International Journal of Computer Science Engineering and Information Technology Research (IJCSEITR)
ISSN 2249-6831
Vol. 3, Issue 3, Aug 2013, 127-134
© TJPRC Pvt. Ltd.

EMPIRICAL ANALYSIS OF TEST CASE PRIORITIZATION TECHNIQUES IN REGRESSION TESTING

K. KALAIVANI[1], A. SARITHA[2] & K. ULAGAPRIYA[3]

[1,2]Assistant Professor, Department of Computer Science and Engineering, Vels University, Chennai, Tamil Nadu, India

[3]PG Student, Department of Computer Science and Engineering, Vels University, Chennai, Tamil Nadu, India

ABSTRACT

Test case prioritization techniques schedule test cases in multiple test suites for execution in an order that attempts to increase their efficiency at meeting some performance goals like increasing the rate of fault detection, reducing the time required to detect the faults in the testing process. The increased rate of fault detection during testing can provide faster feedback on the system under test. The test case prioritization techniques are widely used in regression testing where the software is retested after modifications. There are several techniques to prioritize test cases in regression testing. These techniques are grouped based on the total coverage of code components, coverage of code components not previously covered, and the ability to reveal faults in the code components. These techniques are applied on various test suites and the rate of fault detection is measured.

KEYWORDS: Test Case Prioritization, Software Testing, Regression Testing, Test Suite

INTRODUCTION

Software testing can be stated as the process of validating and verifying that a computer program/application/ product meets the requirements that guided its design and development. A primary purpose of testing is to detect software failures so that defects may be exposed and corrected. The scope of software testing often includes analysis of code as well as execution of that code in various environments and conditions as well as examining the aspects of code. There are many approaches to software testing. Reviews, walkthroughs, or inspections are referred to as static testing, whereas actually executing programmed code with a given set of test cases is referred to as dynamic testing.

Regression Testing is a type of software testing that seeks to uncover new software bugs, or *regressions*, in existing functional and non-functional areas of a system after changes, such as enhancements, patches or configuration changes, have been made to them. Regression testing is an integral part of the extreme programming software development method. Regression testing is nothing but full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.

This testing is done to make sure that new code changes should not have side effects on the existing functionalities. Retesting is different from regression testing. Retesting means testing the functionality or bug again to ensure the code is fixed. If it is not fixed, defect needs to be re-opened. If fixed, defect is closed. Regression testing means testing your software application when it undergoes a code change to ensure that the new code has not affected other parts of the software.

The purpose of regression testing is to confirm that a recent program or code change has not adversely affected existing features. It was found from industry data that good number of the defects reported by customers were due to last minute bug fixes creating side effects and hence selecting the test case for regression testing is an art and not that easy.

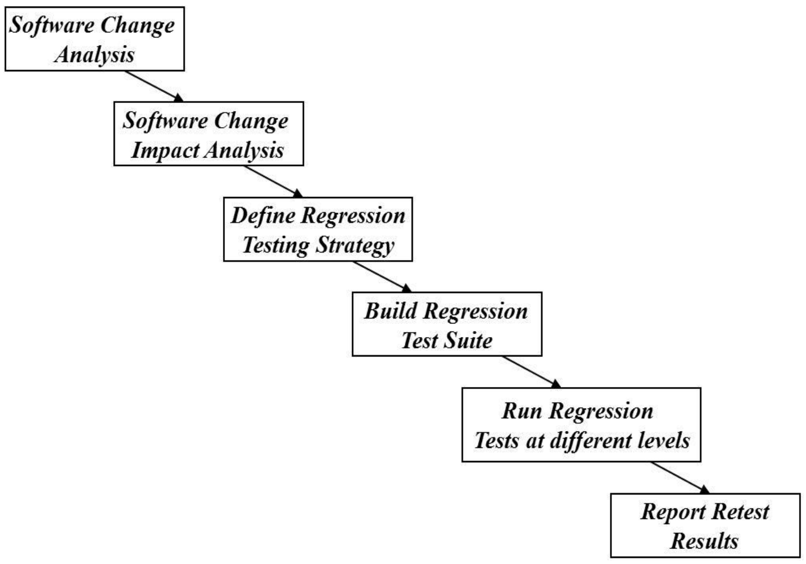
Software Change Analysis
Software Change Impact Analysis
Define Regression Testing Strategy
Build Regression Test Suite
Run Regression Tests at different levels
Report Retest Results

