Prioritizing Test Cases for Resource Constraint Environments Using Historical Test Case Performance Data

Y. Fazlalizadeh, A. Khalilian, M. Abdollahi Azgomi and S. Parsa
Computer Engineering Department, Iran University of Science and Technology
Tehran, Iran
{ya_alizadeh, khalilian}@ comp.iust.ac.ir, {azgomi, pars}@ iust.ac.ir

Abstract—Regression testing has been widely used to assure the acquirement of appropriate quality through several versions of a software program. Regression testing, however, is too expensive in that, it requires many test case executions and a large number of test cases. To provide the missing flexibility, researchers introduced prioritization techniques. The aim in this paper has been to prioritize test cases during software regression test. To achieve this, a new equation is presented. The proposed equation considers historical effectiveness of the test cases in fault detection, each test case’s execution history in regression test and finally the last priority assigned to the test case. The results of applying the proposed equation to compute the priority of regression test cases for two benchmarks, known as Siemens suite and Space program, demonstrate the relatively faster fault detection in resource and time constrained environments.

Keywords—software regression test; test case prioritization; history-based prioritization; historical fault detection effectiveness

I. INTRODUCTION

Changing the software to correct faults or add new functionality can cause existing functionality to regress, introducing new faults. To avoid such defects, one can retest software after modification, a task commonly known as regression testing [1]. Regression testing typically involves the re-running of all test cases [2, 3] and it is often costly and sometimes even infeasible due to time and resource constraints. Hence, such a technique is considered as one of the most expensive tasks in software maintenance activities [1].

To reduce the cost of regression testing, various techniques have been proposed [3]. One technique is selecting all or a portion of the test-suite to execute. This technique is referred to as regression test selection (RTS) [4] and can be very costly. Another approach is test case prioritization which is one the main techniques used to address the problem of regression testing. The goal of test case prioritization is to execute important test cases with respect to some criterion, at first, in order to maximize a score function [5]. This technique is commonly used to increase fault detection rate as far as possible. It orders the test cases based on certain criteria and then run them in the specified order according to time and resource limitations.

The problem of finding optimal execution ordering for test cases is basically NP-hard and does not have any deterministic solution [6]. Thus, prioritization techniques are necessarily heuristic and produce sub-optimal results.

Currently, most prioritization techniques are memoryless and are based only on the analysis of source code and test case profiling information taken from the current and immediately preceding software versions. These techniques ignore historical test case performance data and commonly take a one-time testing model for software regression testing. Whereas software regression testing is continuous and long life [7], and its best model, is consecutive execution of test suite after each change occurs in current version of the software. To address this problem, History-based prioritization was introduced [7] which incorporates the notion of memoryful regression testing. The weakness of this approach, however, is that only effect of last execution of test cases, especially in binary manner (i.e. execution or not) is used to calculate the selection probability of test cases.

The main contribution of this paper is to present a new equation to compute the priority of test cases in each session of regression testing. The proposed equation considers the environment time and resource constraints and incorporates three factors: historical effectiveness in fault detection, each test case’s execution history in regression test and finally the last priority assigned to the test case. We also conduct an empirical study to evaluate the performance of our equation.

The rest of the paper is organized as follows. In Section 2, a brief description of the test case prioritization problem and are discussed. Section 3 presents the proposed approach Section 4 presents an empirical study of the proposed approach. Finally, conclusions are mentioned in section 5.

II. BACKGROUND AND RELATED WORK

A. Test Case Prioritization

Test case prioritization problem [8] was first introduced by Wong et al. in 1997 as a flexible method of software regression testing. Their technique first selects test cases based on modified code coverage and then prioritizes them. Research in this context, has been followed by Rothermel, Elbaum and other researchers [5, 6, 9, 10, 11, 12, 13, 14] which resulted in various techniques for test case prioritization. The formal definition of this problem, which has been widely accepted in the literature [5], can be expressed as follows:

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

Problem: Find T ∈ PT such that \( (\forall T') (T'\in PT) \) \( (T'\not= T) (f(T')) \geq f(T) \).

In this definition, PT represents the set of all possible
prioritizations (orderings) of T, and \( f \) is a function that, applies to any such ordering, yields an award value for that ordering [5].

Most of prioritization techniques are code-based, relying on information relating test cases to the coverage of code elements [3, 5, 9, 14]. Other non-coverage based techniques in the literature include fault-exposing potential (FEP) prioritization [5], model-based prioritization [12], history-based test prioritization [7, 15] and the incorporation of varying test costs and fault severities into test case prioritization [14, 16].

