A Fault based Approach to Test Case Prioritization

Faiza Farooq
Department of Computer Science
Capital University of Science & Technology
Islamabad, Pakistan
faizafarooq287@yahoo.com

Aamer Nadeem
Department of Computer Science
Capital University of Science & Technology
Islamabad, Pakistan
anadeem@cust.edu.pk

Abstract: **Regression testing is performed to ensure that the no new faults have been introduced in the software after modification and the software continues to work correctly. Regression testing is an expensive process because the test suite might be too large to execute in full. Thus to reduce the cost of such testing, regression testing techniques are used. One such technique is test case prioritization. Software testers assign priority to each test case to make sure that the test cases with higher priorities are executed first, in case of not having enough resources to execute the whole test suite. Test case prioritization is mainly used to increase fault detection rate of test suite which is the measure of how early faults are detected. In this paper, we propose an approach which exploits mutation testing in order to assign priorities to test cases. Using mutation testing, we introduce different faults in original program thus creating a number of mutated copies of the program and test case that exposes maximum number of these faults is given the highest priority. We report the outcomes of our experiments in which we applied our technique to test suites and calculated the fault detection rates produced by the prioritized test suites, comparing those rates of fault detection to the rates achieved by existing prioritization technique. The resulting data shows that prioritization technique proposed improved the fault detection rate of test suites.**

Keywords— Regression testing; test case prioritization; mutation testing

I. INTRODUCTION

Whenever software is modified, it is retested to ensure that new errors have not been introduced in the code, which was previously tested and it continues to work correctly. Such testing is called regression testing[1]. It provides the confidence that the new addition to the software does not change the behavior of the existing part of the software. Regressing testing is an expensive process, and becomes even more complicated as the software evolves, because the test cases in the test suite tends to increase, making it even more complex and expensive to execute the entire test suite [2].

Number of techniques including minimization, selection and prioritization have been introduced to select test cases for regression testing, so that the number of test cases do not become too large as the software evolve. Software testers use these techniques to lower the cost of regression testing without reducing the overall effectiveness.

Test suite minimization eliminates redundant test cases based on some coverage criteria [3].

Test case selection is used to select the most relevant test cases which are valid and traverse the modified parts of the program[4].

The test case prioritization, in contrast, does not eliminate any test cases, rather tends to find an ordering of test cases that provides the maximum benefits to the software testers. The software testers assign priority to each test case in test suite to ensure that the test cases with higher priorities are run earlier in the testing process [5].

Many faults might be left undiscovered because of the selection of too few test cases as in test case selection and minimization and execution of entire test suite might not be possible because of resource constraint. Other limiting factors such as budget and time constraints further emphasizes on the importance of test case prioritization [6].

Significant research has been done in prioritization area, many techniques have been developed to prioritize the test cases in order to increase the fault detection rate. Most of those techniques are coverage based and only consider whether a branch, statement or function has been covered by the test case. In coverage based techniques, it is assumed that the coverage will maximize the rate of fault detection but this relation between coverage and fault detection rate does not always hold. The test case with the better ability to find the faults may be given the low priority. There are only few existing test suite prioritization techniques which prioritize test cases based

0-7695-6347-3/17/$31.00 ©2017 IEEE
DOI 10.1109/FIT.2017.00017

on their ability to expose faults but the fault exposing potential of the test case cannot be calculated accurately.

In this paper, we propose a technique which prioritizes the test cases based on their ability to kill mutants generated using mutation testing. In mutation testing, a slight change is introduced in the original program by using any mutation operator, thus creating a mutant of a program. The test case that exposes the maximum number of those faults, i.e. killing the maximum number of mutants is said to be most efficient test case and will be given the highest priority.

We have compared our approach with existing technique in terms of priority lists using a example.

The remainder of this paper of this paper is organized as follows: Section 2 discusses the background knowledge of test case prioritization problem and mutation testing. Section 3 describes related work. Section 4 describes our proposed technique which incorporates mutation testing. Section 5 presents an example and Section 6 includes conclusion and possible future work.

II. Background

This section gives an overview of test case prioritization and mutation testing.

