IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727, Volume 16, Issue 4, Ver. IV (Jul – Aug. 2014), PP 43-51
www.iosrjournals.org

Importance of Selecting Test Cases for Regression Testing

Kamna Solanki[1] , Dr. Yudhvir Singh[2]
[1,2] *(University Institute Of Engg. and Technology, M.D.University, India)*

Abstract: *There is a well-known discussion stating that "Under Testing is a crime and over testing is a Sin". Regression testing also faces the same challenge regarding the selection of test cases which needs to re-run when some changes are made in the source code. Regression Testing assures changed programs against unintended amendments. Since several well-known software failures can be blamed on not testing changes and amendments in a software system thoroughly and properly, many techniques have been developed to support efficient and effective Regression Testing. This paper discusses the Regression Testing Process in detail to describe the importance of selecting Test Cases for Regression Testing..*
Keywords: *Test case Prioritization, Regression Testing, Software Testing, Test Cases, Test Suites*

I. Introduction

Software is expected to work correctly and efficiently along with meeting customer's changing demands, first time and every time, consistently and predictably. Earlier software systems were used for back-office and non-critical operations of organizations. Now more and more critical applications are implemented globally. This increased expectation for error-free functioning of software has increased the demand for quality output from software vendors [**1**]**.** Software Testing is the process used to help and identify the correctness, completeness, security, reliability & quality of developed software.

Software testing is the process of validation and verification of the software product. According to Myers "Software Testing is the process of executing a program with the intent of finding errors" [**2**]. Effective software testing will contribute to the delivery of reliable and quality oriented software product, more satisfied users, lower maintenance cost, and more accurate and reliable result. However, ineffective testing will lead to the opposite results; low quality products, unhappy users, increased maintenance costs, unreliable and inaccurate results. Hence, software testing is a necessary and important activity of software development process. Software testing process typically consumes at least 50% of the total cost involved in software development [**3**]**.** Software development organizations spend considerable portion of their budget and time in testing related activities.

Studies by IBM and others have shown that the cost of correcting a fault after coding is at least 10 times as costly as that before it, and the cost of correcting a production fault is at least 100 times **[4]**. Similar observations have been reported in other literature. The cost-escalation factors range from 5x to 100x, depending on the types and sizes of the software systems **[5]**. Thus, program testing should be as thorough as possible to help software developers detect any failures so that faults will not be propagated through to the final production software, where the cost of removal is far greater. Among various quality assurance techniques, testing remains a popular and important one for improving software quality **[6]**. The potential cost savings from handling software errors within a development cycle, rather than the subsequent cycles, has been estimated at nearly 40 billion dollars by the National Institute of Standards and Technology **[7]**. Current testing methods are often inadequate, and hence reduction of software bugs and errors is an important area of research with a substantial payoff.

The effectiveness of the verification and validation (testing) process depends upon the number of software defects found and rectified before releasing the software to the customer side. This in turn depends upon the quality of test cases generated and order or priority in which they are executed. The solution is to choose the most important and effective test cases which are likely to find more faults of certain type and running them on priority; which in turn leads to test suite prioritization. In regression testing, out of all software test life cycle phases, test case minimization and prioritization phase are most crucial. Test case preparation phase scores over other phases in terms of being a candidate for prioritization and optimization **[8].**

II. Software Testing Terminologies

Software testing is the process of exercising a program with well designed input data with the intent of observing failures. In other words, "Testing is the process of executing a program with the intent of finding errors". Testing identifies faults, whose removal increases the software quality by increasing the software's potential reliability. Testing also measures the software quality in terms of its capability for achieving correctness, reliability, usability, maintainability, reusability and testability **[9] [10]**.

2.2.1 Software Testing Techniques:

A testing technique specifies the strategy to select input test cases and analyze test results **[11] [2]**. Different testing techniques reveal different quality aspects of a software system, and there are three major categories of testing techniques such as "Black Box" (or functional testing), "White Box" (or structural testing) and "Grey Box testing" (or translucent testing) **[12] [10] [13]**.

