I.J. Modern Education and Computer Science, **2016, 4, 33-45**
Published Online April 2016 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijmecs.2016.04.05

Test Case Prioritization based on Fault Dependency

Samia Jafrin
American International University-Bangladesh, Department of Computer Science, Dhaka, Bangladesh
Email: samia.jafrin.13@gmail.com

Dip Nandi, Sharfuddin Mahmood
American International University- Bangladesh, Dhaka, 1213, Bangladesh
Email: {dip.nandi,smahmood}@aiub.edu

***Abstract*—**Software testers should prioritize test cases so that important ones are run earlier in the regression testing process to reduce the cost of regression testing. Test case prioritization techniques schedule test cases for execution in an order that improves the performance of regression testing. One of the performance goals i.e. the fault detection rate, measures how quickly faults are detected during the testing process. Improved rate of fault dependency detection can provide faster feedback on software and let developers debug the leading faults at first that cause other faults to appear later. Another performance goal i.e. severity detection rate among faults, measures how quickly more severe faults are detected earlier during testing process. Previous studies addressed the second goal, but did not consider dependency among faults. In this paper an algorithm is proposed to prioritize test cases based on rate of severity detection associated with dependent faults. The aim is to detect more severe leading faults earlier with least amount of execution time and to identify the effectiveness of prioritized test case.

***Index Terms*—**Software testing, Regression testing, Test case prioritization, Fault dependency, Software quality.

I. Introduction

A software product, once developed, has a long life and evolves through numerous additions and modifications based on its faults, changes of user requirements, changes of environments, and so forth. With the evolution of a software product, assuring its quality, is becoming more difficult because of numerous release versions [1]. Users expect to get a new and better quality software version than before. In some cases, the quality of software becomes worse than before because of the added or modified features which create additional faults into the existing product as well as the newly modified version. For assuring a good quality software, testing is mandatory.

Evaluating a system with the intention of finding faults is known as Software Testing. Once system has been developed, it must be tested before implementation. It is oriented towards Error-detection [2]. Software testing is one of the major and primary techniques for achieving high quality software. It is done to detect the presence of faults, which cause software failure. It can also be referred as the process of verifying and validating software application or program to ensure that software meets the technical as well as business requirements as expected [3] [4].

For testing, a software engineer often use test cases. A test case is a set of conditions or variables and inputs that are developed for a particular goal or objective to be achieved on a certain application to judge its capabilities or features. It might take more than one test case to determine the true functionality of the application being tested. Every requirement or objective to be achieved needs at least one test case. Some software development methodologies like Rational Unified Process (RUP) recommend creating at least two test cases for each requirement or objective; one for performing testing through positive perspective and the other through negative perspective.

Regression testing is a kind of software testing that focuses on selective retesting through various versions of a software system [5]. The following is the formal definition of regression testing used by IEEE.

"Selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirement."[6]

Another popular software testing technique is Test Case Prioritization. In this technique, each test cases are assigned a priority. Priority is set according to specific criterion and test cases with highest priority are scheduled first. Another criterion may be the rate at which fault is detected. [7]

The goal of this research is to find a metric to quantify the rate of dependency detection among faults and provide an algorithm that prioritizes the test cases in an order that has improved dependency detection rate compared to non-prioritized test cases. By the definition of the test case prioritization, problem represents a quantification of such goals.

Test case prioritization is a strategy for improving regression testing, an expensive but necessary process to validate software systems. Despite its use by practitioners, to date, there has been no work regarding how to incorporate the dependent faults as well as independent faults severities into any of the strategies proposed so far. So in [8], researchers worked over both the independent and fully dependent faults. But, we think test case prioritization considering fault dependency is incomplete.

In the paper [8], researchers proposed an algorithm to measure effectiveness of test case prioritization in regression testing and a prioritization technique which can be used to improve the fault detection process for regression testing. In [8], researchers only considered the dependent faults which are fully dependent on other leading faults. But did not consider the fact that, there can be faults that are not fully dependent rather mutually dependent on more than one fault. The detection of the independent and fully dependent faults is covered simultaneously in this software testing approach. But, an efficient example needed to be set along with the independent and dependent faults (both fully & partially) to make an efficient approach. Further, a sizable performance gap can be seen as prioritization is done only with taking the fully dependent faults into consideration, not the partially dependent faults.

Hence, in order to overcome these issues, in current research paper, we will extend this research work to investigate the above mentioned weaknesses and will provide an alternative or improved version of prioritization technique including different methods of fault detection methods. A thorough research in this field, may help to detect faults as early as possible.

In this paper, we will extend the research of prioritizing test cases considering fault dependency mentioned in [8], as we think fault dependency consideration is incomplete there. In this paper, our goal is to include the fault dependency considering both fully & mutually dependent faults for doing complete test case prioritization.

II. Related Works

In this section we are going to discuss about Software testing. Then we will focus about the importance of software testing and test cases. Finally we will narrow down the topic into test case structures, test case designing and test cases. We will also focus on test case prioritization technique, existing techniques for test case prioritization, its problems and our focus area.

A. Software Testing

Every software product has a target audience. When an organization develops or otherwise invests in a software product, it can assess whether the software product will be acceptable to its end users, its target audience, its purchasers, and other stakeholders [9]. Software testing is the process of attempting to make this assessment. Software does not suffer from corrosion, wear-and-tear; generally it will not change until upgrades, or until obsolescence. So once the software is shipped, the design defects or bugs will be buried in and remain latent until activation. In system testing, there are two type of testing:

1. Functionality testing
2. Non-functionality testing

Functionality testing means the testing whether application is functioning as per requirement or not. There are several types non-functionality testing, e.g. Load, stress, performance, reliability, security, usability, configuration, compatibility (forward & backward) and scalability.

Software testing is more than just error detection; testing software is operating the software under controlled conditions, to

1) verify that it behaves “as specified”;
2) detect errors, and
3) Validate that what has been specified is what the user actually wanted. [10]

B. Importance of Software Testing

Testing can never completely identify all the defects within software. Instead, it furnishes a criticism or comparison that compares the state and behavior of the product against principles or mechanisms by which someone might recognize a problem. These oracles may include specifications, contracts, comparable products, past versions of the same product, inferences about intended or expected purpose, user or customer expectations, relevant standards, applicable laws, or other criteria.

Software that does not satisfy the requirement of the customer after development, needs to be changed depending on the clients need. By changing any parts of that software may lead to such fault(s) to occur, which can affect the surrounding test cases and can easily come out with redundant and unexpected or dependent faults. As a result, developers have to do all those testing again which is time consuming and tiresome. Most significantly, when developers require more time to produce improved software consequently it takes higher cost than that of the previous one which reduce customer’s attraction to avail the software. Here testing is needed which can easily solve the problem by tracing faults and bugs. [11]

The main aim of software testing is to find out the error or bugs to improve the quality of the system [12]. During the development phase of software system, cost of testing a program is associated [13]. Tester has to write test plan and test cases for setting up the proper equipment, executing the test cases systematically. They also have to follow up the problems that are identified as well as try to remove most of the identified problems. It is simply impossible to test every possible input-output combination of the system. As a result testers need to consider the economics of testing and strive to discover test cases that will uncover as many faults using minimal number of test cases [3]. That is why testing is necessary when it couldn't guarantee 100% error free software application. And also:

- Cost of fixing the bug will be more expensive if it is found in later stage than it is found earlier.

