International Journal of Applied Engineering Research ISSN 0973-4562 Volume 11, Number 1 (2016) pp 57-63

An Approach for Fault Detection in Software Testing Through Optimized Test Case Prioritization

D. Vivekananda Reddy
Assistant Professor,
Department of Computer Science and Engineering,
S V University College of Engineering, Tirupati,
Andhra Pradesh, India.
E-mail : svuvivek@gmail.com

Dr. A. Rama Mohan Reddy,
Professor,
Department of Computer Science and Engineering
S V University College of Engineering, Tirupati,
Andhra Pradesh, India.
E-mail : ramamohansvg@yahoo.com

Abstract
The main objective of the proposed method is to implement an effective software testing without discarding the provided test cases. So the idea of test case optimization is avoided in designing the proposed method. As we find that the factors considered need to strengthened to obtain the prioritized test cases. From that perspective, a detailed study on the behavior of the requirements is conducted and some conclusions are made. According to the implementation complexity, the time required for implementation can also be considered as a factor. The factor mainly comes into play once a delay in implementation occurs. Thus we consider implementation delay as a factor that affects the test case prioritization. The other major addition is that the feasibility of each requirement after a testing cycle. As the testing process proceeds the necessity of each requirement will be questioned, i.e., some requirement will become non important after a set of testing cycles. These requirements can be identified by considering the completeness factor and traceability factor. The proposed method uses the following factors over the requirement to achieve the objective, Customer's priority, requirement changes, complexity in implementation, fault impact, completeness, traceability, implementation delay and requirement feasibility. The proposed method is based on optimized test case factors, after each testing cycle, there may be chances of alteration of values in the eight factors that are considered for test case prioritization. Based on these values the test case weight (TCW) is generated for each test case. So by optimizing the test case factors after each cycle, we get optimized TCW. According to the definition of the proposed approach, we use a comparison result of this TCW and optimized TCW for test case prioritization. The Optimized TCW is introduced because after each optimization the number of requirements reduces and hence that result in the change value of TCW for each test case.

Keywords: Test Case Weight, Test Case Prioritization, Fault Detection.

Introduction

Software testing is a thorough arrangement of activities carried out with the expectation of discovering mistakes in software. It is one activity in the development procedure of software focused for assessing a software item, for example, system, subsystem & features (e.g. functionality, performance & security) against a given arrangement of framework requirements. Also, software testing is the process of validating & verifying that a program functions properly [11]. Numerous researchers have demonstrated that software testing is a standout amongst the most fundamentally essential periods of the software development life cycle, & consumes significant resources in terms of effort, time & cost. Regression testing is a kind of software testing that intends to ensure the changes which incorporates improvements, corrections of errors, optimization & deletion of existing features [8]. These changes may be the reason for the system to work improperly. Therefore, Regression Testing turns into important in the software testing procedure. This prompts that regression testing in which all the tests in the accessible programs or suite ought to be re-executed. In this manner experiencing extra cost, time & resources. Several research has demonstrated that no less than 50% of the total software cost consisting of testing activities [9].

Test case prioritization (TCP) methods are a vital method implemented in regression testing. TCP is a scheduled test cases for regression testing in an execution order as indicated by some paradigm that endeavors to expand some objective function [12]. For instance, testers may wish to schedule test cases in an order that accomplishes code coverage at the speediest rate conceivable, exercises included, in order of expected frequency of utilization, or exercises subsystems in an order that reflects their past tendency to fail [14]. At the point when the time required to perform all test cases in a test suite is short, test case prioritization may not be financially savvy, it might be more convenient just to schedule test cases in any manner. When the time essential to execute all test cases in the test suite is adequately long, though, test case prioritization may be advantageous [7]. The reason for this prioritization in TCP is to improve the probability that if the test cases are utilized for regression testing in the given direction, they will more firmly meet some target than they would if they were executed in some other direction [7]. TCP place in order the test cases relying upon business effect, significance & frequently used functionalities. Choice of test cases in view of the priority will fundamentally decrease the regression test suite [8].