Black Box Testing: The software program or System Under Test (SUT) is considered as a "black box" in this type of testing. The selection of test cases for functional testing is based on the requirements or design specifications of the software entity under test. Examples of expected results sometimes are called test oracles, which include requirement/design specifications, hand calculated values, and simulated results. External Behavior of the software entity is the main attraction of functional testing.

White Box Testing: The software entity is considered as a "white box". The selection of test cases is based on the implementation of the software entity. The main focus of such test cases is to cause the execution of specific spots in the software entity, such as specific statements, program branches or paths. The expected results are evaluated on a set of coverage criteria like path coverage, branch coverage, and data-flow coverage. Internal Structure of the software entity is the main focus of structural testing.

Gray-box testing: It is a combination of white-box testing and black-box testing. The aim of this testing is to search for the defects if any due to improper structure or improper usage of applications. Gray-box testing is also known as translucent testing.

2.2.2 Test Case and Test Suite

A test case is a set of conditions or variables under which a tester will determine whether a system under test satisfies requirements or works correctly. A test case answers the question: "What am I going to test?" You develop test cases to define the things that you must validate to ensure that the system is working correctly and is built with a high level of quality **[9]**.

Test Suite

The most common term for a collection of test cases is a test suite. A test suite is a collection of test cases that are grouped for test execution purposes. The test suite often also contains more detailed instructions or goals for each collection of test cases. It definitely contains a section where the tester identifies the system configuration used during testing. A test suite is a set of several test cases for a component or system under test (SUT), where the post condition of one test is often used as the precondition for the next one **[11]**.

2.2.3 Code Coverage

Given a set of things that could be tested, code(test) coverage is the portion actually tested. Code Coverage is an often used metric in software testing. It provides a quantitative measurement of the exercised source code after test execution, usually expressed either as a percentage or a ratio of the number of covered source code lines and the total number of lines **[13] [10].**

Code coverage analysis is a white box testing technique which requires access to the source code of the System Under Test (SUT). Analysis helps in finding out which parts of the source code have not yet been tested, and vice versa. The main idea behind coverage testing is that the unexercised parts of the source code are presumed to contain errors. In addition, coverage analysis can help identify redundant test cases which do not increase coverage. The promise behind using any measure of code coverage is that the more code coverage a test suite achieves, the more confidence it provides regarding the reliability of the system under test. Although, a 100% code coverage does not mean that a software is defect free **[14]**.

2.2.4 Goal of Testing

In large-scale software development, testing accounts for a substantial portion of the development cost. An important goal of testing is to expose defects in software; detection early in the development cycle saves time and resources. Testing of software, therefore, occurs continuously throughout the development cycle. For example, developers may run a few simple pre-check-in tests to ensure that their code changes will not keep the program from being built (compiled and linked) and to catch a moderate number of defects. Later, after the whole program is built, verification tests are run before it is released for full testing. These tests are not intended to be exhaustive and must complete within a limited time.

Full testing, running all tests in the test suite, is intended to be exhaustive and may take days or weeks to run. Even during full testing, it remains advantageous to detect the defects as early as possible, e.g., on day 1 rather than day 21. Early detection of defects enables developers to start sooner on finding and fixing defects for the next iteration. Once the software is released, software patches to update released software also go through

the regular test process. However, there are certain rare circumstances, such as emergency patches for critical bugs, when tests must be run under severe time constraints and certain tests must be skipped**[15].**

To address these scenarios effectively, developers and testers must be able to run the right tests at the right time. New defects recently introduced into the system are most likely to be from recent code changes. Therefore, an effective strategy is to focus testing efforts on parts of the program affected by changes. Whenever a developer checks in code, a set of tests can be dynamically selected to exercise parts of the program affected by the developer's code changes, subject to a specified time limit. This same technique can also be applied later to testing the system after it is built. Even for full testing, we first want to run tests that will exercise the affected parts of the program, before any other tests.