- Quality can be ensured only by testing. In the competitive market, only Quality product can exist for long time.
- Testing will be necessary even if it is not possible to do 100% testing for an application.

C. Test Case

IEEE Standard 610 (1990) defines test case as follows: "A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement." (IEEE Std 829-1983) defines test case as follows: "*Documentation specifying inputs, predicted results, and a set of execution conditions for a test item*."

Table 1. Goals for Test Cases

Goal	Description
Find defects	Test is run to trigger failures that expose defects in all parts of the product.
Maximize bug count	Finding the most bugs in the time available is more important than coverage to cover up the high risk
Block premature product releases	Tester stops premature shipment by finding and fixing bugs.
Help managers make ship / no-ship decisions	Managers want to know about coverage and how important the known problems are. Problems which are not lead to customer dissatisfaction are probably not relevant to the ship decision.
Minimize technical support cost	Working in conjunction with a technical support group, the test team identifies the issues that lead to calls for support. These are often peripherally related to the product under test.
Assess conformance to specification	In the specification, any claim made is checked but the non-addressed program characteristics are not checked.
Conform to regulations	Test group is focusing on everything that is covered by regulation and (in the context of this objective) discard that is not covered by regulation.
Minimize safety-related lawsuit risk	Any error that could lead to an accident or injury is needed to be addressed whereas errors that lead to loss of time, data or corrupt data carrying no risk of injury or damage are out of scope.
Find safe scenarios for use of the product	Tester does not looking for bugs whereas trying out ways to do a task through refining and documenting.
Assess quality	To assess quality, one probably need a clear definition of the most important quality criteria for this product, and then need a theory that relates test results to the definition.
Verify correctness of the product	It is done by assessing test-based estimation of the probability of errors.
Assure quality	Quality assurance involves building a high quality product which requires skilled people who have appropriate balance of direction and creative freedom. It is within scope for the project manager and associated executives.

In [14], researchers mentioned that, when a test case is run, several goals can be achieved. The goals are explained in the above table. 1.

D. Test Case Structure

A formal written test case comprises of three parts. These are as follows:

Information: Information consists of general information about the test case. Information incorporates Identifier, test case creator, test case version, name of the test case, purpose or brief description and test case dependencies.

Activity: Activity consists of the actual test case activities. Activity contains:

- information about the test case environment
- activities to be done at test case initialization
- the activities to be done after test case is performed
- step by step actions to be done while testing
- Input data that is to be supplied for testing.

Results: Results are outcomes of a performed test case. Result data consist of information about expected results and the actual results.

E. Designing Test Cases

Test cases should be designed and written by someone who understands the function or technology. A test case should include the following information:

- Purpose of the test
- Software requirements and Hardware requirements (if any)
- Specific setup or configuration requirements
- Description on how to perform the test(s)
- Expected results or success criteria for the test

Designing test cases can be time consuming in a testing schedule, but they are worth giving time because they can really avoid unnecessary re-testing or debugging or at least lower it. Organizations can design the test cases approaching their own context and according to their own perspectives. Some follow a general step way approach while others may opt for a more detailed and complex approach. It is very important for us to decide between the two extremes and judge on what would work. Designing proper test cases is very vital for our software testing plans. It can save save our time on continuous debugging and re-testing test cases.

Regression Testing

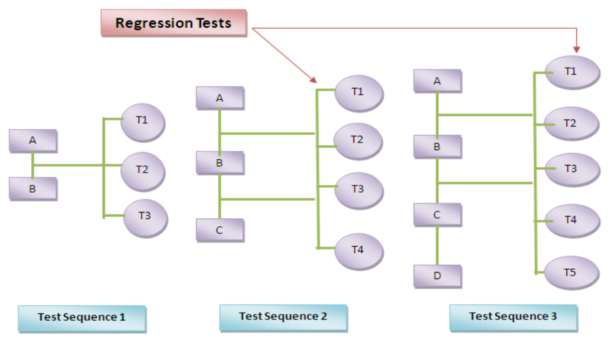

Fig.1. Regression Testing

Regression testing is also known as verification testing. Most common methods of regression testing include re-running of previously run tests and then verifying whether the program behavior has undergone any changes as well as identifying if any previously fixed faults have re-emerged or not. The main reason for carrying out regression testing is that, it gets difficult for a programmer or a developer to identify how a modification in one part of the software affects other parts of it. This is addressed by doing a comparison of results of previous tests with the results of the current tests being run.

Every custom software development organization follows different strategies for regression testing. Some strategies and factors to consider during this process include the following:

- Test fixed bugs promptly.
- Keep an eye on what all can be the side effects of bug fixes. It may be possible that a bug itself might be fixed but that fix might cause occurrence of other bugs.
- Write a regression test for every bug fixed.
- In case, any two or more tests are identical, try to figure out which test is less effective and get rid of it.
- Figure out the tests that the program consistently faces and archive them.
- Rather than focusing on design or cosmetic issues, focus on functional issues of an application.

Now, let *P* be a program, let *P'* be a modified version of *P*, and let, *T* be a test suite developed for *P*. Regression testing is concerned with validating *P'*. To facilitate regression testing, engineers typically reuse *T*, but new test cases may also be required to test new functionality. Both reuse of *T* and creation of new test cases are important. However test case reusing is the main concern, as such reuse typically forms a part of regression testing processes. [11] [15]

As regression testing is highly expensive. Several techniques have been researched for effective and efficient regression testing [16, 17, 18]. There are four major techniques for regression testing: retest-all [19], regression test selection [20], test suite reduction [21], and test case prioritization [15, 22]. Among them, test case prioritization has been perceived as one of the most effective and efficient techniques for regression testing [15, 23].

F. Test Case Prioritization

Regression testing is the re-execution of some subset of test that has already been conducted. It is an expensive testing process used to detect regression faults [7]. Regression test suites are often simply test that software engineers have previously developed and that have been saved so that they can be used later to perform regression testing [11]. Prioritizing test cases provide the opportunity to maximize some performance goals or effectiveness. One of the performance goals may be rate of severity detection among faults. During software testing, pragmatic experiences show that independent faults can be directly detected and removed, but mutually dependent faults can be removed if and only if the leading faults have been removed. That is, dependent (both fully and mutually) faults may not be immediately removed and the fault removal process lags behind the fault detection process. For example, if any software takes limited number of inputs and after functioning, generates several types of outputs then a single fault in input module may generate a large number of faults in output module if they are not mutually independent. Hence in regression testing if the test cases that reveal the faults of output module execute first and test cases reveals faults of input module executes later then it will be delayed and in many cases will take long time to detect the original cause of output faults. If more faults can be detected earlier in regression testing then debugging can be started earlier and fault removal time will improve. In this paper, we will present a metric which measures fault severity detection and also present an algorithm to improve the existing ordering. A comparison between prioritized and non-prioritized test cases is also shown with the help of new technique.

G. Test Case Prioritization with General Term

"Ref [11, 22]" define the test case prioritization problem and describe several issues relevant to its solution; this section reviews the portions of the material that are necessary to understand this article.

Definition I: The Test Case Prioritization Problem
Given: *T*, a test suite; *PT*, the set of permutations of *T*; *f*, a function from *PT* to the real numbers
Problem: Find *T'* belongs to *PT* such that (for all *T''*) (*T''* belongs to *PT*) ($T'' \neq T'$) [$f(T') \geq f(T'')$]

Here, *PT* represents the set of all possible prioritized test case orderings of *T*, and *f* is a function that applied to any such ordering, yields an award value for that ordering (For simplicity and without loss of generality, *Definition II* assumes that higher award values are preferable to lower ones).