One main objective of test case prioritization is to improve the rate of fault detection; that is, to discover most flaws & as early as possible. Discovering flaws prior can improve early defect settling & at last lead to a prior conveyance [10]. Prioritization has regularly been employed into the test suites that take days or weeks to run; on the other hand, with agile development procedures, turning out to be more predominant in the industry, the prospective for prioritization methods to

have an effect is expanding [3]. Moreover, prioritizing test cases give the chance to make the most of some performance goals or efficiency. One of the execution objectives may be the rate of reliance detection among the deficiencies [13]. At the time of software testing, a pragmatic experiences exhibit that independent flaws can be directly detected & removed, but mutually dependent faults can be uprooted if & only if the main deficiencies have been uprooted. That is, dependent flaws may not be instantly uprooted, & the flaws deletion procedure falls behind the flaws discovery procedure [4]. In spite of the fact that test case prioritization approaches have extraordinary advantages for software test engineers, there are still outstanding most important research problems that ought to be addressed. The illustrations of the significant research problems are: (a) current test case prioritization approaches overlook the practical weight factors in their ranking algorithm (b) existing methods have an ineffective weight algorithm & (c) those procedures are absent of the automation in the time of prioritization procedure.

There is a need to maximize the effectiveness of the testing resources via a scheme which: (1) is economical (does not add significant overhead to team); (2) improves perceived software quality; and (3) identifies the more severe faults earlier. Here, we have projected a new approach to test case prioritization to resolve the issues we have discussed earlier.

Contributions to The Paper

- Studied and analyzed fault detection in system level testing and taken measures to handle fault detection
- Designed and implemented a method for fault detection and correction based on optimized test case prioritization
- Analysis of the implemented method over different experimental criteria to prove the effectiveness

Related Work

R. Krishnamoorthi, and S. S. A. Mary [1] have proposed to advance a model for system level test case prioritization (TCP) from software requirement specification to enhance user satisfaction with quality software that can likewise be low cost & to enhance the rate of severe flaw recognition. The proposed model ordered the system test cases depend on the six elements: customer priority, changes in prerequisite, implementation complexity, completeness, traceability & effect of fault. The proposed prioritization method was confirmed with two distinctive approvals procedures & was tested in three stages with student projects & two arrangements of industrial assignments & the outcomes demonstrate convincingly that the proposed prioritization method enhances the rate of extreme deficiency recognition.

H. Do et al. [2] have carried out a series of experiments to evaluate the impacts of time constraints on the costs & advantages of prioritization methods. Their first test was controlled time constraint levels & demonstrated that time constraints do play a critical part in deciding both the cost-effectiveness of prioritization & the relative cost-benefit trade-offs among the methods. Their second trial duplicates the first test, controlling for a few threats to validity, comprising numbers of deficiencies present & exhibited that the outcomes generalize to this wider context. Their third test was controlled the quantity of the flaws present in programs to look at the impacts of defectiveness levels on prioritization & demonstrated that faultiness level influences the relative cost-effectiveness of prioritization methods. All these acquired outcomes gave a few recommendations about when & when not to prioritize, what methods to utilize & how dissimilarities in testing procedures may relate to prioritization cost effectiveness.

S. E. Z. Haidry, and T. Miller [3] have presented a group of test case prioritization methods that utilizes the dependency information from a test suite to prioritize that test suite. The way of the methods conserves the dependencies in the test arrangement. The hypothesis of this work was that dependencies between tests were representations of connections in the system under test, & executing complex interactions prior was prone to increase the flaws detection rate, contrasted with random test orderings. Empirical assessments on six frameworks built toward industry use exhibit that these methods increase the rate of flaw detection contrasted with the rates accomplished by the untreated order, random orders, & test suites ordered utilizing existing "coarse-grained" methods in view of functional coverage.

M. I. Kayes [4] has presented another metric for evaluating rate of fault dependency recognition & an algorithm to prioritize test cases. Utilizing the new metric, the viability of this prioritization was presented comparing it with non-prioritized test case. Investigation demonstrated that prioritized test cases were more efficient in identifying dependency among faults.