Figure 1

Effective Regression Tests can be done by selecting following test cases -

- Test cases which have frequent defects
- Functionalities which are more visible to the users
- Test cases which verify core features of the product
- Test cases of Functionalities which has undergone more and recent changes
- All Integration Test Cases
- All Complex Test Cases
- Boundary value test cases
- Sample of Successful test cases
- Sample of Failure test cases

The process of regression testing comprises of two basic activities:-i) Regression test selection problem- selecting which test cases should be re-executed and ii) Reproducibility problem - performing retesting with the help of these selected test cases.

The problem of regression test selection can be solved by prioritizing test cases. Regression test prioritization techniques reorder the execution of test suite in an attempt to ensure that defects are released earlier in the test execution phase. Regression test prioritization techniques attempt to reorder a regression test suite such that those tests with higher priority are executed earlier in the regression testing process than those with the lower priority.

In this paper, we describe several techniques for prioritizing test cases in regression testing. We then describe several empirical studies we accomplished with these techniques to assess their ability to improve rate of fault detection, a measure of how quickly faults are detected within the testing process. An improved rate of fault detection during regression testing provides earlier feedback on a system under test and lets debugging activities begin earlier than might otherwise be possible. Our results specify that test case prioritization can considerably improve the rate of fault detection of test suites. Our results also highlight several cost-benefit tradeoffs between various techniques.

LITERATURE REVIEW

The existing prioritization techniques are failed to prioritize multiple test suites and test cases with same priority values. Consequently, those techniques are inefficient to prioritize tests in the large commercial systems. although there are *safe* regression test selection techniques (e.g. [3, 7, 29, 34]) that can ensure that the selected subset of a test suite has the same fault detection capabilities as the original test suite, the conditions under which safety can be achieved do not always hold [28, 29]. When the time required to re execute an entire test suite is short, test case prioritization may not be cost-effective — it may be sufficient simply to schedule test cases in any order. When the time required to execute an entire test suite is sufficiently long, however, test case prioritization may be beneficial, because in this case, meeting testing goals earlier can yield meaningful benefits.

PRIORITIZATION TECHNIQUES

Test case prioritization involves scheduling test cases in an order that escalates their effectiveness in meeting some performance goals.

There are several prioritization techniques that can improve the rate of fault detection.

Table 1: Prioritization Techniques

Mnemonic	Description
P1	No Prioritization
P2	Randomized Prioritization
P3	Optimized Prioritization
P4	Code coverage Prioritization
P5	Additional code coverage Prioritization
P6	Branch coverage Prioritization
P7	Additional Branch coverage Prioritization
P8	Fault revealing Prioritization
P9	Additional Fault revealing Prioritization

No Prioritization

The test cases are executed in the existing order after the modifications are made in the code hence there is no prioritization.

Randomized Prioritization

The test cases in the test suite are executed in a random order and the results are observed.

Optimized Prioritization

The program that contains the known faults is used in order to assess the effects of prioritization techniques on rate of fault detection in the case of optimized prioritization.

```
Function opt_priority (program P, known faults)
    For all test cases in a test suite T
        Find the test case that reveals the known faults in P
        Find the ordering of test cases in a test suite
    Return the optimal ordering of the test cases
End
```

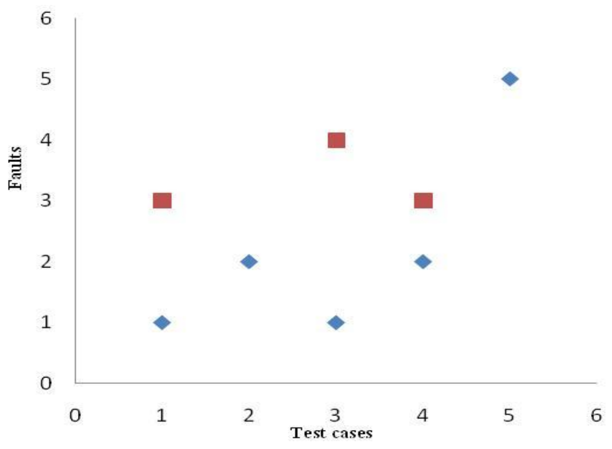

Figure 2

This is not a practical technique, as it requires a-prior knowledge of the existence of faults. A greedy "optimal" prioritization algorithm is employed. Given a program P with a set of faulty versions, a test suite T , and information on which test cases in T expose which faults, our algorithm iteratively selects the test case in T that exposes the most faults not yet exposed by a selected test case, until test cases that expose all faults have been selected. It is observed that this greedy prioritization algorithm may not always choose the optimal test case ordering.