There are several aspects of the test case prioritization problem that are worth describing further. First, there are many possible goals of prioritization, including the following:

- Testers may wish to increase the rate of fault detection of a test suite, that is, the likelihood of revealing faults earlier in a run of regression tests using that test suite.
- Testers may wish to increase the coverage of coverable code in the system under test, at a faster rate, allowing a code coverage criterion to be met earlier in the test process.
- Testers may wish to increase their confidence in the reliability of the system under test at a faster rate.
- Testers may wish to increase the rate at which high-risk faults are detected by a test suite, thus locating such faults earlier in the testing process.
- Testers may wish to increase the likelihood of revealing faults, related to specific code changes, earlier in the regression testing process.

In the definition of the test case prioritization problem, ***f*** represents a quantification of such a goal. Given any prioritization goal, various test case prioritization techniques may be used to meet that goal. For example, to increase the rate of fault detection of test suites, we might prioritize test cases in terms of the extent to which they execute modules that have tended to fail in the past. Alternatively, we might prioritize test cases in terms of greatest-to- least coverage-per-cost of code components, or in terms of greatest-to-least coverage-per-cost of features listed in a requirements specification. In any case, the intent behind the choice of a prioritization technique is to increase the likelihood that the prioritized test suite can better meet the goal than would an ad hoc or random order of test cases.

In [22], researchers distinguished two types of test case prioritization: general and version specific. In general test case prioritization, given program *P* and test suite *T*, test cases in *T* are prioritized with the goal of finding a test case order that will be useful over a sequence of subsequent modified versions of *P*. Thus, general test case prioritization can be performed following the release of some version of the program during off-peak hours and the cost of performing the prioritization is amortized over the subsequent releases. The expectation is that the resulting prioritized suite will be more successful than the original suite at meeting the goal of the prioritization, on an average over those subsequent releases.

In contrast, in version-specific test case prioritization, give program *P* and test suite *T*, test cases in *T* are prioritized with the intent of finding an ordering that will be useful on a specific version *P'* of *P*. Version-specific prioritization is performed after a set of changes have been made to *P* and prior to regression testing *P'*. Because this prioritization is performed after *P'* is available; care must be taken to prevent the cost of prioritizing from excessively delaying the very regression testing activities it is supposed to facilitate. The prioritized test suite may be more effective at meeting the goal of the prioritization for *P'* in particular, than a test suite resulting from general test case prioritization; however may be less effective on average over a succession of subsequent releases.

H. Test Case Prioritization Existing Techniques

Test case prioritization techniques schedule test cases for execution in an order that attempts to maximize some objective function. A variety of objective functions are applicable; one such function involves rate of fault detection - a measure of how quickly faults are detected within the testing process. Test case prioritization techniques [5,15] provide another method for assisting with regression testing. These techniques let testers order their test cases so that, those test cases with the highest priority, are executed earlier in the regression testing process. For example, testers might wish to schedule test cases in an order that achieves code coverage at the fastest rate possible, exercises features in order of expected frequency of use or exercises subsystems in an order that reflects their historically demonstrated propensity to fail.

When the time required to re-execute an entire test suite is short, test case prioritization may not be cost-effective; but it may be sufficiently simple to schedule test cases in any order.

When the time required to execute an entire test suite is sufficiently long, test case prioritization may be beneficial, as in this case, meeting testing goals earlier can yield meaningful benefits. Test case prioritization techniques do not themselves discard test cases. They can avoid the drawbacks that can occur during regression test selection. Alternatively, in cases where the discarding of test cases is acceptable, test case prioritization can be used in conjunction with regression test selection or test suite minimization techniques to prioritize the test cases in the selected or minimized test suite. Further, test case prioritization can increase the likelihood that, if regression testing activities are unexpectedly terminated, testing time can be utilized more beneficially than if test cases were not prioritized.

The prioritization process is further divided in number of sub techniques to assign the priorities. Some of the test case prioritization techniques are presented in table 2.

I. Focus Area

Our focus is in the regression testing part where we want to prioritize test cases based on all types of fault dependency. Therefore, we consider the areas where associate faults of each test case are dependent both mutually and fully on other faults. In our proposed theme, we took the relationship of partially dependent faults into consideration.

Let us discuss this fact with an example. Consider an example of five test cases with ten faults. The test cases are *T1, T2, T3, T4,* and *T*5 which exhibits some faults named *F*1, *F*2, *F*3, *F*4, *F*5, *F*6, *F*7, *F*8, *F*9, *and F10*. Now, we take fault *F*10 is dependent (totally) on fault *F3*; *Fault F*2 is dependent (partially) on both *F2* & *F3* and so on. We consider that associated faults of *T*1 and *T*3 are *F*1, *F4, F10* and *F2*, *F5*, *F*8, F9 respectively. Here, in this case if we can detect and solve the leading faults *F*3 and *F*4 then the partially dependent fault *F*2 and fully dependent *F10* will automatically be solved.

Table 2. Different Test Case Prioritization Techniques

Code	Mnemonic	Description
M1	unordered	no prioritization (control)
M2	ordered	randomized ordering
M3	optimal	ordered to optimize rate of fault detection
M4	branch-total	prioritize in order of coverage of branches
M5	branch-addtl	prioritize in order of coverage of branches not yet covered
M6	FEP-total	prioritize in order of total probability of exposing faults
M7	FEP-addtl	prioritize in order of total probability of exposing faults, adjusted to consider effects of previous tests
M8	stmt-total	prioritize in order of coverage of statements
M9	stmt-addtl	prioritize in order of coverage of statements not yet covered

In this paper, we will take those faults into consideration which is both partially and fully dependent on other leading faults in a test suite. From now, in our proposed theme, dependency will always mean both partial and fully. For this purpose, an algorithm will be build such that it distinguishes the improvements and differences between the independent and fully & partially dependent faults exhibits by the test cases by measuring the severity rates (severity/ execution time). After that, we will analyze the improvement of our proposed algorithm by plotting the graph of percentages of dependent fault severity verses percentages of test case executed which is going to give an efficient and improved result compared to the graph of independent one. However, we assume that, having these two graphs that will be precisely shown by incurring the same percentages test case execution, we will be able to find more severities than the previous independent graph.

J. Literature Review

"Reference [25]", first proposed the approach of test case prioritization, but in the paper [26, 15] researchers proposed and evaluated the approach in a more general context. Later on, many researchers have studied this technique with different goals and perspectives.

While the majority of the prioritization techniques [15, 22, 23] cover some structural coverage such as *branch-total*, *statement-total*, *Fault Exposing Potential* (*FEP*)-*total*, modified condition/decision coverage (MCDC) criteria [26] and so on; there are prioritization based on other criteria such as Case-Based Reasoning (CBR) approach [27], Interleaved Clusters Prioritization (ICP) technique [28], probabilistic approach [39], model based approach [30], "coarse grained " techniques based on function coverage [31] and so on. Authors of [32] used Test Case Selection and Prioritization techniques such as Genetic Algorithms, Ant Colony Optimization. In [33], a hybrid technique was proposed by the researchers, that combines modification, minimization and prioritization-based selection. Which uses a list of source code changes and the execution traces from test cases run on previous versions.