A. Test Case Prioritization

Test case prioritization is used to find an ideal ordering of the test cases so that the tester is provided with the maximum benefit. One testing benefit is increased *rate of fault detection reported as APFD,* which is the measure of how quickly faults can be detected by the test suite. [1] If maximum number of faults can be detected by minimum number of test cases and those test cases are among the top test cases of priority list then the rate of fault detection increases. Improved rate of fault detection provides the feedback on the system in earlier phases. A priority list can be used to decide when to stop testing in case of having not enough resources to execute all tests. Prioritization will also make sure that if, in any case, testing is halted prematurely, then most important tests, with higher priorities, will have been executed [7].

Test case prioritization was first introduced by Wong et al. [8] and was applied to the test cases already selected by the test case selection techniques. Harrold et al. [9] and Rothermel et al. [10] then extended the concept and proposed it in more general context.

Test case prioritization does not involve selection of test cases and rather allows all the test cases to be executed. However, Rothermel discussed that if elimination of test cases is acceptable then in some cases, test case prioritization can be used in conjunction with test suite minimization and test case selection, where it is applied to the selected subset of the test suite[10]. Test case prioritization is formally defined as:

Definition: The Test Case Prioritization Problem:

Given: T, a test suite; PT, the set of permutations of T; and a function from PT to the real numbers, f: PT → ℝ

Problem: To find $T' \in PT$ such that $(\forall T'')(T'' \in PT)(T'' \neq T')\ [f(T') \geq f(T'')]$

B. Mutation Testing

Mutation testing is a fault based technique which is used to measure the effectiveness of a test suite [11]. In mutation testing, we take the original program and introduce a slight change using mutation operators in the original code thus creating a mutant which represents a faulty version of the program.

In mutation testing, mutants are generated by applying mutation operators resulting in mutated copies of the original program. After the generation of mutants, in next step, these mutants are tried to be killed using the test data. If the mutant gives different output than the original program against the same test case, then the mutant is said to be killed by that test case. Each mutant is executed against each test case and it is monitored that whether it gives different output from the original program.

A mutant that is not killed by any test in test suite remains “alive”. Alive mutant can be killed by enhancing the test suite and adding more test cases which can kill those alive mutants. However, a mutant is considered equivalent to its parent program if there exists no test that causes original and mutant program to generate different outputs. Equivalent mutants always give same output as the original program and are not considered while calculating the fault detection ability of test suite.

III. Related Work

Elbaum [12] proposed two types of white box prioritization techniques:

i) Coverage-basedtechniques that prioritize test cases based on their coverage of elements of source code. These coverage based techniques are further divided into two categories; a) Total coverage based prioritization which assigns priorities to test case based on the total number of functions, branches or statements covered. b) Additional coverage based prioritization, which assigns priorities to test cases based on the additional entities, i.e., functions, branches or statements covered. The test case that covers maximum number of uncovered entities is given highest priority.

ii) Fault based techniques that prioritize test cases based on their fault exposing potential. The test cases are ordered in the descending order of their mutation score.

Kim and Porter [13] proposed history-based prioritizationthat uses the information from previous execution cycles to select the set of test cases. They applied RTS technique to test suite T that produced T’ in

the first step. After that in 2nd step they assigned selection probability to every test in T'. In third step, they selected a test case based on probabilities assigned in 2nd step and run it. The last step includes the repetition of step 3 until the testing time is exhausted. The selection probability was assigned to each test case based on its prior performance. They used test histories based on execution history, fault detection rate and program entities covered.

Marre´ and Bertolino [14] introduced a concept of spanning set of entities to be used as criterion to prioritize the test cases. They used the subsumption relationship between entities to find the spanning set, i.e., given two entities E and E', if every complete path that covers E also covers E', then E subsumes E', but there exist some entities that are unconstrained, i.e., en entity E is called unconstrained if it is not subsumed by any other entity E' without being itself subsumed by E, the smallest subset of such unconstrained entities is called a spanning set with a property that any subset of test cases that covers this subset of entities covers every entity in the program. Their technique was divided further into two approaches: Additional Spanning Statements and Additional Spanning Branches. The test case that covers the maximum number of uncovered spanning statements or branches is given the highest priority.

Wang [15] proposed a technique that prioritizes test cases based on the severity of faults detected. For each test case, the detected faults are enumerated and each fault is assigned a number based on its severity. The fault severities of those detected faults are then accumulated the test case that has the highest accumulated value is given the highest priority.