H. Mei et al. [5] have proposed a method to give the priority to the test cases in the absence of coverage information that works on Java programs tested under the JUnit framework-an increasingly popular class of systems. The proposed method, JUnit test case Prioritization Techniques work in the Absence of coverage information (JUPTA), analyzed the static call graphs of JUnit test cases & the program under test to evaluate the capacity of every test case to accomplish code coverage, & after that schedules the order of these test cases depend on those evaluations. Outcomes demonstrated that the test suites made by JUPTA were more compelling than those in arbitrary & untreated test orders regarding fault-detection effectiveness.

S. Sampath et al. [6] have formalized the idea of consolidating different criteria into a hybrid. They had detailed three hybrid combinations, Rank, Merge, & Choice, & exhibit their convenience in two ways. First, they recast, in terms of their formulations, others' previously reported work on hybrid criteria. Second, they utilize their earlier results on test case prioritization to make & assess new hybrid criteria. Their outcomes proposed that hybrid criteria of others can be depicted utilizing their Merge & Rank formulations, & that the hybrid criteria they developed most often outclassed their essential individual criteria.

The Proposed Test Case Prioritization Algorithm

In this section the detailed description of the proposed test case prioritization algorithm is presented. The main objective of the proposed method is to implement an effective software testing process without discarding the provided test cases. So the concept of test case optimization is avoided in designing the proposed method. Even though the method uses an approach on optimization through the requirements. From the past studies, we came to the observation that, instead of

reducing the number of test cases the requirement can be optimized. This has given the core idea of proposing the method for test case prioritization, which is inspired from the method referred in. In the method that is proposed for test case prioritization, there are mainly three processes. The first process includes the setting up the test cases and requirements. The second one refers to the calculation of test case weight based on the prioritization factors and finally optimizing the test case weights with genetic algorithm. The proposed method adapts six prioritizing parameters from and as from the detailed analysis two more parameter are considered for the proposed approach. Thus total of eight prioritizing parameter are used to list the requirement table. This table is then used to map the test cases for evaluating the test case weights. The hypothesis behind the proposed approach is that after each iteration of the testing process, there will be change in requirements. The change in requirement can result in adding or removing the current requirements. So a requirements optimization could result in improved testing efficiency by altering the test case weights.

A. Process of Defining the Prioritization Factors

Based on the projects that are selected for evaluating the proposed approach, the prioritization parameters have to be derived. For the proposed approach we consider eight parameters. There needs to be a thorough analysis for defining the parameters for each requirement. Based on the result of the analysis, we consider the following factors in our proposed algorithm: (1) customer priority of requirements (CP), (2) implementation complexity (IP), (3) requirement changes (RC), (4) fault impact of requirements (FI), (5) completeness (CO) and (6) traceability (TR). According to the implementation complexity, the time required for implementation can also be considered as a factor. The factor mainly comes into play once a delay in implementation occurs. Thus we consider implementation delay as a factor that affects the test case prioritization. The other major addition is that the feasibility of each requirement after a testing cycle. As the testing process proceeds the necessity of each requirement will be questioned, i.e., some requirement will become non important after a set of testing cycles. These requirements can be identified by considering the completeness factor and traceability factor. Thus the factors (7) implementation delay (ID) and (8) requirement feasibility (RF) are added to the suite. Let us discuss the parameters in detail,

i. Customer-assigned priority (CP)

It is a measure of the significance of a necessity for the customer. The customer allocates the values for every prerequisite ranging from 1 to 10 where 10 show the most astounding customer priority. Reasoning: An attention to client necessities for development has been displayed to enhance customer observed value & satisfaction. In this way, the necessities that would be of the highest significance to the client, ought to be tested early & completely to enhance customer satisfaction.

ii. Implementation complexity (IC)

It is a subjective measure of how troublesome the development team observes the implementation of prerequisite to be. Each necessity is examined & value ranging from 0 to 10 is allocated by the developer depend on its implementation complexity; the larger value shows higher complexity. Reasoning: Requirements with high implementation complexity is relied upon to have a higher number of shortcomings.