Code Coverage Prioritization

In Code coverage prioritization, test cases are prioritized in terms of total number of statements by sorting them in order of coverage achieved. If test cases are having same number of statements they can be ordered pseudo randomly. By instrumenting a program we can determine, for any test case, which statements in that program were exercised (covered) by that test case. We can then prioritize test cases in terms of the total number of statements they cover, by counting the number of statements covered by each test case, and then sorting the test cases in descending order of that number. (When multiple test cases cover the same number of statements, an additional rule is necessary to order these test cases; we order them randomly.) Statement coverage (St) is calculated for each test case that is executed in a test suite. Mathematically it is calculated as the number of statements covered upon the total number of statements per cent.

Statement Coverage (St) = (No. of. Statements Covered / Total No. of Statements) * 100

```
     Procedure Codeproc1
1.   Statement1
2.   Statment2
3.   If (cond1)then
4.      Exit
     End if
5.   While (cond2) do
6.   If (cond3) then
7.        Statement4
        End if
```

8. End while
9. Statement5
10. Statment7

Table 2: Code Coverage

Statement	Test Case1	Test Case2	Test Case3
1	X	X	X
2	X	X	X
3	X	X	X
4	X		
5		X	X
6		X	
7		X	
8			
9		X	X
10		X	X

Additional Code Coverage Prioritization

In fact, Elbaum [39], [41], [42] extended the selection technique in order to prioritize the test cases in a test suite, that is, to place the test cases in non-decreasing order with respect to their perceived likelihood of revealing defects. Elbaum’s approach, which he called additional coverage prioritization, involves running a greedy coverage maximization algorithm repeatedly on the set of test cases that have not yet been prioritized. The priority of a test case corresponds to the order in which it is selected during this process. The earlier a test case is selected, the higher its priority is. In the sequel, Leon [3] referred to the prioritization technique as repeated coverage maximization.

Additional statement coverage prioritization requires coverage information for each unprioritized test case to be updated following the choice of each test case. Given a test suite containing m test cases and a program containing n statements, selecting a test case and readjusting coverage information has cost O(m n) and this selection and readjustment must be performed O(m) times. Therefore, the cost of additional statement coverage prioritization is O (m2 n), a factor of m more expensive than total statement coverage prioritization.

Total Branch Coverage Prioritization

By instrumenting a program, we can determine, for any test case, the number of decisions (branches) in that program that were exercised by that test case. We can prioritize these test cases according to the total number of branches they cover simply by sorting them in order of total branch coverage achieved [5]. This prioritization can thus be accomplished in time for programs containing branches.

Procedure Branchproc1

1. *Statement1*
2. *Statment2*
3. *If (cond1) then*
4. *Exit*
5. *End if;*

```
6. If (cond2) then
7.     Statement3
   Else
8.     Statemnt4
9. End if
```

Table 3: Additional Code Coverage

Statement	Test Case 1	Test Case 2	Test Case 3
Entry	X	X	X
3.True	X		
3.False		X	X
6.True		X	
6.False			X

Additional Branch Coverage Prioritization

Total branch coverage prioritization schedules test cases in the order of total coverage achieved. However, having executed a test case and covered certain branches, more may be gained in subsequent test cases by covering branches that have not yet been covered [7]. Additional branch coverage prioritization iteratively selects a test case that yields the greatest branch coverage, and then adjusts the coverage information on subsequent test cases to indicate their coverage of branches not yet covered, and then repeats this process, until all branches covered by at least one test case have been covered. Having scheduled test cases

In this fashion, we may be left with additional test cases that cannot add additional branch coverage. We could order these next using any prioritization technique; in this work we order the remaining test cases using total branch coverage prioritization [3]. Because additional branch coverage prioritization requires recalculation of coverage information for each unprioritized test case following selection of each test case.

