International Journal of Information Technology and Management

Vol. VIII, Issue No. XII, May-2015, ISSN 2249-4510

TEST CASES GENERATION: APPROACH TO REDUCE THE COMPLEXITY TO ANALYZE THE CODE

AN INTERNATIONALLY INDEXED PEER REVIEWED & REFEREED JOURNAL

www.ignited.in

International Journal of Information Technology and Management
Vol. VIII, Issue No. XII, May-2015, ISSN 2249-4510

Test Cases Generation: Approach to Reduce the Complexity to Analyze the Code

Sheo Kumar[1] Dr. Rajan Anand Malik[2]

[1]JJTU Scholar, Jhunjhunu

[2]Director JEMTEC Greater Noida

Abstract – ***The shorter life-cycle of software development, such as the one suggested by the agile programming discipline, also imposes restrictions and constraints on how regression testing can be performed within limited resources. Naturally, the most straightforward approach to this problem is to simply execute all the existing test cases in the test suite; this is called a retest-all approach. However, as software evolves, the test suite tends to grow, which means it may be prohibitively expensive to execute the entire test suite.***

Keywords: Software Testing, Test Cases

-------------X-------------

INTRODUCTION

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. It is a complex procedure that is all the more challenging because of some of the recent trends in software development paradigms. For example, the component based software development method tends to result in use of many black-box components, often adopted from a third-party. Any change in the third-party components may interfere with the rest of the software system, yet it is hard to perform regression testing because the internals of the third-party components are not known to their users. A number of different approaches have been studied to aid the regression testing process. The three major branches include test suite minimization, test case selection and test case prioritization. Test suite minimization is a process that seeks to identify and then eliminate the obsolete or redundant test cases from the test suite. Test case selection deals with the problem of selecting a subset of test cases that will be used to test the changed parts of the software. Finally, test case prioritization concerns the identification of the ‘ideal’ ordering of test cases that maximizes desirable properties, such as early fault detection. Existing empirical studies show that the application of these techniques can be cost-effective.

REVIEW OF LITERATURE:

Software test case generation techniques:

The total cost, time and effort required for overall testing depends on total number of test cases. A test case is a set of inputs given to the software or application to get the pre-specified output. The effort basically depends on the size of the application and the number of test cases.R.P. Mohapatra and Jitendra Singh describe the step by step method for test case generation technique.

1. The first step is to find all possible constraints from start to finish nodes. A constraint is a pair of algebraic expressions which dictate conditions of variables between start and finish nodes.

2. To reduce the test cases, the highest value is assigned to the variable having maximum value and the lowest value is assigned to minimum value within its specified range.

3. After this, the constant value is assigned to the given variable at each node in the defined path.

4. Finally table is created that includes all possible test cases.

Leung and White categorize test cases into five classes as reusable, retest able, obsolete, structural, new specification and new structural test cases.

Reusable test cases only execute the parts of the program that remain unchanged between two versions. Re-testable test cases execute the parts of a program that have been changed in another program. Obsolete Test Cases can be rendered obsolete because their input/output relation is no longer correct, due to changes in specifications and they don't test what they were designed to test due to modifications in the program. There are some methods of test case generation that depends on the application like test case generation for web application, object oriented application, structured based systems, UML applications, applications based on evolutionary and genetic algorithms and many others. Software is tested by using some set of inputs and its success depends on the expected outputs derived from the test case conditions. Throughout the years, several different testing have been proposed for generating test cases.

TEST SUITE MINIMIZATION:

Test suite minimization techniques aim to identify redundant test cases and to remove them from the test suite in order to reduce the size of the test suite. The minimization problem described by Definition 1 can be considered as the minimal hitting set problem. Note that the minimal hitting set formulation of the test suite minimization problem depends on the assumption that each ri can be satisfied by a single test case. In practice, this may not be true. For example, suppose that the test requirement is functional rather than structural and, therefore, requires more than one test case to be satisfied. The minimal hitting set formulation no longer applies. In order to apply the given formulation of the problem, the functional granularity of test cases needs to be adjusted accordingly. The adjustment process may be either that a higher level of abstraction would be required so that each test case requirement can be met with a single test scenario composed of relevant test cases, or that a 'large' functional requirement needs to be divided into smaller sub-requirements that will correspond to individual test cases.

- **Heuristics:**

The NP-completeness of the test suite minimization problem encourages the application of heuristics; previous work on test case minimization can be regarded as the development of different heuristics for the minimal hitting set problem [1–3]. Jeffrey and Gupta extended the HGS heuristic so that certain test cases are selectively retained [4, 5]. This 'selective redundancy' is obtained by introducing a secondary set of testing requirements. When a test case is marked as redundant with respect to the first set of testing requirements, Jeffrey and Gupta considered whether the test case is also redundant with respect to the second set of testing requirements. If it is not, the test case is still selected, resulting in a certain level of redundancy with respect to the first set of testing requirements. The empirical evaluation used branch coverage as the first set of testing requirements and all-uses coverage information obtained by data-flow analysis. The results were compared to two versions of the HGS heuristic, based on branch coverage and def-use coverage. The results showed that, while their technique produced larger test suites, the fault detection capability was better preserved compared to single-criterion versions of the HGS heuristic. Whereas the selective redundancy approach considers the secondary criterion only when a test case is marked as being redundant by the first criterion, Black et al. considered a bi-criteria approach that takes into account both testing criteria [6]. They combined the def-use coverage criterion with the past fault detection history of each test case using a weighted-sum approach and used Integer Linear Programming (ILP) optimization to find subsets. The weighted-sum approach uses weighting factors to combine multiple objectives. For example, given a weighting factor α and two objectives o1 and o2, the new and combined objective, o 0 , is defined as follows:

$$o0 = \alpha o1 + (1 - \alpha)o2$$

Consideration of a secondary objective using the weighted-sum approach has been used in other minimization approaches [7] and prioritization approaches [8].