iii. Requirement changes (RC)

It depends on the total number of times a requirement has been changed in the development cycle concerning its starting date & the value ranges from a scale of 1 to 10 is allotted by the developer. If the requirement is changed more than 10 times, the unpredictability values for all the requirements are quantified on a 10-point scale. The necessity changes RCi for a prerequisite i can be calculated by dividing the number of changes for prerequisite i to the highest number of changes for any prerequisite in the midst of all the project prerequisites. If ith requirement is changed N times & the maximum number of requirement changes amongst all the requirements is M then the requirement change of i, RCi can be computed as,

$$RC_i = (N / M) * 10 \qquad (1)$$

Reasoning: Roughly 50% of all shortcomings recognized in the project are the errors presented in the requirement phase. The noteworthy element that causes the disappointment of the project is attributed to varying necessities.

iv. Fault impact of requirements (FI)

It permits the development team to recognize the necessity which has had client reported disappointments. As a system progresses to several versions, the developers can utilize the earlier data gathered from versions to recognize necessities that are inclined to be error. FI is depends on the number of field failures & in-house failures. FI is considered for those necessities that have already been in a released product. In this work, we propose to compute the fault effect of a necessity, depend on the severity of the fault recognized in the previous run. Severity of the fault 1, 2, 3, 4 & 5 corresponds to the severity values 25, 24, 23, 22 & 21. If a prerequisite i with the set of t test cases, discovered the set of d faults of the set of severity V, then the severity Si of the prerequisite i is calculated as,

$$S_i = \sum_{x=1}^{t} \sum_{y=1}^{d} Vx, y \qquad (2)$$

and if S = {Si for all i = 1. . .n} where n is the aggregate number of necessities, is the arrangement of all severities of every prerequisite i, then the fault impact FI of a necessity i is calculated as,

$$FI_i = \frac{S_i}{\max(S)} * 10 \qquad (3)$$

Reasoning: Test effectiveness can be enhanced by concentrating on the function that is expected to contain higher numbers of flaws.

v. Completeness (CO)

One component of ‘‘prerequisites completeness” is a test to confirm that every necessity, individually, is complete (i.e., states a prerequisite in terms of a function to be performed (‘‘what”), the measure of success (performance requirement or ‘‘how well”), the conditions under which the function is to be performed, & any limitations which influence the possible solution, for example, an interface constraint. Every necessity is analyzed for its fulfillment & value ranging from 0 to 10 is allotted by the client, when that requirement is considered for reuse. Reasoning: Customer satisfaction, for example, the quickness of the response of the software to the client request be enhanced by considering completeness of the prerequisite.

vi. Traceability (TR)

Requirements traceability talk about the “capacity to follow the life of a necessity, in both forwards & backwards direction, i.e., from its starting point, through its improvement & specification, to its consequent deployment & utilize, & through periods of ongoing refinement & iteration in any of these stages. Every necessity is examined for its traceability & value ranging from 0 to 10 is allotted by the tester when that prerequisite is considered for reuse. Reasoning: The quality of the software can be enhanced by considering the traceability of the requirement.

vii. Implementation delay (ID)

According to the implementation complexity, the time required for implementation can also be considered as a factor. The factor mainly comes into play once a delay in implementation occurs. Thus we consider implementation delay as a factor that affects the test case prioritization.

viii. Requirement feasibility (RF)

The other major addition is that the feasibility of each requirement after a testing cycle. As the testing process proceeds the necessity of each requirement will be questioned, i.e., some requirement will become non important after a set of testing cycles

B. The Step by Step Process for Implementing the Proposed Test Case Prioritization

Step 1: initially the set of requirements are accepted by the system and then the set of test cases. The test cases are mapped into the requirements to evaluate the faults.

The set of requirements are given by,

$$R=\{r_1,r_2r_3,...r_n\} \qquad (4)$$

Where n is the total number of requirements. Now the set of test cases are given as,

$$T=\{t_1,t_2t_3,...t_m\} \qquad (5)$$

Where m is the total number of test cases.

