International Journal of Electronic and Electrical Engineering.
ISSN 0974-2174, Volume 7, Number 5 (2014), pp. 463-474
© International Research Publication House
http://www.irphouse.com

A Review of Software Testing Techniques

Manpreet Kaur[1] and Rupinder Singh[2]

[1]*Department of CSE CGC, Gharuan Mohali, India*
[2]*CSE CGC, Gharuan Mohali, India*

Abstract

Software testing is important to reduce errors, maintenance and overall software costs. One of the major problems in software testing area is how to get a suitable set of test cases to test a software system. We identify a number of concepts that every software engineering student and faculty should have learned. There are now many testing techniques available for generating test cases. This set should ensure maximum effectiveness with the least possible number of test cases. The main goal of this paper is to analysed and compare the testing technique to find out the best one to find out the error from the software.

Keywords: Software testing; process model; testing techniques;

1. Introduction

Software testing is a process or a series of processes designed to verify computer code does what it was designed to do. According to *ANSI/IEEE* 1059 standard [1, 2], Testing can be defined as —A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item. Another more appropriate definition is this: [3] Testing is the process of executing a program with the intent of finding errors. The concept of testing is as old as coding and is change along with time. Gelperin and Hetzel [4] proposed the concept of the testing process model based on associated publishing event.

The Debugging Process Model (Before 1956): During that period, the concept of program checkout, debugging and testing were not clearly distinguishable. They used testing and debugging interchangeably. At that time, Alan Turing wrote two articles and addresses some questions and also defined an operational test for intelligent behaviour.

The Demonstration Process Model (1957-78): During that period, the debugging and testing were clearly distinguishable by including efforts to detect, locate, identify and correct fault. Charles baker emphasis on program checkout with two goals: make sure the program runs, and program solves the problem.

The Destruction Process Model (1979-82): Myers wrote the book 'The Art of Software Testing', discussed software analysis and review testing technique. The software testing was first time described as "the process of executing a program with the intent of finding errors".

The Evaluation Process Model (1983-87): The Institute for Computer Sciences and Technology of the National Bureau of Standards published Guideline, specifically targeted at federal information processing system(FIPS) for Validation, Verification, and Testing of Computer Software in 1983, in which a methodology that combine analysis, review, and test activities to provide product evaluation during the software lifecycle was described. The guidelines assured that a carefully chosen set of VV&T techniques can help to ensure the development and maintenance of quality software.

The Prevention Process Model (Since 1988): Beizer wrote the book 'Software Testing Techniques' which have most complete catalog of testing techniques, and defined that "the act of designing tests is one of the most effective bug preventers known."

This period was distinguished from the evaluation-oriented by the mechanism, that the prevention model emphasis on test planning, analysis, and design activities playing a major role, whereas the evaluation model mainly emphasis on analysis and reviewing techniques other than testing. At that time, Hetzel defined the comprehensive methodology called the "systematic test and evaluation process(STEP)" and testing was decomposed of three phases such as planning, acquisition and measurement.

Table 1: Various Testing Process Model.

Sr. No.	**Model name**	**Period**	**Function**
1.	The Debugging Process Model	Before 1956	Testing and debugging was used inter-changeably.
2.	The Demonstration Process Model	1957-78	Testing to make sure that the software satisfies its specification.
3.	The Destruction Process Model	1979-82	Testing to detect implementation faults.
4.	The Evaluation Process Model	1983-87	Testing to detect faults in requirements, design and implementation.
5.	The Prevention Oriented Period	Since 1988	Testing to prevent faults in requirements, design, and implementation.

2. Literature Survey

Gelperin and Hetzel [4] presented the evolution of software test engineering which traced by examining changes in the testing process model and the level of professionalism over the years. Two phase models such as the demonstration and destruction models and two life cycle models such as the evolution and prevention models are given to describe the growth of software testing. They also explain the prevention oriented testing methodology according to the models.

Richardson and Malley[5] proposed one of the earliest approaches focusing on utilizing specifications in selecting test cases. They proposed approaches to specification-based testing by extending a wide range of implementation-based testing techniques to be applicable to formal specification languages and determine these approaches for the Anna and Larch specification languages.

Rapps et al. [6] presented a family of program test data selection criteria derived from data flow analysis technique. The authors argued that currently used path selection criteria that examine only the control flow of a program, are inadequate. Definition/use graph is introduced and compared with a program graph based on the same program. And discuss the interrelationships between these data flow criteria.

Harrold et al. [7] presented a new approach to class testing that supports data flow testing for data flow interaction in a class. They also describe class testing and the application of dataflow testing to class.