B. History-Based Test Case Prioritization

History-based test case prioritization is defined in [7] as follows: given test suite T, and T' be a subset of all test cases (T' \( \subseteq \) PT), \( P_{t,c,t}(H_{t,c}, \alpha) \) is selection probability of each test case tc in time t, and \( P_{t,c} \) is a set of t time ordered observations, \( \{h_0, h_1, \ldots, h_t\} \), drawn from the previous runs of tc which shows each test case's execution history up to now. According to this definition, selection probability of each test case in T' based upon execution history, is defined as follows:

\[
P_0 = h_1
P_i = a h_i + (1 - \alpha) P_{i-1} \quad k \geq 1, 0 \leq \alpha < 1
\]

In this equation, \( \alpha \) is a smoothing constant for weighting individual history observations. Based on different definitions of \( H_{t,c} \) in (1), history-based prioritization can be performed in the following ways:

\textit{Execution history.} For each test session i which tc is executed in it, \( H_{t,c} \) takes value 0 for next selection probability, otherwise takes value 1. In other words, after execution of high priority test cases in a session, they have lower selection probabilities in next sessions. Thus, instead of discarding low priority test cases in a session, they have lower selection probability, otherwise takes value 1. In other words, after execution of high priority test cases in a session, they have lower selection probabilities in next sessions. This definition of \( H_{t,c} \) limits the possibility that any function goes unexecuted for long period of time.

The issue with this definition is that the execution of effective test cases with respect to fault detection weakens their selection probability the same as other test cases and their effectiveness does not have any effect on increasing their priority and faster selection in next executions.

\textit{Demonstrated fault detection effectiveness.} For each testing session i, which tc exposes fault(s) in it (cause program to fail), \( H_{t,c} \) takes the value 1, otherwise takes value 0. This definition of \( H_{t,c} \), restricts execution of those test cases which rarely, if ever, reveal faults.

\textit{Coverage of program entities.} Program entities include statement, path, function, def-use pair, etc. In this case, higher priority values are given to those test cases which cover functions that are infrequently covered in past testing sessions. This definition of \( H_{t,c} \) limits the possibility that any function goes unexecuted for long period of time.

Notice that the equation in (1) uses only one of the mentioned definitions of \( H_{t,c} \) for test case execution history in order to determine test cases' selection probability.

Using \( h_i \) to determine selection probability, especially only with two values 0 and 1, and only based on the recent execution of each test case, is not an appropriate criterion to provide an execution history for the test cases. Furthermore, increasing test case selection probability only based on whether or not a test case has been executed in recent execution, or if it has exposed fault in recent execution, will not produce efficient ordering for test cases in history-based prioritization.

In a recent study [15], history based test case selection and cost-cognizant prioritization technique [10], were combined to form an approach for cost-cognizant history-based test case prioritization. This technique uses historical information of exposed faults' severity and cost, which is kept in a repository, for estimating current faults' severity and cost. Empirical studies showed that historical value-based approach is more effective than most of existing code-based approaches.

III. THE PROPOSED APPROACH

There are some factors, which are effective in determining test case priority in next selections. One of these factors is test case fault detection performance in number of times it has been executed in successive tests. We assume that in each new regression test session, some test case in ordered test suite have not been executed because of time and resource constraints. For example, consider two test cases, tcA and tcB in a test suite, such that tcA leads to 3 program fails in 20 sessions of execution and tcB leads to 9 program fails in 18 sessions of its execution. We observe that in spite of the number of executions of tcA is more than tcB, historical performance shows that tcA performs about 35 percent better than tcB with respect to fault detection. Thus, it must have greater priority with respect to tcA.

This example shows that we should not take the number of regression test sessions the test case executes and the number of test sessions it reveals fault(s) separately. But also these two factors together have effect on test cases’ historical performance as a single factor. Therefore, assuming \( k \)th execution of regression test which means that the software has been modified \( k \) times, and each time leads to a new version of software, let \( f_{c_k} \) be the number of times that the execution of test case tc fails and let \( e_{c_k} \) be the number of tc executions up to now, then we can show the relation between each test case’s priority with its fault detection performance in \( k \)th execution as follows:

\[
PR_k = \frac{f_{c_k}}{e_{c_k}}
\]

(2)

(3)

(4)