Lou [16] proposed a fault based technique, which exploits mutation testing to prioritize test cases. They used mutation testing in order to make mutants of the original code, which represent faulty versions. For each test case, they calculated its weight that represents the total number of mutants killed and the test cases are prioritized in descending order based on their weights.

Henard [17] evaluated all of the above techniques based on different subject programs and concluded that the additional branch coverage and additional spanning branches based prioritization yielded the highest APFD.

IV. Proposed Technique

In this section we describe our approach for test case prioritization using mutation testing. In mutation based prioritization, we will prioritize the test cases based on the number of mutants killed by test cases. Figure 1 depicts our approach where the ovals show the main activities and boxes represents the input/output associated with the main activities. Our approach consists of three main activities: mutants' generation, test cases to mutants killed mapping and test case prioritization. The following subsections describe each activity in detail.

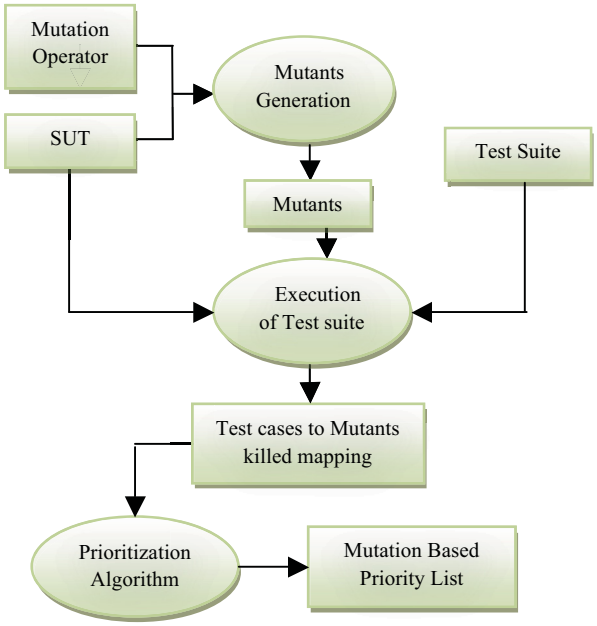

Figure 1. An overview of proposed approach

A. Mutants' Generation

In our approach we have one operator to generate mutants of the original program. The selected operator sees which similar operator is used in a condition and replaces it with other operators. For example, if ROR operator is selected, it generates 7 mutants against one condition in the original program having relational operator by replacing it with any 5 other relation operators in each mutated copy and 2 mutated programs are generated by replacing relational operator with "*true"* and *"false"* values.

For example, consider the following conditions in the original program:

```
if (marks > 80 && attendance >= 30)
   {
     grade = "A";
   }
```

Since there are 2 conditions in the decision so following 14 mutants will be generated for this program. An example of mutant generation is shown in Table 1.

B. Execution of Test Suite on SUT and Mutants

After creating these mutants, in next step, test data is used to kill these mutants. In this phase, we record which test case kills which mutant by executing the test cases on SUT and mutants. The above mutants will be executed one by one against each test case and if against any test case mutant produces different output than the original program then the test case will be said to have killed the mutant.

Mutants Killed = {1,2,3,4,5,6,7,8,9,10,11,12,13,14}

Test case	Killed mutants covered
1	2,3,5,6
2	1,3,4,6
3	9,10,12,13
4	2,3,4,7,8,9,11,12,14
5	2,3,4,5,7,9,10,13

Step 1. T4 covers the maximum number of killed mutants so it has been selected. Remaining mutants are {1,5,6,10,13}.

Step 2. Now that T4 has been selected as the first test, T5covers the most remaining killed mutants. Remaining mutants are {1,6}.

Step 3. Test T2 covers the maximum uncovered killed mutants. Now that all the entities have been covered, the remaining test cases are appended at the end of priority list. The prioritized list will be PrT= {T4, T5, T2, T1, T3}

Table 1. Example of Mutants

Original Condition	Mutated Condition
1. marks > 80	1. marks > = 80
	2. marks <80
	3. marks < = 80
	4. marks = = 80
	5. marks!= 80
	6. true
	7. false
2. attendance>=30	8. attendance > 30
	9. attendance < 30
	10. attendance < = 30
	11. attendance = = 30
	12. attendance!= 30
	13. true
	14. false

An example of the execution of test case on SUT and mutant is shown below.