III. Regression Testing

Regression testing is an important and yet time consuming software development activity. It executes an existing test suite (denoted by T) on a changed program (denoted by P) to assure that the program is not adversely affected by unintended amendments.

IEEE defines regression testing as follows**[16]:**

"Regression testing is 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 requirements."

Regression testing is a testing activity that is performed to provide confidence that changes do not harm the existing behavior of the software. Regression testing is performed when changes are made to existing software; the purpose of regression testing is to provide confidence that the newly introduced changes do not obstruct the behaviors of the existing, unchanged part of the software. Regression testing is also an important software maintenance activity that involves repeatedly running a test suite whenever the program under test and/or the program's execution environment changes. Executing a regression test suite upon the introduction of either a defect fix or a new feature ensures that the modification of the program does not negatively impact the overall correctness.

Holopainen **[17]** has identified eight regression testing stages for software components. It begins from scratch, when there are no test cases for the component. After that, test cases are developed and testing continues until no defects are detected. In this model, K represents a tested component and K' represents its new or changed version. Respectively, T represents the test cases corresponding to the component K and T' represents new or modified test cases for the component K'. The states are then described as follows:

1. When the component K is tested for the first time, corresponding test cases T are developed.
2. The component K is tested against the test cases T. If defects are found, they must be verified and fixed (go to state 3). If no defects are detected, the testing ends (go to state 7).
3. Defects are verified and fixed. After that, a changed version K' of the component K is developed.
4. Regression test cases T' are selected from the original test cases T. Also, new test cases can be created to cover the fixed functionality of the component. These new test cases are added to both T and T'.
5. The component K' is tested against the new regression test cases T'. If no defects are detected, testing ends (go to state 7).
6. Defects that are detected on regression testing are verified and fixed. After that, new test cases are selected or created for the fixed component (go back to phase 4).
7. Testing is finished.

However, regression testing can be prohibitively expensive, particularly with respect to time, and thus accounts for as much as half the cost of software maintenance. An industrial collaborator reported that for one of its products of approximately 20,000 lines of code, the entire test suite required seven weeks to run **[18.19,20].**

3.1 Regression Testing Framework

Figure 1 provides an overview of a model for the regression testing process**[19][21][22].** In the general regression testing (GRT) framework, we apply selection, reduction, and/or prioritization to the test suite and then use the modified suite during many subsequent rounds of test suite execution **[20,21].** This cost-effective approach to testing is motivated by empirical studies demonstrating that the adequacy of a test suite does not markedly change across subsequent versions of a program. Alternatively, the version specific regression testing (VSRT) model suggests that the test suite should be re-analyzed after each modification to the program under test. VSRT requires efficient implementations of the (i) test suite executor, (ii) test coverage monitor, and (iii) selection, reduction, and prioritization techniques. If the method for constructing the modified test suite is expensive, then the GRT framework supports the amortization of this cost over many executions of the tests.

Yet, VSRT is more likely to improve the effectiveness of regression testing because it always leverages the most current information about the program and the tests. Furthermore, testers should consider the VSRT approach whenever the program and/or the test suite undergo a series of substantial changes. Of course, any regression testing approach that can efficiently operate in a version specific fashion should also enable GRT **[23][24].**

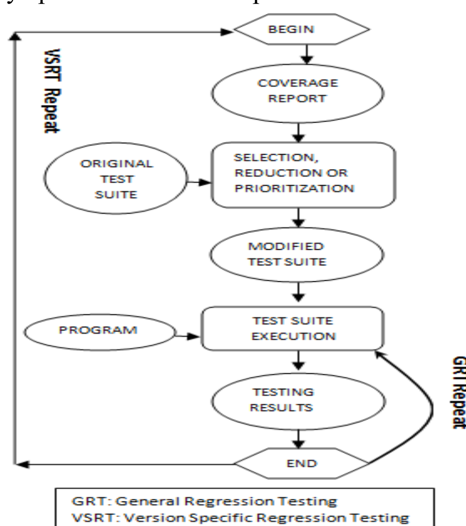

Fig1. Regression Testing Framework

IV. Classification Of Test Cases For Regression Testing

Selecting test cases for regression testing is one of the most important question. Let P be a program, pi its modified version and T a test suite for P. Also let S stand for specified output of P, S' stand for specified output for pi and D denote the domain of some specification. A test case t is a three-tuple < n, i, S(i) >, where n is a number (identifier) of the test case, i is a set of inputs and S(i) is a set of specified outputs.