Step 2: The next step is to create a requirement table with prioritization values. Each prioritization values are assigned to each of the requirements based on the definition of each prioritization factor.

Table 1: Requirements Table

	CP	IC	CR	FI	CL	TR	IP	FR
r1	9	8	5	2	7	6	2	5
r2	8	7	4	1	5	5	4	5
r3	9	7	3	2	6	7	3	4
r4	6	6	3	1	8	5	2	4
r5	7	5	2	2	7	5	1	4
r6	3	4	2	2	7	9	4	3
r7	6	6	3	1	9	7	3	3
r8	3	8	4	1	8	5	3	1
r9	7	9	5	2	6	4	3	4
r10	8	8	2	2	5	7	2	5

The above table 1 represents the prioritization values for ten requirements. These are based on how the code responds to the listed requirements.

Step.3. Once the requirements table is filled with value, we calculate requirement factor value for each requirement listed in the table. The requirements are calculated based on the following equation,

$$RFV_i = \left(\sum_{i=1}^{8} p\_factor_i \right) \Big/ 8 \qquad (6)$$

So for the requirements table depicted in Table.1 the RFV value can be listed as,

Table 2 : Requirement table with RFV values

	CP	IC	CR	FI	CL	TR	IP	FR	RFV
r1	9	8	5	2	7	6	2	5	5.500
r3	9	7	3	2	6	7	3	4	5.125
r9	7	9	5	2	6	4	3	4	5.000
r2	8	7	4	1	5	5	4	5	4.875
r10	8	8	2	2	5	7	2	5	4.875
r7	6	6	3	1	9	7	3	3	4.750
r4	6	6	3	1	8	5	2	4	4.375
r6	3	4	2	2	7	9	4	3	4.250
r5	7	5	2	2	7	5	1	4	4.125
r8	3	8	4	1	8	5	3	1	4.125

Step.4. After the calculation of RFV values for each requirement, the set of test cases are selected and mapped based on the requirements. Then the test case weights are calculated accordingly. I.e., if a test case t maps I number of requirements, the test case weight is calculated as,

$$TCW_j = \left[\sum_{x=1}^{i} RFV_i \Big/ \sum_{y=1}^{n} RFV_y \right] * i / n \qquad (7)$$

Here TCWj is the test case weight of the jth test case. In similar manner we calculate the test cases of each test case. Normally the test case is prioritized based on their TCW values.

Step.5. Now we look for the change in requirements after the testing process. If there is now changing in requirement, the requirements are passed to the defined genetic algorithm. The fitness function for the genetic algorithm is the difference between RFV values of the new set of requirements to the existing set of requirements.

$$RFV_{fit} = RFV_j - RFV_i \qquad (8)$$

Here RFVi is derived from the initial set of requirements and RFVj is derived from the genetic algorithm based requirements. If fitness is positive, then we use the new set of requirements for the test case prioritization.

Experimental Analysis

This section explains about the experimental evaluation of the proposed test case prioritization algorithm. The method is evaluated based on the false detection rate. The prioritized test cases are selected and executed on selected program code and the severity of the fault is evaluated. The method is designed and developed in Java language with JDK 1.7.0. Let us consider the experimental evaluation in detail scale.

Validation of the proposed prioritization algorithm is carried out through two different validation techniques. The first validation technique is based on the analysis of the faults detected for a product and the second validation technique is based on the analysis of the number of test cases executed to detect the faults. We present these validation techniques in the following Sections. Even though there are many validation techniques, we mainly concentrate on the severity of the fault detected. The severity levels can be defined as follows, For the purpose of analysis, each failure is assigned a severity value (SV) on a 10-point scale as shown in Table.3

A. Severity Levels

Highly severe (Very High (VH) - Severity 1): Severity 1 had been allotted to a failure that could cause loss of life or property and/or loss of a system. SV of 32 had been assigned for the Severity 1 failures.

Severe (High (H) - Severity 2): Severity 2 had been allotted to a failure when a customer can no longer use the product and/or testing must cease until the defect causing the failure is fixed. SV of 16 had been assigned for the Severity 2 failures.