Claessen et al. [8] developed a lightweight and easy to use tool named “quickCheck”, that is a combination of two old techniques (specifications as oracle and random testing) works extremely well for Haskell program. They present a number of case studies, in that the tool was successfully used and also point out some pitfalls to avoid.

Vilkomir et al. [9] presented a new concept of tolerance of a testing criterion that describes the ability of every test set, satisfying this criterion, to give a similar level of effectiveness. And evaluation shows the low level of tolerance for such criteria as DC, CC, D/CC, and FPC and, in contrast, the high level of tolerance for MC/DC and RC/DC.

Ntafos [10] presented the comparisons of random testing, partition testing and proportional partition testing. The author guaranteeing that partition testing has at least as high a probability of detecting a failure comes at the expense of decreasing its relative advantage over random testing.

Madeyski Lech et al.[11] presented the concept of using a set of second order mutants by applying them to large open source software with number of different algorithms. They shows that second order mutation techniques can significantly improve the efficiency of mutation testing at a cost in the testing strength.

Jia et al. [12] presented a detailed survey and analysis of

trends and results of Mutation Testing. These findings provide evidence to support the claim that the field of Mutation Testing is now reaching a mature state.

Graves Todd et al. [13] proposed regression test selection techniques to reduce some expense. They performed an experiment to examine the relative costs and benefits of several regression test selection techniques. The experiment considered five techniques for reusing test cases that focusing on their relative abilities.

Juristo et al. [14] analyzed the maturity level of the knowledge about testing techniques. For this, they examined existing empirical studies about testing techniques. According to knowledge, they classified the testing techniques and choose parameters to compare them.

J. A. Whittaker[15] presented a four phase approach to determine how bugs escape from testing. They offer testers to a group related problems that they can solve during each phase.

Duran and ntafos[16] presented the comparing random testing to partition testing, in which, using hypothetical input domains, partitions, and distribution of errors, and then compared the probabilities that each technique would detect an error. their conclusion was that, despite random testing did not always compare fairly to path testing, but a cost effective alternative.

Hamlet and taylor [17] presented more extensive simulations, and reach at more precise results about the relationship between partition probability, failure rate, and effectiveness.

Li et al. [18] Presented a comparison of four unit level software testing criteria such as mutation testing, prime path coverage (PP), edge pair coverage (EP) and all-uses testing (AU), which were compared on two bases, first, the number of seeded faults found and second, the number of tests needed to satisfy the criteria. They concluded the result as mutation was the most efficient criterion.

3. Classification of Testing Techniques

This paper focuses on all currently known existing testing technique.

1) Random Testing: The adjective "random" is slang which means unexpected, unpredictable. In this sense, Random testing of a program specifies testing badly or rapidly done, which is opposite of systematic testing like functional testing or structural testing. This slang meaning, that might be presented "haphazard testing" in normal language. Random testing technique is composed of the oldest and most intuitive technique which is most used and least useful method. In opposition, the technical, mathematical meaning of "random testing" indicates to an explicit lack of "system" in the choice of test data. Hence, there is no correlation among different tests.[19]

Random testing is a black-box technique. Therefore, this is useful, when information of internal structure of the software is not required. This technique is most effective when the output from the result of each test can be automatically checked. The random values could be derived manually as well as by a pseudo-random number generator.

This family of techniques proposes randomly generating test cases without any predefined guidelines. But, pure randomness infrequently occurs in reality and the other two alternatives of the family are the most commonly used.

a) Pure random: Test cases are randomly generated until appear to be enough.

b) Guided by number of cases: Test cases are randomly generated until a given number of cases has been reached.

c) Error guessing: Test cases are generated by the subject’s knowledge of what typical errors occur during programming. It stops until they all appear to have been covered. [14]

2) Functional Testing: The software under test is viewed as a “black box”. Here, the selection of test cases is based on the requirement or design specification of the software under test. Functional testing emphasizes on the external behaviour of the software entity under test. The technique of this family is given below [20].

a) Equivalence Partitioning: This technique divides the input domain of a program into different equivalence classes. This method is used to reduce the total number of test cases to a finite set of test cases.

Equivalence classes – set of valid or invalid states for input conditions and can be defined in the following way:

- An input condition specifies a range of one valid and two invalid equivalence classes are defined;
- An input condition needs a specific value that is one valid and two invalid equivalence classes are defined;
- An input condition specifies a member of a set that is one valid and one invalid equivalence class are defined;
- An input condition is Boolean that is one valid and one invalid equivalence class are defined. By using this technique, one can get test cases that identify the classes of errors.[21]

b) Boundary Value Analysis: This technique is similar to the Equivalence Partitioning technique, except that for creating the test cases beside input domain use output domain. Because in many applications, errors occur at the boundaries of input domain. This technique is used to identify errors at boundaries instead of finding at center of input domain.