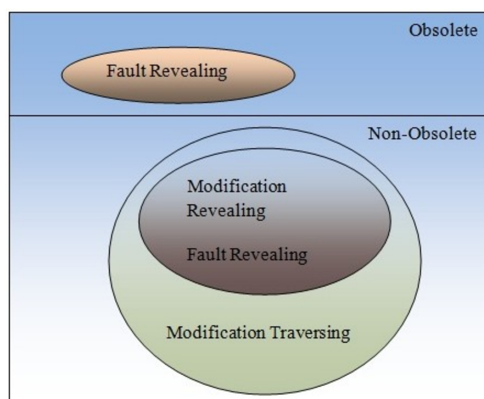

Fig. 2 Obsolete and Non-Obsolete Test Cases

There is a notion of obsolete test cases which either specify invalid input for pi or specify an invalid input-output relation for P'. It can be formalized as follows**[22][25]:**

t is obsolete if i tJ- D(S') VS'(i) -1= S(i)

Test case t is called fault-revealing if it makes pi fail and thus detects a fault in P'. pi fails when, after executing t, the output of pi does not satisfy its specification.

Formally: t is fault-revealing

if -,obsolete(t) /\ Pl(i) =I- B'(i)

Unfortunately, there is no effective procedure for selecting fault-revealing test cases from T. t is called modification-revealing if it generates different outputs for P and pl.

Modification-revealing test cases are formalized as following:

t is modification-revealing if -,Obsolete(t) /\ PI(i) =I- P(i)

Modification-revealing test cases are fault-revealing under the following two assumptions:

1. t ϵ T when P was tested with t, P halted and produced a correct output.
2. There is an effective procedure for identifying obsolete test cases for P'.

Even if these two assumptions hold, there is still no effective procedure for precisely identifying modification-revealing test cases.
Another variant of test cases is modification-traversing. They execute new or modified code in pI or used to execute deleted code in P. These test cases can be a superset to modification-revealing test cases under the following assumption:
When P is tested with t, all factors that might affect output of P except the code are kept constant in exactly the same state as they were when t was executed on P.

The figure 2 visually illustrates the relations between different types of test cases. As can be seen from figure 2 ,fault-revealing test cases can be both obsolete and non-obsolete. Obsolete fault-revealing test cases are those that do not halt for pI, but were supposed to halt and produce a certain output. This itself is a fault and the only way to identify this kind of test case is to execute every obsolete test case to check if they do not exceed a certain time limit. If they do then they are fault-revealing. Also note that all modification-revealing and modification-traversing test cases are non-obsolete, because they specify valid inputs and valid input-output relation for P.
There are several properties of RTS techniques that describe their effectiveness and indicate why one would choose one RTS technique over the others **[20,21,22]**. They are:
1. Inclusiveness: the extent to which an RTS technique includes modification revealing test cases in a subset. If an RTS technique chooses all modification revealing test cases from T then it is called safe.
2. Precision: the extent to which an RTS technique omits non-modification-revealing test cases in a subset.
3. Efficiency: is estimated from the computational time that a RTS technique requires. Any effective RTS technique should satisfy the following:
Tall > Tselection +Tsubset where
Tall - time to run the whole test suite;
Tselection - time spent to effectively select a subset of a test suite that will test only necessary part of a software;
Tsubset - time to run a subset of a test suite.

4. Generality: is the ability to handle different languages, language constructs and the ability to test real applications.