Medium (Medium (M) - Severity 3): Severity 3 had been allotted to a failure when there is a work around for failure & the product can be used with the work around. SV of 08 had been assigned for the Severity 3 failures.

Less severe (Low (L) - Severity 4): Severity 4 had been allotted to a failure for which a fix can be done in later versions. SV of 04 had been assigned for the Severity 4 failures.

Least severe (Very Low (VL) - Severity 5): Severity 5 had been allotted to a failure for which a fix can be done in later versions or not done at all. SV of 02 had been assigned for the Severity 5 failures.

Total severity of faults detected (TSFD) is the summation of severity values of all flaws detected for a product. TSFD of *f* number of faults detected in a program is computed as follows,

$$TSFD = \sum_{i=1}^{f} SV_i \qquad (9)$$

The Average severity of faults detected (ASFD) for each requirement with f defects is calculated as follows,

$$ASFD = \sum_{j=1}^{f} SV_j \Big/ TSFD \qquad (10)$$

ASFD is utilized to investigate the efficiency of the prioritization algorithm & we propose to experimentally test whether the requirement with a higher calculated RFV really has higher average severe faults when the product is system tested or utilized by the client. Regression experiments have been conducted to determine our above mentioned claim. The total percentage of severe faults recognized for the requirement are mapped to the RFV for that requirement.

B. Efficiency of The Prioritization Method

i. Analysis Based ASFD

We conducted a study to test the efficiency of the proposed approach. In the study, we tested four projects, for the analysis, which is coded in the Java language. Each program contains about 2000 LOC. A set of 15 requirements are subjected for the evaluation. The programs are then system tested using a testing tool known as QTP (quick test professional). For each project, in programmer perspective, we assigned values for the factors RC and IC. Then, as a customer, assigned values for the factor CP. Then on the tester's perspective, we have a written test cases for the execution of each requirement and then been also asked to assign the values of requirement factors CT, TR and FI. The priority factor for each requirement can be listed as follows. Once the priority factors are assigned to each of the requirements, then the RFV value for each project with respect to the requirements are calculated. The RFV values are then selected for the TCW calculation as defined the above sections. The initial level of evaluation of the proposed approach is based on the ASFD and the RFV range. Then we evaluate the percentage of TPFD over the different percentage levels of requirement.

Table 3: Requirement and priority factors for project 1

Requirements	CP	IC	CR	FI	CL	TR	ID	RF
r1	9	8	5	2	7	6	2	5
r2	8	7	4	1	5	5	4	5
r3	9	7	3	2	6	7	3	4
r4	6	6	3	1	8	5	2	4
r5	7	5	2	2	7	5	1	4
r6	3	4	2	2	7	9	4	3
r7	6	6	3	1	9	7	3	3
r8	3	8	4	1	8	5	3	1
r9	7	9	5	2	6	4	3	4
r10	8	8	2	2	5	7	2	5

Similar to the table.3, the requirements and priority factors for all the 4 projects are assigned. Then RFV values are calculated. After calculating the RFV values, the requirements are mapped with the test cases and executed for the system level testing. After the testing the fault detection rate are calculated. Based on that the ASFD values are calculated for each project.

