among the 5 computed lists) to remove and report to a developer. This is the one with highest suspiciousness values at the front of the list. Ranked lists 1, 2, and 5 have the _rest ranked element with highest suspiciousness. However, since these lists happen to be identical (no ties can be broken), an arbitrary choice is made from these lists. Suppose list 1 is selected, removed, and reported to the developer. Then faulty statement 2 is immediately identified because it occurs at the front of the selected list. The developer can then this faulty statement. how the situation might look after faulty statement 2 is fixed. In this case, statement 4 is the only remaining faulty statement. Assume that run 3 is the only run that still fails. Further assume that on the new version of the program, run 3 is associated with an IVMP at only statement 4. Next, the second main step of the



Failing Runs

Run Number	Execution Trace	Statements with IVMPs
3	1, 2, 4, 5	4

Revised Ranked Lists: (statement$_{suspiciousness}$)

2 $2_2, 1_1, 4_1, 5_1, 3_0$ [based on runs 1, 2, 3 (3 was updated)]
3 $2_2, 1_1, 5_1, 3_0, 4_0$ [no updates needed here]
4 $4_1, 1_0, 2_0, 3_0, 5_0$ [based on run 3 (updated)]
5 $2_2, 1_1, 4_1, 5_1, 3_0$ [based on runs 1, 2, 3 (3 was updated)]

Figure 3. Abstract example for the Partial-Recomputation technique, part 2

Partial-Recomputation technique is executed. First, the approach identifies the subset of newly-failing runs that need to be re-searched for IVMPs. In the example, failing run 3exercises statement 2 (the most recently-fixed statement), so run 3 must be re-searched for IVMPs. In practice, not all failing runs may need to be re-searched for IVMPs in this step. Next, from among the remaining ranked lists, only lists 2, 4, and 5 are a reacted by the new IVMPs and need to be updated (list 3 was not originally computed using run 3).In the original version of the program, run 3 was associated with IVMPs at statements
2, 4, and 5. However, in the new version of the program (with corrected statement 2), run 3 is associated with IVMPs at only statement 4. Thus, ranked lists 2, 4, and 5 are updated to reject a decrease of 1 in the suspiciousness values for statements 2 and 5 (shown in Figure 3). Now, the next ranked list to remove and report to the developer is selected. In this case, the technique selects the ranked list from among those remaining, that is most different from the first-selected ranked list, in terms of the elements near the fronts of the lists. Since the _first-selected ranked list had started with statement 2, then from among the remaining lists, list 4 is the most different because it is the only remaining list that does

6. Techniques and Metric for Comparison

The experiments compare the effectiveness for locating errors using the following techniques that generalize Value Replacement and handle multiple simultaneous errors. Program Name # 5-Error Avg. Test Suite Size Faulty Versions (# Failing Runs/# Passing Runs)

6.1. Minimal-Computation technique (MIN).

Under this technique, the core Value Replacement technique is applied only once to obtain a single ranked
example, in the replace program, the FULL approach has an average score of 86.98%, while the PARTIAL approach

list of program statements, which is then consulted as necessary until all errors are located.

6.2. Full-Recomputation technique (FULL)

Under this technique, the original Value Replacement technique is iteratively applied to locate and each error, one at a time.

6.3. Partial-Recomputation technique (PARTIAL).

Under this technique, multiple ranked lists of program statements are computed and iteratively revised through partial recomputation of IVMPs (from only a subset of failing runs) to locate and each error.

6.4. The ideal situation (IDEAL).

The ideal" situation for finding each error is considered to be the case where that error exists in isolation in a program (with no other errors present). This situation is most likely to lead to the best location results for each error, when using Value Replacement. These ideal" single-error results are used to compare against the above three techniques that handle multiple errors. Note that this definition of \ideal" is given with respect to the Value Replacement technique locating single errors that are present in isolation. This definition is different than the more general notion of ideal" results for error location, in which a faulty statement is uniquely given highest suspiciousness.

7. Effectiveness Results and Discussion

The average score values achieved for each located error (from among all individual errors contained within the multiple-error versions associated with each benchmark program). As shown in the table, the FULL approach is able to achieve average score values that are very close to the IDEAL values in most cases (within one or two percentage points). The exceptions are programs in which the FULL approach achieves average results that are about 5% {6% less than the IDEAL results. It was found that for these three programs that contain relatively few distinct errors, there were a small number of particular errors in which IVMPs could not be found at the faulty statements, thus resulting in poor ranking results. Since these \problem" errors were repeatedly selected from a relatively small set of total errors, they were present in relatively many of the multiple-error faulty versions for these programs, negatively affecting the average results. In all cases, the PARTIAL approach is able to achieve average score values that are within 5% of the FULL approach. In some cases, the difference is quite small. For yields almost the same average score: 86.50%. For ptok2, FULL has an average score of 84.37% while PARTIAL has 84.13%.

8. Conclusion

In this chapter, the state alteration technique called Value Replacement was developed for assisting in the task of locating software errors. This technique repeatedly alters the state of failing executions by performing value replacements to identify IVMPs. These IVMPs show how values used at particular program statements can be altered so that failing runs instead produce correct output. Using these IVMPs, executed statements can be ranked according to their likelihood of being faulty. Experimental results show that for the benchmark programs studied, Value Replacement can produce ranked lists of statements that are generally very effective at quickly leading a developer to a faulty statement.

References

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. A dynamic modeling approach to software multiple-fault localization. Proceedings of the 19th International Workshop on Principles of Diagnosis, pages 7{14, September 2008.

[2] H. Agrawal, J. R. Horgan, S. London, and W. E. Wong. Fault localization using execution slices and dataow tests. Proceedings of the 6th IEEE International Symposium on Software Reliability Engineering, October 1995.

[3] P. Ammann and J. Knight. Data diversity: An approach to software fault tolerance.
IEEE Transactions on Computers,

[4] M. Bond. Diagnosing and tolerating bugs in deployed systems. Ph.D. Thesis, The University of Texas at Austin, 2008.

[5] N. Gupta, H. He, X. Zhang, and R. Gupta. Locating faulty code using failure-inducing chops. IEEE/ACM International Conference on Automated Software Engineering,November 2005.