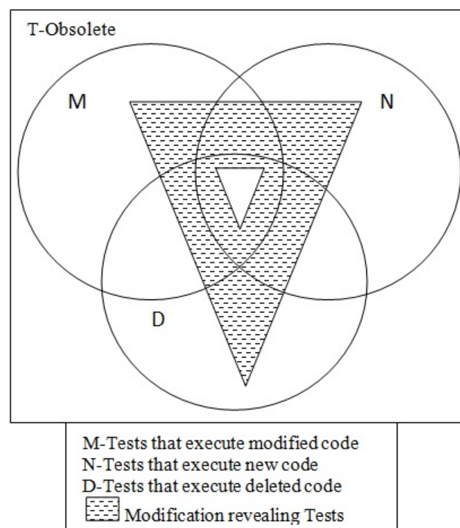

Fig. 3: Types of Test Cases

When discussing different existing RTS techniques it is very convenient to show their inclusiveness and precision properties in a graphical way such as shown in the following figure. In this figure, circles represent modification-traversing test cases such as those that execute new or modified code in pI or used to execute code in P that was deleted in P'. Some test cases can execute some or all of them, that is why circles intersect in the figure. Dashed area represents modification-revealing test cases. Since there are modification-traversing test cases that are not modification-revealing there are areas inside of the circles that are not dashed.

V. Selecting Test Cases For Regression Testing

Selecting Test cases for regression testing is one of the most important task. There are two approaches for selecting test cases for regression testing:

5.1 Re-Test All Method

One obvious way to perform regression testing is to run the entire existing test suite of the program on its modified version. This is the so called "retest all" technique. The following figure 4 depicts the worst case selection which is basically a retest-all technique.

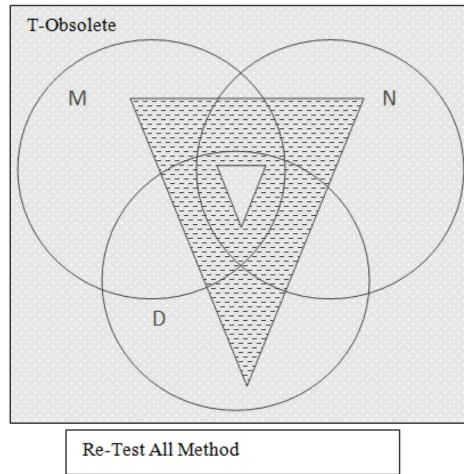

Fig. 4 Re-Test All Method

During evolution of some software products, their test suites become extremely large. So it would be very costly to run the whole test suite both in terms of time and human effort, which eventually ends up being very expensive in terms of money as well. So,this is impractical to retest the entire test suite. Figure 5 describes the balancing process of regression testing between the limited resources and increasing test suite size.

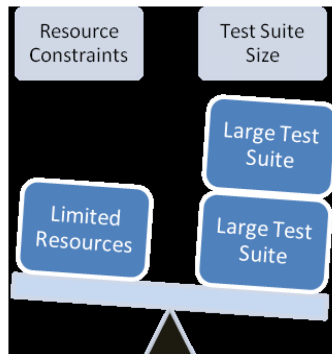

Fig. 5 Resource Constraints vs. Test Suite Size in Regression Testing

5.2 Selective Re-Testing Method

While Re-test All Method is impractical because it requires the re-running the entire test suite. On the other hand, if only a few test cases are run, it might be insufficient for identifying many of the faults in a modified software, so the whole regression testing will fail as being unreliable. In a paper , Harrold et al.**[18, 20]** investigated a random test selection technique, where using the mathematical concept of Chernoff bound, they calculate what the number of test cases should be to gain a certain percentage of confidence in performed regression testing. This technique does not depend on the size of the test suite. But because the selection of the test cases is random, there is still a danger that some software faults might go undetected. Even if some test cases fail during regression testing, it still does not mean that this is a fault. It might be the case that the test case is not satisfactory for the modified software as it was for a previous version. This is why, after identifying such test cases, further analysis is needed to see if those test cases are adequate.

Best case test selection is followed when only modification-revealing test cases are selected (optimum). The following figure 6 depicts the Optimum Test Case Selection.