Fault Revealing Prioritization

Statement and branch-coverage-based prioritization considers only whether a statement or branch has been exercised by a test case [2]. This consideration may mask a fact about test cases and faults: the ability of a fault to be exposed by a test case depends not only on whether the test case reaches (executes) a faulty statement, but also, on the probability that a fault in that statement will cause a failure for that test case [6]. Although any practical determination of this probability must be an approximation, we wished to determine whether the use of such an approximation could yield a prioritization technique superior in terms of rate of fault detection than techniques based on simple code coverage.

Additional Fault Revealing Prioritization

Analogous to the extensions made to total branch (or statement) coverage prioritization to additional branch (or statement) coverage prioritization, we extend total FEP prioritization to create additional fault-revealing-potential (FRP) prioritization [4]. This lets us account for the fact that additional executions of a statement may be less valuable than initial executions.

COMPARISON OF PRIORITIZATIONS TECHNIQUES

A case study was performed to analyze the impact of test cost and the fault severity of the prioritization techniques. The results showed that all the techniques performed better than the original and the random order prioritization. Also, the additional fault revealing potential prioritization performed the best. Moreover, the branch coverage techniques were better than the corresponding statement coverage techniques. Erlbaum et al. [1] examined two techniques, total/additional function coverage along with the random and the optimal ordering to understand the effect of change on the cost effectiveness of the regression testing techniques.

They made use of a large number of measures to accomplish the comparative case study. The analysis found that the change attributes played a significant role in the performance of the techniques. Also, the additional function coverage technique outperformed the total function prioritization technique regardless of the change characteristics. The total technique gave varied results and was sometimes worse than random prioritization. An empirical comparison among four different prioritization techniques was put forward by Leon et al. in 2003 [2].

These techniques included test suite minimization, prioritization by additional coverage, cluster filtering and failure pursuit sampling (FPS). The former two techniques were broadly classified as coverage based and the latter two as distribution based. The comparisons yielded the following findings: when the sample sizes are small, basic coverage maximization can detect the facts efficiently; one per cluster sampling achieves comparably good results and at the same time does not achieve full coverage; for large sampling sizes, FPS is more efficient than cluster sampling.

CONCLUSIONS AND FUTURE WORK

In this paper we discussed about Regression test selection and Test Case Prioritization Selection. Regression testing is a style of testing that focuses on retesting after changes are made. In traditional regression testing, we reuse the same tests (the regression tests).

Test case prioritization involves scheduling of test cases in an order that increases their effectiveness in meeting some performance goals. One such goal is APFD(average percentage of faults detected) measure that increases the chances of finding faults earlier in the software testing lifecycle and may facilitate the ultimate goal of software development by improving quality. We want to use many more techniques which help in this direction, particularly the data mining techniques. As it is well known that test suite development is quite expensive and more often, running an entire suite is not possible in its entirety as it takes more time to run and more human resources are required to actually execute them. This method can address this issue very successfully.

An effective regression strategy, save organizations both time and money. As per one of the case study in banking domain, regression saves up to 60% time in bug fixes (which would have been caught by regression tests) and 40% in money.

REFERENCES

1. S.Elbaum, P.Kallakuri, A.Malishevsky, G.Rothermel and S.Kanduri, “Understanding the effects of changes on the cost-effectiveness of regression testing techniques”, Journal of Software Testing, Verification, and Reliability, Vol.12, No.2, pp.65-83, 2003.
2. D. Leon,A. Podgurski, "A Comparison of coverage based and distribution based techniques for filtering and prioritizing test cases", In Proceedings of the 14th International Symposium on software reliability engineering(ISSRE 03),pp 442-453,2003.
3. SebastianElbaum, AlexeyG.Malishevsky, Gregg Rothermel.”Test Case Prioritization” IEEE transactions on

software Engineering, vol.28, No.2, February 2002.

4. Kuo –Chung Tainand Yu Lei. “A Test generation strategy for Pairwisetesting” IEEE transactions on software Engineering, vol.28, No.1, January 2002.

5. Wei-Tek Tsai and Lian Yu, Feng Zhu, Ray Paul. “Rapid embedded system testing using verification patterns”. IEEE software 2005.

6. P. M. Duvall, S. Matyas, and A. Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Addison Wesley, Upper Saddle River, NJ, 2007.

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