COMPLEXITY TO ANALYZE THE CODE:

The Software complexity is based on well-known software metrics, this would be likely to reduce the time spent and cost estimation in the testing phase of the software development life cycle (SDLC), which can only be used after program coding is done. Improving quality of software is a quantitative measure of the quality of source code. This can be achieved through definition of metrics, values for which can be calculated by analyzing source code or program is coded. A number of software metrics widely used in the software industry are still not well understood [9]. Although some software complexity measures were proposed over thirty years ago and some others proposed later. Sometimes software growth is usually considered in terms of complexity of source code. Various metrics are used, which unable to compare approaches and results. In addition, it is not possible or equally easy to evaluate for a given source code [10]. Software complexity, deals with how difficult a program is to comprehend and work with [11]. Software maintainability [12-13], is the degree to which characteristics that hamper software maintenance are present and determined by software complexity. There dependencies are shown in Fig. 1.

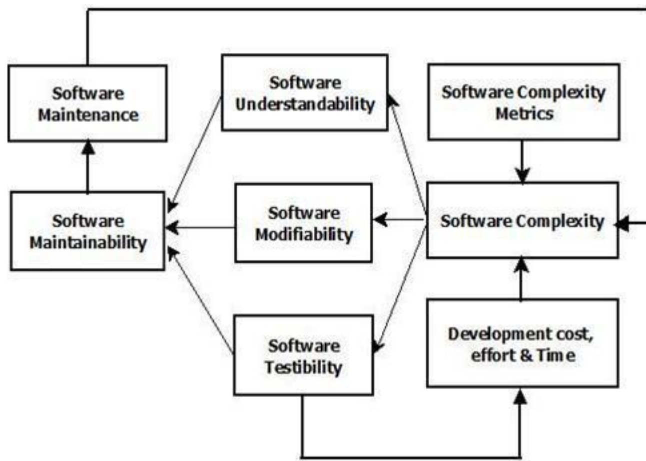

Fig. 1: Relationship between software complexity metrics and software systems

CONCLUSION:

From software engineering point of view software development experience shows, that it is difficult to set measurable targets when developing software products. Produced/developed software has to be testable, reliable and maintainable. On the other side, "You cannot control what you cannot measure" [79]. In software engineering field during software process, developers do not know if what they are developing is correct and guidance are needed to help them accustom more improvement. Software metrics are facilitating to track software enhancement.

REFERENCES:

1. Chen TY, Lau MF. Dividing strategies for the optimization of a test suite. Information Processing Letters 1996; 60(3):135–141.

2. Harrold MJ, Gupta R, Soffa ML. A methodology for controlling the size of a test suite. ACM Transactions on Software Engineering and Methodology 1993; 2(3):270–285.

3. Horgan J, London S. ATAC: A data flow coverage testing tool for c. Proceedings of the Symposium on Assessment of Quality Software Development Tools, IEEE Computer Society Press, 1992; 2–10.

4. Offutt J, Pan J, Voas J. Procedures for reducing the size of coverage-based test sets. Proceedings of the 12th International Conference on Testing Computer Software, ACM Press, 1995; 111–123.

5. Jeffrey D, Gupta N. Test suite reduction with selective redundancy. Proceedings of the 21st IEEE International Conference on Software Maintenance 2005 (ICSM'05), IEEE Computer Society Press, 2005; 549–558.

6. Jeffrey D, Gupta N. Improving fault detection capability by selectively retaining test cases during test suite reduction. IEEE Transactions on Software Engineering 2007; 33(2):108–123.

7. Black J, Melachrinoudis E, Kaeli D. Bi-criteria models for all-uses test suite reduction. Proceedings of the 26th International Conference on Software Engineering (ICSE 2004), ACM Press, 2004; 106–115.

8. Hsu HY, Orso A. MINTS: A general framework and tool for supporting test-suite minimization. Proceedings of the 31st International Conference on Software Engineering (ICSE 2009), IEEE Computer Society, 2009; 419–429.

9. Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. Time aware test suite prioritization. Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2006), ACM Press, 2006; 1–12.

10. T. J. M. Cabe, "A complexity measure," IEEET Ransactions on Software Engineering, vol. 2, 1976

11. I. Herraiz, J. M. G. Barahona, and G. Robles, "Towards a theoretical model for software growth," in 29th International Conference on Software Engineering Workshops (ICSEW'07).

12. W. Harrison, K. Magel, R. Kluczny, and A. Dekok, Applying Software Complexity Metrics to Program Maintenance Compute, vol. 15, pp. 65-79, 1982.

13. T. D. Marco, "Controlling software projects," Prrntice Hall, New York, 1982.