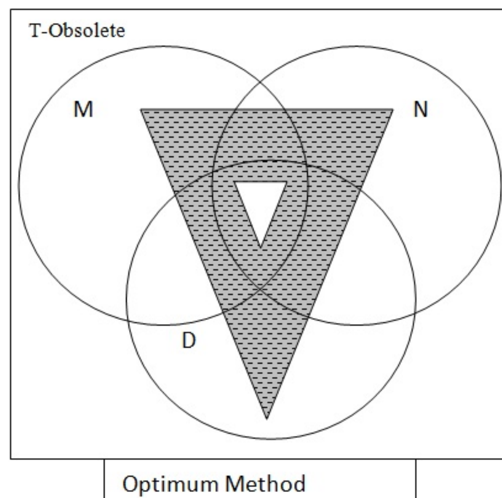

Figure 6 Optimum Test Case Selection for Regression Testing

Researchers have developed diverse techniques to optimize the test case selection process for improving the cost-effectiveness of regression techniques. These techniques are discussed and formally described by Yoo and Harman as **[26]:**

5.2.1 Test Suite Minimization (TSM)/ Test Suite Reduction (TSR)

These techniques remove the redundant test cases permanently to reduce the size of test suite. However, the fault detection capability of a test suite may decrease due to reduction in the number of test cases.
The Formal Test Suite Minimization (TSM)Problem can be defined as:
Given: a set (test suite) T of candidate test cases t1, t2, ..., tn and some set of coverage requirements R, where each test case covers a set of software requirements r1, r2, ..., rn, respectively, such that r1 ∪r2 ∪... ∪rn = R
Problem: find a minimally-sized subset of test cases T′ ⊆T, comprised of tests t′ 1, t′ 2, ..., t′ m, each test covering a set of software requirements r′ 1, r′ 2, ..., r′ m , respectively, such that r′ 1 ∪r′ 2 ∪· · · ∪r′ m = R

A test requirement is a certain part of a software that the test case must satisfy or cover. The test suite minimization problem is an instance of the more general set- cover problem, which when given as input a collection S of sets, each set covering a particular group of entities, is to find a minimally-sized subset of S providing the same amount of entity coverage as the original set S. The test suite minimization/Optimization process involves generation of effective test cases in a test suite that can cover the given SUT within less time. The behavior of tests effectiveness with respect to allocated resources (work, money, time, computation) is asymptotic. This means that, among all possible tests, only a part of them is economically useful. Test suite minimization is an optimization problem to find a minimally-sized subset of the test cases in a suite that exercises the same set of coverage requirements as the original suite. The key idea behind minimization techniques is to remove the test cases in a suite that have become redundant in the suite with respect to the coverage of some particular set of program requirements.

5.2.2 Test Case Selection (TCS)/ Regression Test Selection (RTS):

These techniques select some of the test cases and focus on the ones that test the changed part of the software. Contrary to the TSR, RTS does not removes test cases, but selects the test cases that are related to the changed portion of the source code.
The Formal Test Case Selection Problem
Given: The program, P, the modified version of P, P', and a test suite, T.
Problem: Find a subset of T, T', with which to test P'.

Both the selection and the minimization technique's goal is to reduce the size of the test suite; however, the difference is that in the selection technique we try to select test cases that are "modification aware" , which means that they are relevant to modified parts of the software and thus this technique is related to white box testing where software's source code is examined.

5.2.3 Test Case Prioritization (TCP):

Test case prioritization techniques schedule test cases for execution in an order that maximizes some objective function or prioritization goal **[30,31].** The purpose of prioritization is to increase the likelihood that this objective function would be better met if the test cases used for regression testing are executed in the given

order than if the test cases were executed in an ad-hoc order. These techniques identify the efficient ordering of the test cases to maximize certain properties such as rate of fault detection or coverage rate.
The Formal Test Case Prioritization Problem

Given: a test suite, T, the set of permutations of T, PT, and a function from PT to real numbers, f : PT→ R
Problem: Find T' E PT s.t. V(T"IT" E PT, T" f:: T' : f(T') ~ f(T")).