Test case		Mutated Condition marks > 80	Original Condition marks < 80
Marks	**Attendance**	**Mutant Output**	**Original Output**
0	30	null	A

	Prioritized Test Selection		
Original Test No.	No. of uncovered killed mutants		
	Step 1	**Step 2**	**Step 3**
T1	4	2	1
T2	4	2	**2**
T3	4	2	0
T4	**9**	-	-
T5	8	**3**	-
Best Test	T4	T5	T2

Figure 2. Example trace of prioritization algorithm

Since the output of the original program is different than mutant program in the example, it is said to be killed by the test case. Similarly, all the mutants will be executed one by one against each test case and their output will be compared with the output of the original program generated against that particular test case. When all the mutants have been executed against each test case, a mapping between test cases and mutants will be obtained.

C. Applying Prioritization Algorithm

After collecting the above data, this data will be used as an input to the greedy prioritization algorithm, which is mutation based prioritization algorithm generating mutation based priority list. The algorithm in the first step, will find a test case that kills maximum number of mutants. This test case is assigned the highest priority and data is updated to indicate only the uncovered killed mutants. In next step, test case which covers maximum number of uncovered killed mutants will be added to the priority list, if two or more such test cases exist, then the test case is selected randomly and it will be removed from the test suite for further consideration, coverage information will also be updated to indicate the uncovered killed mutants. The process is repeated until all mutants have been covered by atleast one test case or the test suite has no unprioritized test case. If all the killed mutants have been covered and there are still unprioritized test cases left in the test suite, then they will be appended to the priority list randomly as these are the redundant test

cases. Figure 2 shows the working of mutation based test case prioritization algorithm.

V. Comparison of Proposed and existing Techniques

In this section, an example is used to show the difference between proposed technique and existing prioritization technique in terms of priorities assigned to test cases. The existing technique we have used for comparison is additional branch coverage based prioritization because of it being the strongest among existing prioritization techniques. The nextDate problem, having 40 branches out of which 28 are covered by the test suite and 63 ROR mutants out of which 44 are killed, is used as an example program. The test suite contains 10 test cases. The coverage of these test cases is shown in Table 2 and Table 3.

Table 2. Branch Coverage of test cases

Test case	**Branches covered**
T1	1,13
T2	2,4,6,8,10,12,16,18,20,22,26,29,31
T3	2,4,6,8,10,12,16,17,23
T4	2,4,6,8,10,12,16,18,20,21,23
T5	2,4,6,8,10,12,16,18,20,22,25,27
T6	2,4,6,8,10,12,16,18,20,22,26,29,32,33,36
T7	2,4,6,8,10,12,16,18,20,22,26,29,32,33,35
T8	2,4,6,8,10,12,16,18,20,22,26,29,32,33,35
T9	2,4,6,8,10,12,16,18,20,22,26,29,32,34,37,40
T10	2,4,6,8,10,12,16,18,20,22,26,29,32,34,37,39

Table 3. Mutant Coverage of test cases

Test case	**Killed mutants**
T1	22,24,25,28
T2	43,45,46,49
T3	29,31,32,35
T4	29,31,32,35
T5	36,38,39,42
T6	2,4,5,6,44,45,46,48,50,51,54,56
T7	1,2,5,7,8,10,12,14,50,51,54,56
T8	1,2,5,7,15,16,19,21,50,51,54,56
T9	2,4,5,6,43,45,47,48,51,53,54,55,57,58, 61,63
T10	1,2,5,7,8,10,12,14,43,45,47,48,51,53,54, 55,57,58,61,63

The priority lists are given in Table 4.

Table 4. Prioritized Test Suites

Prioritized Test suites
Branch coverage based: {T9, T1, T3, T5, T6, T2, T4, T7, T8, T10}
Mutation based: {T10, T6, T1, T3, T5, T8, T2, T4, T7, T9}

From the above prioritization techniques, it can be seen that branch coverage based prioritization may assign low priorities to those test cases which reveal maximum number of faults. In above example, branch coverage based prioritization assigned lowest priority to T10 which detected maximum number of changes in the original program, also in branch coverage based prioritization only first 8 test cases are enough to cover all branches so T8 is treated as redundant test case but in mutation based prioritization it is among high priority test cases.