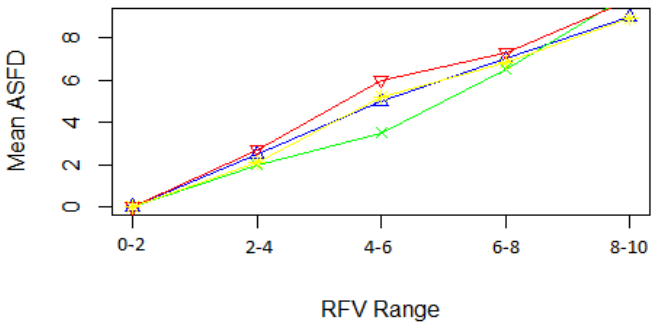

Figure 1: Evaluation based on RFV and ASFD

The tester group tests the projects using the testing tool they are familiar with. When the testers identified a failure, based on the discussion with developers, we assign a severity level to the failure. For each project an average of 10 faults are identified. These faults are analyzed to determine the requirements it maps to. The RFV and ASFD for each of the 10 requirements and the TSFD for each project is computed. RFV is divided into five ranges of categories based on the values of RFV. The requirements are grouped into one of these five ranges. A lower RFV value indicates a lower priority for the particular requirement to be tested. As for the figure 1 it is evident that as the RFV range increases the mean of the ASFD value also increases. On the basis of that inference, if the requirements possess high RFV value, then the chances of fault detection are also higher. So the priority factor affects the responses of the testing process. Since the test case weight is calculated with the help of RFV values, we can state that the prioritized test cases can make an effect on the testing process. In the next section, we produce a more elaborated analysis of the proposed approach, which can give more insight to the effectiveness of the proposed approach.

ii. Analysis Based TSFD

The next major evaluation aspect of the proposed method is to analyze the total severity over the requirements. The total severity is the measure that states the amount of fault identified by the tester over testing with the test cases that maps the requirement specified. In this experiment, we select three methods for testing the programs. In the first method, we select the test cases in a random manner, which is a common testing method. Secondly, we use the existing prioritization method and finally the proposed optimal prioritization method. The experiments are conducted in similar environments.

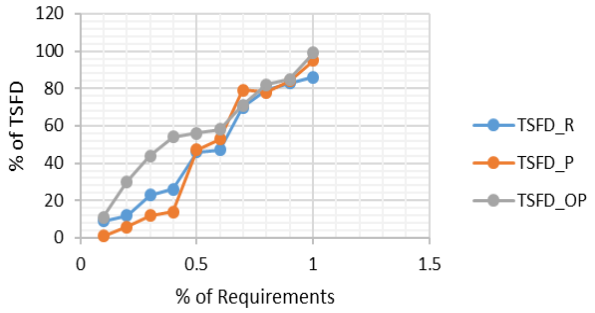

Figure 2: Project 1 responses

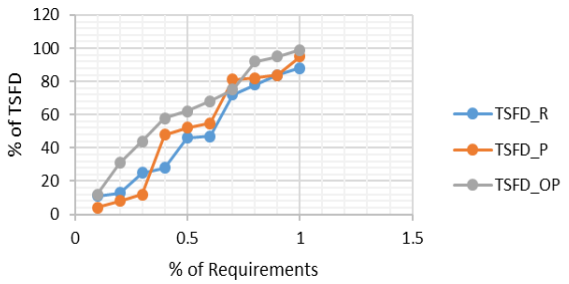

Figure 3: Project 2 responses

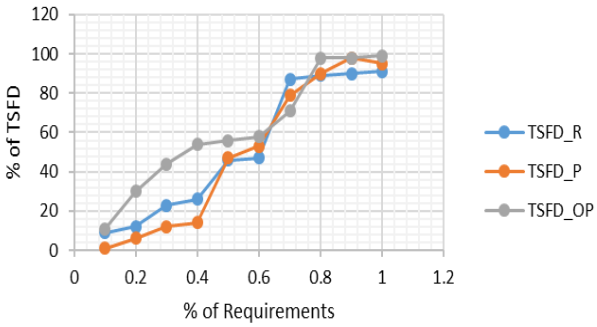

Figure 4: Project 3 responses

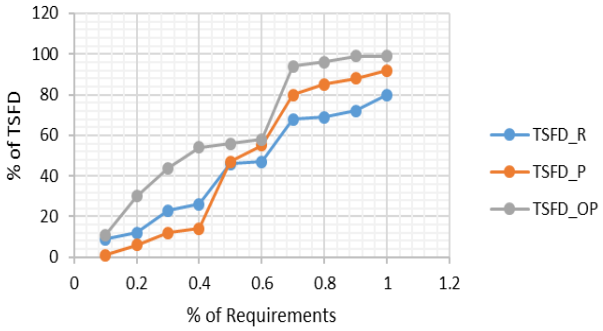

Figure 5: Project 4 responses

The figure above is plotted over the percentage of requirements to percentage TSFD values. The value TSFD_R represents the TSFD values obtained from the random testing method. The value TSFD_P represents the TSFD values from prioritization method and finally the value TSFD_OP represents the TSFD values from the proposed optimal prioritization method. The figures 2 from 4 represents the responses of total severity of four different projects tested under three testing methods. Analyzing all these figures, we can understand that all the three methods are able to detect the faults on the increase of requirement. But the optimal prioritization method seems to be more consistent with the identification of the faults. The responses of the optimal prioritization method are linear over the requirements and TSFD values. Thus we can state that, by optimizing the requirements, we can improve the efficiency of testing process significantly.