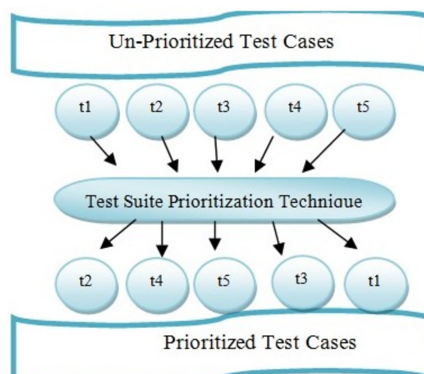

Fig. 7 Prioritized Test Cases

While both TCM and TCS techniques reduces testing time, they can omit some significant test cases that can detect certain types of faults and hence can increase the software cost **[28]**. However, TCP techniques uses the entire test suite and reduce testing cost by parallelization of the debugging and testing activities.TCP improves the cost-effectiveness of regression testing and provides several benefits such as earlier defect detection or earlier feedback to tester. Many papers provide evidence that these prioritization techniques can be beneficial to the regression testing **[31,32]**.

Whenever the researchers deveop any novel test case prioritization technique, the first step is to define a prioritization goal i.e. the objective function that needs to be minimized or maximized **[29,30].** Afterwards, the technique is developed and its effectiveness is validated using some well known Metrics like APFD (Average Percentage of Faults Detected) and Modified APFD etc.

5.3 Importance of Filtering Appropriate Test Cases for Regression Testing

Software testing and retesting occurs continuously during the software development lifecycle to detect errors as early as possible. The sizes of test suites grow as software evolves. Regression testing is used to ensure the validity of the changed software. Due to time and budget constraints; the entire test suite could not be executed during regression testing. Hence it becomes an essential to minimize the test suite and choose a subset of test cases from test suite which covers the modifications and will be executed in least time and has the capability to cover all the faults.

Regression testing is a very costly process and consumes significant amounts of resources. During regression testing, an already designed (original) test suite is available for reuse. A regression test filtration may help us to select an appropriate number of test cases from this test suite. The simplest technique is to run all test cases for verifying the modified program. This is the safest technique, but it is practical only when the size of test suite is fairly small. To be worthwhile, the sum of the cost of the filtering process and the costs of executing and auditing the selected tests should be less than the cost of executing and auditing all of the tests in the original test suite.

VI. Conclusion

Quicker time-to-market is the need of the hour. This is true more so for testing since it is always at the tail end of the software development lifecycle. The product with more number of features to be modified or added will have more amount of testing to be done, and more numbers of test cases. Such situation will have direct impact on delivery and schedule of the product and would increase the cost of the product. Hence the selection of effective test suites that finds the maximum no. of defects at the earliest and covers all the functionalities and risk prone areas of the product is most important aspect of regression testing.