Test case prioritization does not filter out test cases; rather the entire test suite is executed, that is not always cost effective. A number of *cost-aware* prioritization technique [34],[35],[36] addressed this problem. With respect to cost awareness, "Ref [35]" extended the basic *APFD* (Average Percentage of Faults Detected) metric to *APFDC* (Cost-Cognizant Weighted Average Percentage of Faults Detected) that incorporates not only the cost of test cases but also the severity of faults detected. Researchers of [37], proposed the Historical Value-Based Approach, which is based on the use of historical information, to estimate the current cost and fault severity for cost-cognizant test case prioritization. "Ref [38]" presented a metric for assessing the rate of fault detection of prioritized test cases, APFDc, that incorporates varying test cases and fault costs. Authors of [39], proposed a system of test cases sequencing as well as reduction by using an intelligent dynamic approach

On the other hand, fault dependency [40], [41], [42] has been studied in many cases such as in integration testing [43], test case filtering [44], and software reliability analysis [42] and so on.

While prioritizing test cases, fault dependency was not considered earlier. Hence in an unpublished paper [8] researchers included fault dependency in cost-aware test case prioritization proposed in [11] [35]. They detected more severe leading faults earlier with least amount of execution. We also have compared this new approach with the previous approach and have shown the effectiveness of the new ordering.

In "Reference [8]", authors discussed and defined fault dependency as "faults can be dependent to one another and without considering it the prioritization is less effective". In terms of dependency degree there are two types of dependency:

- ➔ Fully dependent faults: If the leading fault is removed then the dependent fault is also removed.
- ➔ Mutually/Partially dependent faults: The dependent fault is not immediately removed after the leading fault is removed, but requires some correction to fully remove it. Because a fault may depend on more than one leading faults.

During software testing, pragmatic experiences show that independent faults can be directly detected and removed, but dependent faults can be removed if and only if the leading faults have been removed or solved. That is, dependent faults may not be immediately removed, and the fault removal process lags behind the fault detection process [42].

In almost all software testing, maximum amount of faults (in test cases) needed to be detected with the highest degrees of severities. If more severities can be detected in the same cost (Execution time), then that will be an efficient regression testing process. Regression Testing is the selective re-testing of a system or a component to verify that modifications have not caused unintended effects and the system or component is still in accordance with its specified requirements [45]. Regression testing activities are triggered based on software changes or evolutions [46]. As a result, by considering dependent faults one can detect more severe faults in same amount of execution time than the previous case.

Moreover, in some test cases, there can be both independent and dependent faults. When there are all or most of the faults are mutually independent, the measurement of severity rate is quite normal. But there remain problems with dependent faults. In test suites, where a presence of dependent faults is seen, severity calculations possess significant improvement in detecting the severity rate of dependent faults than that of the independent one. If we consider the dependent faults, then severity of faults will surprisingly increase due to the same percentages of test case execution.

Therefore in the software testing process, running same test cases definitely will make the testing process efficient and will eventually reduce the software cost (prices) as well. For this purpose we will propose a matrix where a significant amount of improvement will occur in the field of severity rate detection of both totally and mutually dependent faults in test cases.

III. Problem Formulation and Proposed Work

In this section we will formulate the problem we have found out from the previous discussions. Based on the findings we will propose our technique to prioritize test cases.

A. Cost Cognizant Test Case Prioritization

Majority of the prioritization techniques that are concerned about some structural coverage do not consider varying test costs and fault severities. But in practice faults vary in severity and test cases vary in cost. In [1] the authors proposed cost-cognizant test case prioritization which tradeoffs between cost and severity.

Definition 3.1: (Cost Cognizant Test Case Prioritization):

Given T, a test suite of n test cases with costs $c_1, c_2, \ldots c_n$; F be a set of m faults revealed by T with severities $s_1, s_2, \ldots, s_m$; T' be an ordering of T such that if $f_c(Ti) > f_c(Tj)$ then Ti appears before T_j in ordering T'.

The function $f_c(T_i)$ in its simplest form is calculated as:

$$f_c(T_i) = \sum k \in R_i \, s_k / c_i \, . \quad (1)$$

Where, T_i is the test case and c_i is its corresponding cost. s_k is the severity of fault F_k and R_i is the set of fault numbers revealed by test case T_i.

B. Dependency Cognizant Test Case Prioritization

In [8], researchers considered the fault dependency of the test cases. The goal of this paper was to extend the cost cognizant test case prioritization technique [11] by introducing dependency among faults. More specifically we replace the function $f_c(T_i)$ by $f_d(T_i)$ so that the leading faults are identified earlier based on their severity per unit cost. This proposed matrix showed a significant amount of improvement occurred in the field of severity rate detection of totally dependent faults in test cases.

Definition 3.2: (Dependency Cognizant Test Case Prioritization):

Given T, a test suite of n test cases $T_1, T_2 \ldots T_n$, with costs $c_1, c_2, \ldots, c_n$; F be a set of m faults $F_1, F_2, \ldots, F_m$ revealed by T with severities $s_1, s_2, \ldots, s_m$; $F_i \leftarrow F_d$, where F_d is a set of faults that are dependent on fault Fi; T' be an ordering of T such that if $f_d(T_i) > f_d(T_j)$ then T_i appears before T_j in ordering T'.

Here $f_d(T_i)$ computes severity/cost of T_i but it considers all dependent faults that are discovered by T_i. The new function thus equates severity/cost according to the following equation:

$$f_d(T_i) = \frac{\sum_{k \in R_i} s_k + \sum_{k \in R_i} \sum_{u \in D_{F_k}} s_u}{c_i} \quad (2)$$

In paper [8], for simplicity they only considered the fully dependent faults. But there could be mutually or partially dependent faults too in the test suite. As a result, in this paper, we will consider both types of dependencies and from now on dependency will mean both partial and fully dependency. So we will incorporate all sorts of faults considering independent and dependent faults in my proposed work.

C. Test Case Prioritization Considering Independent Faults

To quantify how rapidly a prioritized test suite can detect dependency among faults, an objective function is required. For this reason, I first consider test cases' with independent faults; calculate severity rate and then consider the same test cases with both fully and partially dependent faults to calculate the total dependent severity. In each case of severity detection process, we will prioritize test cases in descending order. That means, higher severity rate is given more award then rest of the other.

From table. 3, we found that there are six test cases with ten faults occurred in the test suite. In the test case *T1*, the faults *F1*, *F4* and *F10* have occurred at first. Similarly *F3* and *F8* are found in the test case *T2*. For *T3*, faults *F2*, *F5*, *F8* and *F9*, for test case *T4*, faults *F7* and *F10*, for test case *T5*, faults *F3*, *F6* and *F8* and finally for test case *T6*, faults *F2* and *F9* are found.

Test suite generally contains several numbers of test cases where there is various numbers of faults occurred in each. Those faults may be of various types. They can be independent, can be mutually (partially) dependent or totally dependent among themselves in a test suite. I have taken such an example with a dependency graph where there are some independent and some dependent (both mutually and totally) faults in the fig. 2.