The test cases can be formed in the following way:

- An input condition specifies a range bounded by values a and b, test cases should be made with values just above and just below with respect to a and b;
- An input condition specifies various values, test cases should be generate to exercise the minimum and maximum numbers;
- Rules 1 and 2 apply to output conditions; if internal program data structures have specify boundaries, produce test cases to exercising that data structure at its boundary.[21]

3) Control Flow Testing: Control-flow testing assures that program statements and decisions are executed by code execution and useful for white-box testing. But, they could be also successfully exercised in black-box testing as specification-based criteria. The control flow graph which is mainly used for static analysis of the program, is also used for defining and analyzing control-flow test criteria. It check these Boolean decisions of the program based on the specification.

The basic control-flow criterion is Statement Coverage (SC) which requires that test-data must execute every statement of the program at least once. Because this requirement is not directly connected with the main parts of the criteria. We use definitions of the Random Coverage (RC), Decision Coverage (DC), Condition Coverage (CC), Decision/Condition Coverage (D/CC), etc.:

a) Decision Coverage (DC): Every possible branch must be executed at least once. Specially, the Boolean decisions in the program must take both **T** and **F** values. The DC criterion deals a decision as a single node in the program structure, in spite of everything, the complexity of the Boolean expression and dependencies between the conditions creating the decision. For critical real-time applications in which half of the executable statements may involve Boolean expressions, the complexity of the expression is of concern.

b) Condition Coverage (CC): Every condition in each decision has taken both **T** and **F** values at least once. Whereas CC deal with the individual condition that makes a decision, it does not define any rule that checks the outcome of decision. The following two criteria improve on DC and CC by taking into account values of both the decision and conditions that occur in the decision.

c) D/CC criterion: Every decision in the program has taken all possible outcomes at least once and every condition in each decision has taken all possible outcomes at least once.The requirements of these criteria are uncertain and often not sufficient for safety-critical software testing. So we can use more complicated and stronger criteria such as FPC, MC/DC, and RC/DC.

Full Predicate Coverage criterion (FPC): **Each** condition in a decision has taken all possible outcomes where the value of a decision is directly correlated with the value of a condition. This means that the test-set must include test-cases like the value of the decision is different when the condition changes. The difference between D/CC and FPC is that, for D/CC, a test set could contain only two test cases with different outcomes of a decision and test cases with different values for each condition could be chosen regardless of the values of a decision[9]. For FPC, test cases chosen for testing a condition, should provide different outcomes for the decision at the same time.

Modified Condition/Decision Coverage criterion (MC/DC)*:* Every point of entry and exit in the program has been invoked at least once, each condition in a decision in the program has taken on all possible outcomes at least once, and every condition in a decision has been shown to independently affect the decision's outcome. A condition is independently affect a decision's outcome by varying just that condition during holding all other possible conditions [9].

This criterion takes into account only that situation where an independent change in the condition causes change in the value of a decision. The shortcoming of this approach is that it does not check for the situation where a change in condition should keep the value of decision.

Reinforced Condition/Decision Coverage criterion (RC/DC): To improve the shortcoming of MC/DC, a new criterion called Reinforced Condition/Decision Coverage (RC/DC) has been proposed. Moreover MC/DC, RC/DC requires all the decisions to independently keep the value of decision remain the same at both **T** and **F**. Each condition in a decision has been independently affect the decision's result, and each condition in a decision has been independently keep the decision's result. A condition is independently affect and keep a decision's outcome by varying just that condition while holding fixed all other conditions [9].

4) Data Flow Testing: Data flow testing is a kind of structural testing: As in functional testing, the program can be tested without any knowledge of its internal structures, but the structural testing techniques require the details of the program's structure. Data flow testing emphasis on the variables that are defined and used at different points within the program; in data flow testing, they use program graph to chart the changing values of variables within the program.

There are two major types of data flow testing: the first, called define/use testing, the second uses “program slices”.

Define/Use testing: Define/use testing was first introduced by Sandra Rapps and Elaine Weyuker in the early 1980s. The term “Define/Use” refers to the two main aspects of a variable: it is either defined (a value is assigned to it) or used (the value assigned to the variable is used in a different place). Define/Use testing uses paths of the program graph to generate test cases. Define/use testing is intended to use with structured programs.

Defining nodes, specified as DEF(v, n): A node n in the program graph P is a defining node for variable v – DEF(v, n) – if the value of v is defined at the statement chunk in that node.

Usage nodes, specified as USE(v, n): A node in the program graph P is a usage node for variable v – USE(v, n) – if the value of v is used at the statement chunk in that node.