References

[1] D. Srinivasan and R. Gopalaswamy, Software Testing: Principles and Practices, 1st ed., New Delhi: Person Education, 2006
[2] G. Myers, The Art of Software Testing, NY,USA: John Wiley, 1979
[3] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation and balancing automated and manual testing with opportunity cost," in International Workshop on Automation of Software Testing, 2006.
[4] W.E.Perry, Effective Methods for Software Testing, John Wiley Publication, 2006.
[5] M. Grottke and K. Trivedi, "Fighting Bugs: Remove, Retry, Replicate," IEEE Transactions on Software Engineering, vol. 40, no. 2, pp. 107-109, 2007
[6] T. Shepard and M. Lamb, "More testing should be taught," Communications of ACM, vol. 44, no. 6, pp. 103-108, 2001.
[7] D. J. Berndt and A. Watkins, "High volume software testing using Genetic Algorithm," in Proceedings of 38th (IEEE) International Conference on System Sciences,, Waikolova, 2005
[8] A. Sahoo, J. Mishra and S. Dash, "Optimized software test management using risk based approach," International Journal of Software Engg. Applications, vol. 7, no. 3, pp. 179-193, 2013.
[9] A. Mathur, Foundation of Software Testing, Pearson Education, 2008.
[10] R. Pressman, Software Engineering: A Practitioner's Approach, 6th Edition, McGrawHill Publication, 2005.
[11] B. Beizer, Software Testing Techniques, Thomson Computer Press, 2nd Edition, 1990
[12] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong and Z. Guoliang, "Generating Test Cases from UML Activity Diagram based on Gray-Box Method," in Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC'04), 2004.
[13] G. J. Myers and C. Sandler, The Art of Software Testing, John Wiley, 2011.
[14] T. Parttimaa, Test Suite Optimization based on response code and code coverage, Master's Thesis, University of Oulu, 2013
[15] A. Srivastava and J. Thiagarajan, "Effectively Prioritizing Tests in Development Environment"., ISSTA , Proceedings of ACM Sigsoft International symposium on Software Testing and Analysis,pp 97-106, 2002.
[16] IEEE Standard Glossary of Software Engineering Terminology, IEEE Std. 729-1983, IEEE Press, 1983.
[17] Holopainen, J., "Regression tesing", Kuopion University, 2004. Accessed on 25 Jan 2014
[18] Rothermel, G., Harrold, MJ., "A Safe, Efficient Regression Test Selection Technique", ACM Transactions on Software Engineering and Methodology, Vol. 6, No. 2, pp 173-210, 1997.
[19] Rothermel, G., Roland J.U., and Chengyun C. Prioritizing test cases for regression testing. IEEE Transactions on Software Engineering, Vol. 27, No.10, 2001.
[20] Rothermel, G., and Harrold M. J., "Empirical studies of a safe regression test selection technique". IEEE Transactions on Software Engineering, Vol. 24, No.6,pp 401–419, June 1998.
[21] Rothermel, G., and Harrold M.J., "Analyzing regression test selection techniques". IEEE Transactions on Software Engineering, Vol. 22, No. 8 pp529-551, August 1996.
[22] Abdrakhmanov A.," A Regression Test Selection Technique Applied to Legacy Systems". McMaster University, 2010. Accessed on 25th Jan 2014
[23] Beizer B., "Software Testing Techniques, Van Nostrand Reinhold", Inc, New York NY, 2nd edition. ISBN 0-442-20672-0, 1990.
[24] Srikanth H., Williams L., and Osborne J., "System Test Case Prioritization of New and Regression Test Cases", Proceedings of the 4th International Symposium on Empirical Software Engineering (ISESE), pp 62–71. IEEE Computer Society, 2005.
[25] Kapfhammer G. M., "The Computer Science Handbook", chapter 105: Software Testing. CRC Press, Boca Raton, FL, second edition, 2004.
[26] Yoo S., and Harman M., "Regression Testing Minimisation, Selection and Prioritization : A survey," Journal of software testing , Verification and Reliability, Vol. 22, No. 2, pp. 67-120, 2012.
[27] Do H., and Rothermel G., "On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques", IEEE Transactions on Software Engineering, Vol. 32, No. 9, pp 733-752, 2006.
[28] Mathur A., "Foundation of Software Testing", Pearson Education, 2008.
[29] Tahat L. H., and Korel B., "Regression Test Suite Prioritization Using System Models," Journal of software testing, verification and reliability, vol. 22, no. 7, pp. 481-506, 2012
[30] Mohanty S., "A survey of model based test case prioritization," International Journal of Computer Science and Information Technologies, Vol. 2, No. 3, pp. 1042-1047, 2011.
[31] Kaur A., and Goyal S., "A Genetic Algorithm for regression test case prioritization using code coverage," International Journal on Computer Science and Engineering (IJCSE), Vol. 3, No. 5, pp. 1839-1847, 2011.
[32] Jacob P., and Ravi T., "Optimization of Test cases by Prioritization," Journal of Computer Science, vol. 9, no. 8, pp. 972-980, 2013