We have total ten faults in a test suite and they are *F1, F2, F3, F4, F5, F6, F7, F8, F9* and *F10.* Now let us consider the following fault dependencies:

F2 --> F3, F4;   F10 --->F3;

Here as it can be seen from the table. 3 and fig. 2 that the faults *F1, F3, F4, F5, F6, F7, F8* and *F9* are independent faults which means they do not have any dependency mentioned above. But the faults *F2 and F10* are dependent faults. This means that fault *F2* is mutually dependent on both *F3* and *F4*. Fault *F10* is dependent on *F3.*

Table 3.Example of Test Suite and Faults Exposed

	F1	***F2***	***F3***	***F4***	***F5***	***F6***	***F7***	***F8***	***F9***	***F10***
T1	*			*						*
T2			*					*		
T3		*	*		*			*	*	
T4							*			*
T5			*			*		*		
T6		*	*				*			

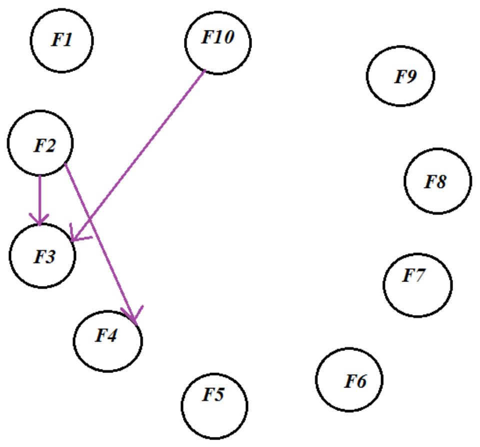

Fig.2. Dependency Graph.

Table 4. Execution Time (Cost) of the Test Cases

Test cases	**Execution time**
T1	8
T2	5
T3	8
T4	5
T5	6
T6	5
Total Execution Time	**37**

Table 5. Severities of the Faults

Fault	**Severity**
F1	6
F2	4
F3	3
F4	7
F5	10
F6	4
F7	5
F8	4
F9	2
F10	3
Total Severity	**48**

The execution time (cost) of the six test cases and the fault severities of the ten faults are mentioned above.

D. Analysis & Calculations

The steps to calculate the severity rate of six test cases are given bellow:

1. At first, I have to look at the test suit to get the information about the total number of faults and test cases.
2. Then I have to keep track of which fault has occurred in which test case.
3. Next task is to look at the dependency graph to get the idea of independent and dependent faults.
4. After that, it is the time to build the calculation of severity rate of each fault. For example, in the above table, I can calculate the severity rate of associated fault for each test case. For the test case *T1*, three faults occurred. These are *F*1, *F*4 and *F*10.
5. According to [11],

$$f_c\,(T_i) = \sum k \in R_i\, s_k \,/\, c_i$$

The steps to calculate the severity rate of six test cases are where, T_i is the test case and c_i is its corresponding cost. s_k is the severity of fault F_k and R_i is the set of fault numbers revealed by test case T_i.

Now the next step will be to calculate the severity rate of independent faults following the Table. 3 & fig. 2.

For example: For the test case *T*1, three faults have occurred *F*1, *F*6 and *F*10.The severity of the three faults are 5, 4 and 1 respectively. As a result we take the summation of those three fault severities and divide it by the execution time of the test case *T*1.The execution time can be found from the table. 4. So the desired result will be: *T*1= (6+7+3)/8=2;

Now, the calculations are shown below:

Table 6. Severities Considering Independent Faults

Severity Rate (Independent Faults) of each Test Case, $f_c\,(T_i)$
T1 = (6+7+3)/8 = 2
T2 = (3+4)/5 = 1.4
T3 = (4+3+10+4+2)/8 = 2.875
T4 = (5+3)/5 = 1.6
T5 = (3+4+4)/6 = 1.83
T6 = (4+3+5)/5 = 2.4

6. Once all the severities of six test cases is calculated then all the test cases is rearranged in a descending order. Therefore, following the above steps the desired order will be:

T3-T6-T1-T5-T4-T2

The above steps covered have taken only the independent faults into consideration. But now my newly proposed theme will be implemented and I will show the improvement and difference between the independent and dependent (both fully and mutually) faults associated with the test cases.

E. Test Case Prioritization Considering Dependent (Both Fully & Mutually) Faults

Now, we will consider the fault dependency of the test cases. In [8], researchers discussed about the types of dependencies. Dependencies could be of two types (fully & partially). Here for simplicity we only considered the fully dependent faults and kept the partially dependent faults for future research. Now, in this paper mutually/partially dependent fault will be the main focus to be considered. The partially dependent fault is not immediately removed after the leading fault is removed and requires some correction to fully remove it.

Hence the goal of this paper is to extend the dependency cognizant test case prioritization technique [8] by incorporating both types of dependency (full and mutual) among faults. More specifically, we will replace the function $f_d(T_i)$ by f_{pf} so that the leading faults are identified earlier based on their severity per unit cost.

Definition: Given T, a test suite of n test cases T_1, $T_2 \ldots T_n$, with costs $c_1, c_2, \ldots, c_n$; F be a set of m faults $F_1, F_2, \ldots, F_m$ revealed by T with severities $s_1, s_2, \ldots, s_m$; $F_i \leftarrow F_{pf}$, where f_{pf} is a set of faults that are dependent on fault F_i; T' be an ordering of T such that if $f_{pd}(T_i) > f_{pd}(T_j)$ then T_i appears before T_j in ordering T'.

Here $f_{pd}(T_i)$ computes severity/cost of T_i but it considers all dependent (both partial and fully) faults that are discovered by T_i. The new function thus equates severity/cost according to the following equation:

$$f_{pf}(Ti) = \frac{\{f_{fd}(Ti) + f_{pd}(Ti)}{ci} + f_c(Ti) \cdot \qquad (3)$$

If the equation is generalized, then it will be as following:

$$f_{pf}(T_i) = \frac{\sum_{k \epsilon R_i} s_k + \sum_{k \epsilon R_i} \sum_{u \epsilon D_{F_k}} s_u + \left(\frac{\sum_{k \in D_{pF_k}}}{2} s_k\right)}{c_i} \qquad (4)$$

Here, T_i is the test case and c_i is its corresponding cost. s_k is the severity of the dependent fault F_k and R_i is the set of fault numbers revealed by test case Ti. D_pF_k is the set of the faults on which F_k is mutually dependent.

In this proposed algorithm, we have the input and output part. The aim is to show that if the fault dependency (both partial and fully) is considered then more severity can get than from the independent one through prioritizing the test cases. For simplicity, we consider that, the mutually dependent fault will be dependent with an equal dependency rate that means it will equally be dependent on its associate faults on which it depends.

```
Algorithm: Test Case Prioritization Considering Dependent Faults
Algorithm Input: Test suite T, Fault severity f, and test costs or execution time of each test case t
Output: Prioritized test suite T'
  1: begin
  2:     set T' empty
  3:         for each test case t ∈ T do
  4:           Take (the summation of number of fault severities f covered by t + the severities of the totally dependent faults of occurred fault covered by t + (sum of partially dependent fault severities covered by t /2) ) / execution time of each t
  5:         end for
  7:       sort T in descending order based on the award value of each test case
  8:     let T' be T
  9: end
```

First the same faults and test suite with same above dependency graph is considered. Severities of the faults and the execution time of the test cases are same too. Following is the simple representation of the dependency graph.