VI. Conclusion and Future Work

A number of white box prioritization techniques have been proposed. Although existing prioritization techniques perform well, but most of these prioritization techniques are not fault based techniques. They all prioritize test suites based on some coverage criteria assuming that the coverage will maximize rate of fault detection. There are only few prioritization techniques which are fault based. The technique proposed in this research work involves the simple use of mutation testing in order to prioritize the test cases based on the number of additional killed mutants covered. The test case that exposes the maximum number of faults in the original program is given highest priority. We concluded that the test case that exposes maximum number of faults was given lower priority in branch coverage which is the major drawback of the technique. Mutation based prioritization addressed this drawback and assigned higher priorities to those test cases.

In future, more than one mutation operator can be used to generate more than one changes in each faulty version. Mutation testing can also be combined with some existing coverage based techniques by which we can solve the problem of random selection in case of tie.

VII. References

[1] Elbaum, S., Malishevsky, A. G., Rothermel, G. (2002). Test case prioritization: a family of empirical studies. IEEE Transactions on Software Engineering, 28(2), pp, 159–182.

[2] Burnstein, I. (2003). Practical software testing: a process-oriented approach. Springer-Verlag, New York, Inc.

[3] Garey, M.R., Johnson, D.S. (1979). Computers and Intractability: A guide to the theory of NP-Completeness. W. H. Freeman and Company: New York, NY.

[4] Rothermel, G, Harrold MJ. (1994). A framework for evaluating regression test selection techniques. Proceedings of the 16th International Conference on Software Engineering (ICSE), IEEE Computer Society Press, pp, 201–210.

[5] Tahvili, S., Afzal, W., Saadatmand, M., Bohlin, M. (2016). Towards earlier fault detection by value driven prioritization of test cases using fuzzy TOPSIS. Information and Software Technology.

[6] Srivastava, P. R. (2008). Test Case prioritization. Journal of Theoretical and Applied Information Technology (JATIT), BITS Pilani, India333031.

[7] Yoo, S., Harman, M. (2012). Regression testing: minimization, selection and prioritization: A survey. Software Testing, Verification and Reliability, 22(2), pp, 67–120.

[8] Wong, W. E., Horgan, J. R., London, S., Agrawal, H. (1997). A study of effective regression testing in practice, Proceedings of the 8th International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, pp, 264–275.

[9] Harrold, M.J. (1999). Testing evolving software. The Journal of Systems and Software, 47(2–3), pp, 173–181.

[10] Rothermel, G. Untch, R.H., Chu, C., Harrold, M.J. (1999). Test case prioritization: An empirical study. Proceedings of International Conference on Software Maintenance (ICSM), IEEE Computer Society Press, pp, 179–188.

[11] Ma, Y. S., Offutt, A. J., Kwon, Y. R. (2006). MuJava: A Mutation System for Java. Proceedings of the 28th international Conference on Software Engineering (ICSE), IEEE Computer Society Press, pp. 827–830.

[12] Elbaum, S.G., Malishevsky, A.G., Rothermel, G(2001). Prioritizing test cases for regression testing. Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2000), ACM Press, 2000; 102–112.

[13] Kim, J., Porter, A. A. (2002). A history-based test prioritization technique for regression testing in resource constrained environments. Proceedings of the 24th International Conference on Software Engineering (ICSE), ACM Press, pp, 119–129.

[14] Marre, M., Bertolino, A. (2003). Using spanning sets for coverage testing. IEEE Transactions on Software Engineering; 29(11), pp, 974–984.

[15] Y. Wang, X. Zhao, and X. Ding. (2015). An effective test case prioritization method based on fault severity. In International Conference on Software Engineering and Service Science, pp, 737–74.

[16] Y. Lou, D. Hao, and L. Zhang. (2015). Mutation-based test-case prioritization in software evolution. In Proc. ISSRE, pages 46–57.

[17] Henard, C., Papadakis, M., Harman, M., Jia, Y. and Traon, Y.L. (2016). Comparing White-box and Black-box Test Prioritization. In Proceedings of the 38th International Conference on Software Engineering (ICSE). ACM.