In other words, for each test case in each test session, historical performance is the proportion of the test case program fails to the number of its executions up to now. Remember that program fails in this ratio indicates the number of sessions in which the test case execution causes to fault revealing. We call this ratio \textit{historical fault detection effectiveness.}
Another effective factor in priority is a period of time that a test case is not being executed. In other words, we would like to ensure that after some sessions, all test cases in test suite will be executed and it will cycle through all test cases over multiple sessions and the related faults will be revealed. Based on what was described, we define $h_k$ in the $k$th execution as follows:

$$
\begin{align*}
    h_0 &= 0 \\
    h_k &= \begin{cases} 
    h_{k-1} + 1 & \text{if test case has been executed in test session } i \\
                           0 & \text{otherwise}
    \end{cases}  \quad (5)
\end{align*}
$$

Actually $h_k$ factor in (5) performs similar to a counter. In the context of operating systems and process scheduling, there is a known problem which is process starvation. If during the execution of various processes in a system, a process has not been selected for execution for a long time, the job scheduler may take the number of times the process has not been executed as process age and increases its priority. To do this, the job scheduler assigns a counter to each of the processes. The factor $h_k$ also performs the same job for the test cases. We will call it test case execution history. Based on what we described, there is a relationship between test case priority and its execution history in the $k$th execution:

$$
PR_k \approx h_k  \quad (6)
$$

Each time a test case is not executed, its execution history will be increased by one. Once the test case is executed, execution history becomes 0 and the operation is repeated as well. In this manner, we can ensure that none of the test cases has remained unexecuted for a long time, and the corresponding faults will be revealed.

Finally, the third factor, shown by empirical studies, is the recent priority of each test case in each test suite during past executions of the regression test. Then we have:

$$
PR_k \approx PR_{k-1}  \quad (7)
$$

There are some reasons using this factor: First, it causes smoother selection of test cases in successive executions of regression test. This limits severe changes in selection of executing test cases in test suite in each run with respect to the previous run. Second, in cases where the historical fault detection effectiveness of the test cases and their execution history are the same, we should consider another factor to establish a proper priority between them.

We define $PR_0$ for each test case as the percentage of code coverage of the test case and thereby various control-flow and data-flow coverage criteria can be used. Thus, the influence of test case code coverage will propagate in the next prioritizations due to recursion of the equation. In code-based prioritization techniques researchers empirically showed that the code coverage of each test case properly indicates its ability with respect to fault detection [3, 5]. In fact, it is a wise idea to suppose that test cases that cover more software code components are more likely to reveal faults than test cases which have less code coverage [5, 10, 14]. Thus, they should take higher priorities. It is worth to say that our proposed test case prioritization approach which performs based on the history of test case fault detection performance, could be also considered as a coverage-based approach. Based on the (5), (6) and (7) we can write the equation of each test case priority in the $k$th execution as follows:

$$
PR_k = \alpha \frac{fc_k}{ec_k} + \beta PR_h + \gamma h_k, \quad 0 \leq \alpha, \beta, \gamma < 1, k \geq 1 \quad (8)
$$

Changing $\alpha$, $\beta$ and $\gamma$ in (8), which are smoothing constants, we can control the effect of mentioned factors in test case prioritization. Constant $\gamma$ must be smaller than the other two constants ($\alpha$ and $\beta$ coefficients). This is because $h_k$ increases one unit per time according to whether or not the test case executes. In contrast, $fc_k$/$ec_k$ and $PR_h$ are numbers between 0 and 1. So, it is necessary to control the effect of $h_k$ against those two factors, such that it has not excessive effect in prioritization and do not mask other factors effects by mistake.

Note that $PR_k$ in (8) is the test case priority in the $k$th execution. Considering test time and resource constraints we will execute sufficient number of prioritized test cases, beginning from highest priorities.

### IV. EMPIRICAL STUDIES

#### A. Subject Programs

In our experiment, we used eight C programs as subjects. Siemens suite [17] includes seven programs in C which are widely used in other related works, in order to evaluate various prioritization techniques. Siemens contains programs, their associated test pools and test suites for each program. Siemens programs’ test suites were generated such that they could exercise different control-flow and data-flow coverage criteria. For each program, single-fault versions of the program have been created in which faults have been seeded by separate teams of programmers.

The other empirical study is a case study, which we have done on Space [14] benchmark program. Space is a big program (10KLOC) written in C, with real faults. It has been developed for the European Space Agency. Space has 38 associated versions, each containing a single fault.