Table 7. Severities Considering Dependent Faults

Severity Rate (Dependent Faults) of each Test Case, $f_{pf}(T_i)$
$T1 = 2 + (3/8) = 2.375$
$T2 = 1.4$
$T3 = 2.875 + \{((\frac{3+7}{2}) + 3)/8\} = 3.875$
$T4 = 1.6 + (3/5) = 2.2$
$T5 = 1.83$
$T6 = 2.4 + \{((\frac{3+7}{2}) + 3)/5\} = 4$

Fault ***F10*** is dependent on ***F3***. Fault ***F2*** is mutually dependent on both of the faults ***F3*** and ***F4***. For simplicity,we consider that fault ***F2*** depends on both ***F3*** and ***F4*** with an equal dependency rate that is 50% each. Therefore, by applying the above proposed algorithm and (4), we get the new dependency fault severity rate of the corresponding test cases. For example:

In the test case $T1$ our previous severity rate was 2 and as $F1$, $F4$ and $F10$ have occurred in test case $T1$. Among the three faults $F10$ is dependent on $F3$. So, $F3$ will be considered here. The severity of $F3$ is 3 and the execution time of test case $T1$ is 8.Therefore, the total dependent severity rate will be: $T1 = 2 + (3/8) = 2.375$;

Then we repeat the same heuristics to calculate the severity rates of the rest. Eventually the following table is found. The calculation shown is done using (4) considering the dependency graph fig. 3. These severity rates are given in Table 7.

Therefore, following the proposed algorithm & formula (4), the new desired order will be:

T6-T3-T1-T4-T5-T2

Test case ordering considering Independent faults:

T3-T6-T1-T5-T4-T2

Test case ordering considering Dependent (both fully & mutually) faults:

T6-T3-T1-T4-T5-T2

From table 7., we observe that, in the previous order *T*3 was in the first position but in my improved one, we found *T*6 at the highest award value because in the test case *T*6, faults *F2, F3* & *F7* occurred where *F2* has the highest amount of dependencies than rest of the other and the fault *F3* have dependent fault *F10*. In the test case *T*6, the occurred faults are *F*2, *F3* and *F*7 which is even less in number than the test case *T*3 (possesses faults *F2, F3, F5, F8* & *F9*). But the fact is *T*6 has higher amount of dependencies *F*3 and *F*4 as *F2* depends on *F3* & *F4* as well *F10* as it depends on *F3*. Therefore, if I consider the dependencies (using the proposed algorithm) and add their severities to the previous severity rate of *T6*, then we will have greater award value than that of *T*3.That is why in my improved test case ordering, *T*6 gets the highest award value than of the other test cases.

Similarly, in previous order *T5* comes before *T4* whereas in the proposed ordering *T4* comes before *T5*. *T*4 has the highest award value in *T*4, as faults *F4, F7* & *F10* occurred where *F10* has the dependency that is *F10* depends on *F3*. On the other hand, faults *F3, F6* & *F8* occur in test case *T5* and these faults do not have any dependency. So following the proposed algorithm, *T*4 gets the highest award value than *T5* and thus it came before *T5*.

The new ordering is evidently showing the strength of considering the dependent fault severities in prioritizing the test cases. Now plotting is done into graph for the both previously and newly proposed test case orderings to show that the new ordering can cover more severities due to the same percentage of test case execution than the old ordering system without considering total dependent faults. Then according to [11], two graphs are plotted to show the cumulative fault severity detected and its associated cost.

In table 8, the ordering of both new and existing one has shown. Now, first the cumulative of the previous one is taken in table 9.

The previous ordering is:

T3-T6-T1-T5-T4-T2.

Then I take the cumulative of proposed ordering in table 10.

The proposed ordering is:

T6-T3-T1-T4-T5-T2.

Table 8. Comparison between the existing and new ordering

Previous Test Case Order	Execution Time	Severity	New Test Case Order	Execution Time	Improved Severity
T3	21.622	47.92	*T6*	13.51	45.833
T6	13.51	10.42	*T3*	21.622	33.33
T1	21.622	33.33	*T1*	21.622	12.5
T5	16.22	8.33	*T4*	13.51	0
T4	13.51	0	*T5*	16.22	8.33
T2	13.51	0	*T2*	13.51	0

Table 9. Cumulative of the previous order

Cumulative of execution time	Cumulative of Severity
21.622	47.92
35.132	58.34
56.754	91.67
72.974	100
86.484	100
100	100

Table 10. Cumulative of the new order

Cumulative of execution time	Cumulative of Severity
13.51	45.833
35.132	79.163
56.754	91.663
70.264	91.663
86.484	100
100	100

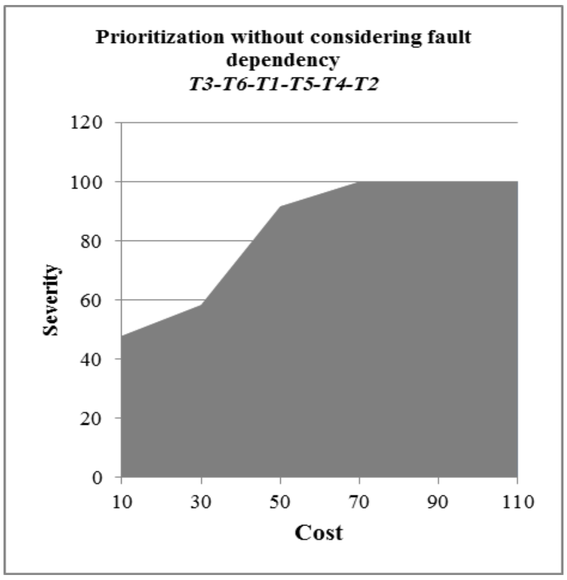

Fig.3. Prioritization Graph without Considering Fault Dependency

In this example, fault of *T4* and *T2* i.e. *F*7, *F10* and *F3* and *F*4 respectively was already considered before. So I get zero severities, i.e. no severity have found. But, in the new ordering of test cases, as I considered, the dependent fault *T6* get the highest award value. Therefore there is no question of redundant fault calculation. Here, in the new order we can detect *F*3 and *F4* merely earlier than the old detection technique. Fault *F3* and *F4* is detected in the execution period of *T6*; whereas by using the old process *F3* and *F4* can be detected in the execution of *T1* and *T5* which is in the third and fourth position.

According to [1], we have plotted two graphs to show the cumulative fault severity detected and its associated cost. Figure. 3 shows that **without considering fault dependency** 22% (approx.) costs is incurred to cover 48% (approx.)of severity of the faults. Figre.4 shows that if I consider **fault dependency**, 80% (approx.) of severity is covered by executing the same percentage of cost in this example. Therefore, it is obvious that the new ordering is better one than the former one.

We consider this problem because we want to show that, by the execution of same percentage of test cases we can detect more total dependent severity rate than the previous one. We are considering the dependent fault because in our process no fault repetition will be taken. For example:

T6= F2, F3, F7, F4, F10;
T3= F5, F8, F9;
T1 = F1;
T4 =0;
T5= F6;
T2=0;

Our previous ordering was:

T3-T6-T1-T5-T4-T2.

And the proposed order is:

T6-T3-T1-T4-T5-T2.