Conclusion

In this paper, we propose an optimal prioritization technique for requirement based System level test cases to improve the rate of fault detection in regression testing. Here, we select a total of eight priority factors prioritizing the regression test cases. The algorithm uses a genetic algorithm based optimization on the requirements to improve the test case prioritization. The optimal prioritization algorithm is validated through two phases of analysis with four different program codes. Results indicate that the proposed technique leads to an improved rate of detection of severe faults in comparison to random ordering of test cases and the prioritized test cases. Over the different analysis based on ASFD and TSFD, we came to the conclusion that our proposed prioritization technique is greater than that for a randomly chosen order and the basic prioritization technique.

References

[1] R. Krishnamoorthi, and S. S. A. Mary, “Factor oriented requirement coverage based system test case prioritization of new and regression test cases,” Information and Software Technology, Vol. 51, No. 4, pp. 799-808, 2009.

[2] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, “The effects of time constraints on test case prioritization: A series of controlled experiments,” IEEE Transactions on Software Engineering, Vol. 36, No. 5, pp. 593-617, 2010.

[3] S. E. Z. Haidry, and T. Miller, “Using dependency structures for prioritization of functional test suites,” IEEE Transactions on Software Engineering, Vol. 39, No. 2, pp. 258-275, 2003.

[4] M. L. Kayes, “Test case prioritization for regression testing based on fault dependency,” Proc. 3rd International Conference In Electronics Computer Technology (ICECT), 5, pp. 48-52, 2011.

[5] H. Mei, D. Hao, L. Zhang, J. Zhou, and G. Rothermel, “A static approach to prioritizing junit test cases,” IEEE Transactions on Software Engineering, Vol. 38, No. 6, pp. 1258-1275, 2012.

[6] S. Sampath, R. Bryce, and A. M. Memon, “A uniform representation of hybrid criteria for regression testing,” IEEE Transactions on Software Engineering, Vol. 39, No. 10, pp. 1326-1344, 2013.

[7] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Test case prioritization: An empirical study,” Proc. IEEE International Conference in Software Maintenance (ICSM'99), pp. 179-188, 1999.

[8] T. Muthusamy, and K. Seetharaman, “Effectiveness of Test Case Prioritization Techniques Based on Regression Testing,” International Journal of Software Engineering & Applications, Vol. 5, No. 6, pp. 113, 2014.

[9] H. Srikanth, and L. Williams, L., “Requirements Based Test Case Prioritization,” IEEE Trans. on Software Engineering, vol. 28, 2002.

[10] M. J. Arafeen, & H. Do, “Test case prioritization using requirements-based clustering,” Proc. IEEE Sixth International Conference in Software Testing, Verification and Validation, pp. 312-321, 2013.

[11] P. R. Srivastava, “Test case prioritization,” Journal of Theoretical and Applied Information Technology, Vol. 4, No. 3, pp. 178-181, 2008.

[12] S. Elbaum, A. G. Malishevsky, and G. Rothermel, “Test case prioritization: A family of empirical studies,” IEEE Transactions on Software Engineering, Vol. 28, No. 2, pp. 159-182, 2002.

[13] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, “Test case prioritization: An empirical study,” Proc. IEEE International Conference on in Software Maintenance, pp. 179-188, 1999.

[14] S. Elbaum, G. Rothermel, S. Kanduri, and A. G. Malishevsky, “Selecting a cost-effective test case prioritization technique,” Software Quality Journal, Vol. 12, No. 3, pp. 185-210, 2004.