#### B. Evaluation Metric

APFD (Average Percentage of Fault Detection) metric is commonly used for evaluating test case prioritization techniques. It is the weighted average percentage of fault detection in test suite lifetime, and was introduced by Rothermel et al. [3] in 1997 to assess and compare test case prioritization techniques in terms of how quickly faults are revealed during regression testing. Basically APFD metric is calculated as follows:

$$
APFD = 1 - \frac{TF_1 + TF_2 + \ldots + TF_{m}}{nm} + \frac{1}{2n} \quad (9)
$$
In (9), \( n \) is the number of test cases and \( m \) is the number of existing faults in software. Each \( TF_i \) in this equation shows the place of a test case in ordered suite which first reveals the fault \( i \). In simple words, the higher APFD number for a prioritization technique, the faster (better) that technique reveals faults during software regression testing [14].

C. Analysis Tools

In our experiments, SAS 9.1.3 [18] is used to create box plots. Box plot diagrams are commonly used to visualize the empirical results in test case prioritization studies. Using these diagrams, we can statistically analyze results and observe any differences between experiments.

D. Experiment Setup and Results

Our experiments follow a setup similar to that used by Rothermel et al. [14]. As mentioned above, each subject program has several single fault versions. For evaluating prioritization techniques, one needs versions with varying numbers of faults for each program. To this end, a set of multi-fault versions composed of non-interfering single faults (all faults that can exist simultaneously), have been created [14]. These faults are based on programmers' experience about common faults and therefore they are closer to real faults.

In our experiments, we have compared results of the proposed prioritization technique to the random ordering approach as it is common in previous studies [3, 5, 14]. We randomly selected 29 multi-fault versions of each program in order to simulate 29 sessions of regression testing. The 29 versions, is selected because it is the minimum number of non-interfering multi-fault versions that can be generated using single fault versions [14]. For each of the 29 versions, we have executed the proposed approach on 1000 branch coverage adequate test suites. \( \alpha \), \( \beta \) and \( \gamma \) coefficients were assigned 0.7, 0.7, and 0.04, respectively. To balance individual factors' effect in test case prioritization \( \gamma \) must be smaller than the two other coefficients since \( h_i \) increases one unit each time a test case does not execute. To implement the proposed approach, we assume that only a fraction of total prioritized test suite (about fifty percent) could be executed due to limitations in testing resources.

For representing the efficiency of the proposed approach, we have used box plot diagram [3, 5, 6, 13, 14]. Using box plot diagrams, we have represented the average amount of fault detection results of prioritizing 1000 branch coverage adequate test suites by the proposed approach versus results of the random ordering approach.

Since 29 multi-fault versions have been prioritized for each program, 29 pairs of box plots have been plotted in eight diagrams (Fig. 2). The box plots with same color in each diagram are respected to a specific faulty version of that program. In each same color pairs of box plots, results of APFD by prioritizing 1000 test suites using the proposed approach has been represented in the left side box and the corresponding plot for random ordering approach on the same 1000 test suites has been represented in the right side. This process has performed for each of the 8 programs, and is displayed in 8 diagrams from A to H in Fig. 2.

The box plots in the Fig. 2, display dispersion of APFD values along vertical axis and program versions along horizontal axis. The less spread of box plot along y-axis the most stable is prioritization technique's behavior in fault detection. Obviously, the higher the place of the box plot, the faster prioritization technique reveals faults. It can be observed in Fig. 2 that the proposed approach has considerable improvements both in fault detection and stability of the results, for various test suites.

E. Threats To Validity

In this section, we describe the potential threats to validity of our study. Our goal in this study is to increase the rate of fault detection in constraint environments. In our studies, measurements for the rate of fault detection and APFD values are accurate. But APFD supposes that the test case costs and fault severities are uniform. For this reason, another metric known as APFD, has been proposed [10] in order to consider different test costs and fault severities. Another important threat is instrumentation of programs' source code which was done by hand-instrumentation of the code. To ensure the quality of this process, we verified it by two persons. The other issue deals with the representativeness of the programs we used. The Space is a big and real program, but it is one of such a program we used. The faulty versions in Siemens suite are single-fault versions, but we needed multi-fault versions. Therefore, we made faulty versions with random number of non-interfering faults.

V. CONCLUSIONS

In this paper, we proposed a new history-based approach for prioritizing test cases. Our approach considers time constraints and enormous cost of repeatedly executing all test cases each time the regression test must be done. Therefore, in this approach we only execute a fraction of the prioritized suite.

In the proposed approach, three factors are effective in determining the test case execution priority: 1) Priority of the test case in previous regression test session, 2) Historical demonstrated performance in fault detection during the regression test lifeline, and 3) Duration of not execution for each test case. This approach directly uses these factors in prioritization. Empirical studies shows considerable improvements in fault detection rate which was measured using APFD.
REFERENCES