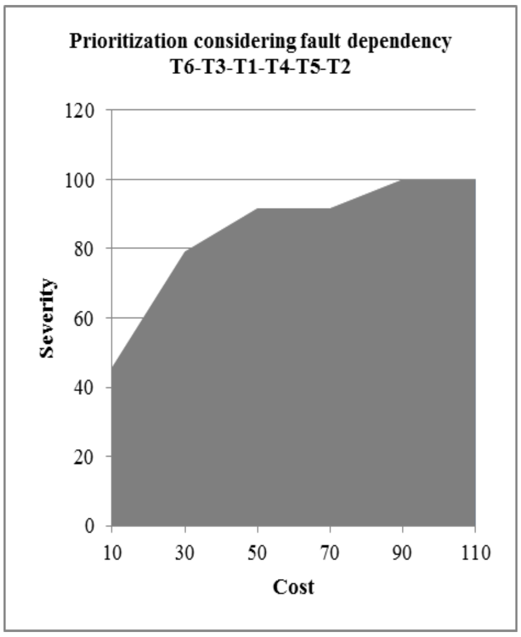

Fig.4. Prioritization Graph without Considering Fault Dependency

By applying this, we can omit the repetition of severity. In test case *T*6, I take *F*2, *F*3, *F*7 and *F*4. But in test case *T*3, I considered *F*5, *F*8 and *F*9 by omitting the repetition of severities for *F2* and *F*4. Similarly, in *T*1, I take only *F*1. For the test case *T*5, only *F6* have taken. For *T*4 and *T2* no fault will be taken as all faults are considered earlier.

IV. Conclusion And Future Work

Test case prioritization is a method to schedule and prioritize test cases. The technique is developed in order to run the test cases of higher priority for minimizing the time, cost and effort during the software testing phase. The literature review shows that many researchers propose many test case prioritization methods and approaches to prioritize and reduce the effort, time and cost in the software testing phase. Yet despite its use by practitioners, to date, less work has been done regarding to consider the dependent faults as well as independent faults severities and to incorporate it into any of the strategies proposed so far. This paper proposed an algorithm to measure effectiveness of test case prioritization in regression testing and a prioritization technique which can be used to improve the fault detection process for regression testing. Analysis is done for dependent (both fully & mutually) and independent test cases with the help of a proposed metric. Graphs prove that considering the dependent faults in the test cases make the detection more effective.

In this paper we proposed a new approach for test case prioritization. Here, we did an extension work of our unpublished research paper [8] which prioritizes the test cases based on the fault dependency. In [8], there were limitations of considering only the dependent faults which are fully dependent on other leading faults. But the fact is there can be faults that are not fully dependent rather mutually/partially dependent on other faults. Considering the dependent (both fully & mutually) fault make the regression testing process more effective than only considering the independent ones. By applying our algorithm, we demonstrated how my proposed technique is better than the existing one.

This report shows three primary contributions or gains to attempt all types of dependent faults under considerations.

- Firstly, as the algorithm contains the all types of dependent faults, so one can detect more faults earlier compared to the old version process. Here, we overcome the limitations available in [8].
- Secondly, by fulfilling the purpose of generating the new ordering using the improved algorithm, one does not have to consider the repetition of faults in the new ordering.
- Thirdly, our proposed algorithm detects both the leading faults as well as the fully & mutually dependent faults at a time. Detection and then elimination of the leading faults will automatically erase the dependent faults which lead less number of test cases to run. Thus it becomes time effective and less expensive due to the detection of dependent faults at earlier stage.

For enhancing the proposed approach and providing the best test case prioritization technique, we have many future add-ons.

In our research, we have used a simple case or scenario and verified it by examining and comparing its improvement with the previous approach. In future, we will consider more complex scenario and use real life example or program with diversified data to make my research more realistic.

In this paper, we have considered all types of dependencies, showed how dependency consideration helped in fault severity detection and compared the improvement from the independent fault detection process. We have consider mutually dependent fault to be dependent with an equal dependency rate on its associate faults. But in reality, dependency rate on associate faults cannot always be equal. The percentage of dependency rate may vary according to the scenario. So in future, we may propose a new approach to calculate the percentage of the dependency rate which may open new perspective to the fault severity detection process.

We did not apply the $APFD_C$ metric, which assesses the rate of fault detection of prioritized test cases that incorporates varying test costs and fault severities. Test costs are greatly diversified in software testing. Depending on the criteria, a test cost can be refined through several factors such as machine time, human time, test case execution time, monetary value of the test execution, and so forth [11]. So, we can make this approach as a cost effective approach by including any of these test cost factors under consideration. Then this approach may generate a prioritization technique which will be cost cognizant. It will make the approach more improved.

REFERENCES

[1] H.Park, H.Ryu, & J.Baik, "Historical value-based approach for cost-cognizant test case prioritization to improve the effectiveness of regression testing In Secure System Integration and Reliability Improvement", SSIRI'08. Second International Conference, IEEE pp. 39-46, July, 2008.

[2] G. M. Kapfhammer, Software testing. In The Computer Science Handbook, 2004.

[3] A. Singh, "Prioritizing Test Cases in Regression testing using Fault Based Analysis". *International Journal of Computer Science Issues (IJCSI) ,vol: 9(6),2012.*

[4] N. Chauhan, "Software Testing: Principles and Practices." *Oxford university press*, 2010.

[5] W. E.Wong, J. R. Horgan, S. London, H. Agrawal, (1997, November). "A study of effective regression testing in practice", *The Eighth International Symposium on Software Reliability Engineering Proceedings (pp. 264-274). IEEE, November, 1997.*

[6] Institute of Electrical and Electronics Engineers (IEEE). IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.

[7] P. R. Srivastava, (2008). *Test case prioritization. Journal of Theoretical and Applied Information Technology, vol: 4(3), pp: 178-181.*

[8] B. Hoq, S. Jafrin, S. Hosain, "Dependency Cognizant Test Case Prioritization". [Unpublished research work, Undergraduate thesis].

[9] E. Dustin, "Effective Software Testing: 50 Ways to Improve Your Software Testing", *Addison-Wesley Longman Publishing Co. Inc., 2002.*

[10] S. H. Trivedi, "Software testing techniques", *International Journal of Advanced Research in Computer Science and Software Engineering, vol: 2(10), pp: 433-438, 2012.*

[11] A. G. Malishevsky, J. R.Ruthruff, G. Rothermel, & S. Elbaum, "Cost-cognizant test case prioritization", *Department of Computer Science and Engineering, University of Nebraska-Lincoln, Techical Report.2006*

[12] I. Sharma, J. Kaur, M. Sahni, "A Test Case Prioritization Approach in Regression Testing",2014.

[13] C. Sharma, S. Sabharwal,R. Sibal, "A survey on software testing techniques using genetic algorithm", arXiv preprint arXiv:1411.1154, 2014.

[14] C. Kaner, "What is a good test case". Star East, 16, 2003

[15] G. Rothermel, R. H. Untch, , C. Chu, M. J. Harrold, "Test case prioritization: An empirical study. In Software Maintenance", IEEE International Conference on 1999.(ICSM'99) Proceedings. , pp: 179-188, 1999.

[16] J. M. Kim, A. Porter, G. Rothermel, "An empirical study of regression test application frequency", *Software Testing, Verification and Reliability,* vol: 15(4), pp: 257-279,2005.

[17] G. Rothermel, S. Elbaum, , A. G. Malishevsky, P. Kallakuri & X.Qiu, "On test suite composition and cost-effective regression testing", *ACM Transactions on Software Engineering and Methodology* (TOSEM), vol: 13(3), pp: 277-331, 2004.

[18] A. Srivastava & J. Thiagarajan, "Effectively prioritizing tests in development environment", *ACM SIGSOFT Software Engineering Notes* , vol. 27, No. 4, pp: 97-106), July,2002

[19] H. K. Leung & L. White, "Insights into regression testing [software testing]", *Software, Maintenance, 1989., Proceedings., Conference on, IEEE,* pp. 60-69, October,1989.

[20] G. Rothermel & M. J. Harrold, "Analyzing regression test selection techniques" , *Software Engineering, IEEE* Transactions on, vol: 22(8), pp: 529-551, 1996.

[21] S. Elbaum, D. Gable & G. Rothermel, "Understanding and measuring the sources of variation in the prioritization of regression test suites", *In Software Metrics Symposium, 2001. METRICS 2001. Proceedings. Seventh International IEEE*, pp: (pp. 169-179), 2001.

[22] G. Rothermel, R. H. Untch, C. Chu & M. J. Harrold, "Prioritizing test cases for regression testing", *Software Engineering, IEEE Transactions on*, vol: 27(10), pp: 929-948, 2001.

[23] X. Zhang, C. Nie, B. Xu & B. Qu, "Test case prioritization based on varying testing requirement priorities and test case costs", *In Quality Software, 2007. QSIC'07. Seventh International Conference on IEEE*, pp. 15-24, October,2007.

[24] W. E. Wong, J. R. Horgan, A. P. Mathur & A. Pasquini, (1999). "Test set size minimization and fault detection effectiveness: A case study in a space application", *Journal of Systems and Software*, vol: 48(2), pp: 79-89, 1999.

[25] M. J. Harrold, "Testing evolving software", *Journal of Systems and Software*, vol: 47(2), pp: 173-181, 1999.

[26] J. A. Jones & M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage",

Software Engineering, IEEE Transactions on, vol: 29(3), pp: 195-209,2003.

[27] P. Tonella, P. Avesani & A. Susi, “Using the case-based ranking methodology for test case prioritization”, *In Software Maintenance, 2006. ICSM'06. 22nd IEEE International Conference on IEEE,* pp:123-133, September, 2006 .

[28] S. Yoo, M.Harman, P. Tonella & A. Susi, “Clustering test cases to achieve effective and scalable prioritization incorporating expert knowledge”, *In Proceedings of the eighteenth international symposium on Software testing and analysis ACM*, pp. 201-212, July,2009.

[29] S.Mirarab & L. Tahvildari, “An empirical study on bayesian network-based approach for test case prioritization”, *In Software Testing, Verification, and Validation, 2008 1st International Conference on IEEE*, pp. :278-287, April,2008.

[30] B. Korel, G. Koutsogiannakis & L. H. Tahat, “Model-based test prioritization heuristic methods and their evaluation”, *In Proceedings of the 3rd international workshop on Advances in model-based testing ACM*, pp: 34-43, July,2007.

[31] T. Parthiban, R. Kamalraj & S. Karthik, “Establishing a Test Case Prioritization Technique Using Dependency Estimation of Functional Requirement”, *International Conference on Engineering Technology and Science-(ICETS’14). International Journal of Innovative Research in Science, Engineering and Technology, 2014.*

[32] P. Bansal, “A critical review on test case prioritization and Optimization using soft computing techniques”, *In 2nd International Conference on Role Of Technology in Nation Building (ICRTNB), ISBN*: 97881925922-1-3, 2013.

[33] Sunita, & M. Gulia, “Study of Regression Test Selection Technique”, *International Journal of Advanced Research in Computer Science and Software Engineering. India*, 2014.

[34] H. Do, S. Mirarab, L.Tahvildari & G. Rothermel, (2008, November). “An empirical study of the effect of time constraints on the cost-benefits of regression testing”, *In Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering ACM*, (pp. 71-82).

[35] S. Elbaum, A. Malishevsky & G. Rothermel, “Incorporating varying test costs and fault severities into test case prioritization”, *In Proceedings of the 23rd International Conference on Software Engineering. IEEE Computer Society*, pp. 329-338, July, 2001.

[36] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer & R. S. Roos, (2006, July). “Time aware test suite prioritization”, *In Proceedings of the 2006 international symposium on Software testing and analysis ACM*, pp. 1-12, July, 2006.

[37] P. R. Srivastva, K. Kumar & G. Raghurama, (2008). “Test case prioritization based on requirements and risk factors”, *ACM SIGSOFT Software Engineering Notes*, vol: 33(4), 2008.

[38] M. A. Askarunisa, M. L. Shanmugapriya & D. N. Ramaraj, “Cost and coverage metrics for measuring the effectiveness of test case prioritization techniques”, *INFOCOMP Journal of Computer Science*, vol: 9(1), pp: 43-52, 2010.

[39] A. Singh, (2012). “Prioritizing Test Cases in Regression testing using Fault Based Analysis”, *International Journal of Computer Science Issues (IJCSI)*, vol: 9(6), 2012.

[40] V. B. Singh, P. K. Kapur & A. Tandon, “Measuring reliability growth of software by considering fault dependency, debugging time Lag functions and irregular fluctuation”, *ACM SIGSOFT Software Engineering Notes*, vol: 35(3), pp: 1-11, 2010.

[41] Y. Wu, R. H. Yap & R. Ramnath, “Comprehending module dependencies and sharing”, *In Software Engineering, 2010 ACM/IEEE 32nd International Conference on. IEEE*. vol. 2, pp. 89-98, May,2010.

[42] C. Y. Huang & C. T. Lin, “Software reliability analysis by considering fault dependency and debugging time lag”, *Reliability, IEEE Transactions on*, vol:55(3), pp: 436-450, 2006.

[43] A. Srivastava, J. Thiagarajan & C. Schertz, “Efficient integration testing using dependency analysis”, *Microsoft Research, TechReport MSR-TR-2005-94*, 2005.

[44] D. Leon, W. Masri & A. Podgurski, “An empirical evaluation of test case filtering techniques based on exercising complex information flows”, *In Proceedings of the 27th international conference on Software engineering ACM,* pp. 412-421, May, 2005.

[45] H. Kumar, N. Chauhan, “A module coupling slice based test case prioritization technique”, *I.J. Modern Education and Computer Science*, vol: 7(7), pp: 8-16, 2015.

[46] I. Aslmadi, S. Alda, “Test cases reduction and selection optimization in testing web services”, *I.J. Information Engineering and Electronic Business*, vol: 4(5), pp: 1-8, 2012.

Authors’ Profiles

Samia Jafrin was born in Dhaka, Bangladesh, in 29th June, 1989. She received the BSc. degree in computer engineering from the North South University, Dhaka, Bangladesh, in 2010, and the MSc. degree in computer science from American International University Bangladesh, Dhaka, Bangladesh, in September, 2015. She has done her major field of study in software engineering.

In 2011, she joined a school named Scholastica, in Dhaka, Bangladesh as a Teacher and continuing. Her current research interests include software testing, regression testing, test case prioritization and fault dependency.

Dip Nandi has completed his PhD in Computer Science from RMIT University, Melbourne, Australia. His research interests include E-Learning, Software Engineering and Information Systems. Email: dip.nandi@aiub.edu

Sharfuddin Mahmood has completed his B.Sc and M.Sc degree in Computer Science from American International University-Bangladesh. His major was Information and Database Technologies. Currently his is focusing on Data Mining technologies and algorithms. His area of research is Data mining and knowledge discovery, Software Engineering and intelligent systems.