Usage nodes can be split into a number of types according to the variable is used. The two major types of usage node are:

- ***P-use:*** **predicate use** – the variable is used when making a decision. (e.g., if c >16).
- ***C-use:*** **computation use** – the variable is used in a computation for calculating values.(e.g. c= 2 + a – with respect to the variable a).
- There are also three other types of usage node, which are as followed:
- ***O-use:*** output use – the value of the variable is output to the external environment.
- ***L-use****:* location use – the value of the variable is used, to determine which position of an array is used.
- ***I-use:*** iteration use – the value of the variable is used to control the number of iterations made by a loop.

Definitions of these metrics are as followed:

All-definitions: In All definitions, test set involves each definition of each variable be covered by atleast one use of the variable, that use either c- use or p- use.

All c-uses/some p-uses: In this, test set involves atleast one path from each definition of each variable to every c- use of that definition, If there are variable definitions that are not covered, then use p-uses.

All p-uses/some c-uses*:* In this, test set involves atleast one path from each definition of each variable to every p- use of that definition, If there are variable definitions that are not covered, then use c-uses.

All-c-uses: This is derived from All-c-uses/some-p-uses by removing the requirement of p- use. In this, there would be atleast one path between each definition of each variable and each c-use of the variable. if there are variable definitions that are not covered, then leave them.

All-p-uses*:* This is derived from All-p-uses/some-c-uses by removing the requirement of c- use. In this, there would be atleast one path between each definition of each variable and each p-use of the variable. if there are variable definitions that are not covered, then leave them.

All-uses: This test set involves at least one path from each definition of each variable to each use of the definition.

All definition-uses-paths: Each definition of each variable includes the entire possible path to each use of the definition. It is the strongest data-flow testing strategy. Furthermore, this requires greatest number of paths for testing.

All definition-uses: Each definition of each variable includes all the possible executable path to each use of the definition.

Another approach is to also look at the program according to its variables; but, this time to divide the program into a number of independently executable components, each emphasis on one particular variable at one particular location within the program. Then we can examine the program with respect to those variables without having to examine the entire program.

5) Mutation testing:

Mutation Testing is a fault-based testing technique which is used for testing software at the unit level, integration level and specification level. Mutation Testing provides a testing criterion called the mutation adequacy score. This score can be used to measure the ability to detect faults. This process is called mutation analysis. [12]

In mutation testing, the faults represent the mistakes made by a programmer so they are deliberately introduced set of faulty programs called mutants. Each mutant program is developed by applying a mutant operator to a the original program. Many mutation operators must produce equivalent mutants. The resulting program is equivalent to the original one and obtain the same output as the original one[22].

The problem with the techniques is scalability. A mutation operator can generate several mutants per line of code. Therefore, there will be number of a mutants for long programs. Different types of mutation testing as:

a) Strong mutation: mutation testing is most effective testing methods proposed to detecting the faults. But the problem is performance. The number of mutants generated can be prohibitively large even for the programs that are less than 100 line of code. so, mutation testing is too expensive to use in practice.

b) Weak mutation: To reduce the cost of mutation testing is to examine a small percentage (say x%) of randomly selected mutants of each mutant type and ignore the remaining mutant.

This gives rise to weak mutation variants, depending on the percentage covered, for example, randomly selected 10% mutation, ex-weak, st-weak, bb-weak/1, or bb-weak/n.[14]

c) Selective Mutation: Another approach to reduce the cost of mutation testing is selective mutation. Using selective mutant one examines only a few specified types of mutants and ignores the others. This gives rise to selective mutation variants depending on the selected operators, like, for example, two, four or six selective mutation (depending on the number of mutation operators not taken into account) or abs/ror mutation, which only uses these two operators.(14)

6) Regression testing: Regression testing is applied when changes are made to existing software. This is performed between two different versions of software to provide confidence that the newly introduced changes do not interrupt the behaviour of the existing, unchanged part of the software.[23] regression testing technique would utilize the information of program fault as it reordered and reduced the test cases. As it is normally difficult to collect information about the existence of faults within the program under test, regression testing methods can use a proxy for this complete information. The aim is to control both the size and execution time of a test suite by eliminating the tests that redundantly cover the test requirements. [24]

The specific techniques of regression testing as below:

a) Minimization Techniques: Minimization based regression test selection techniques are used to select minimal sets of test cases from test suite that yield coverage of modified or affected portions of program.

b) Dataflow Techniques: Dataflow regression test selection techniques is used to select test cases that exercise data interactions that have been affected by modifications.

(c) Safe Techniques: Most regression test selection techniques such as minimization and dataflow are not designed to be safe. And these techniques are failed to select a test case that would have informed a fault in the modified program. On the other hand, when certain set of safety conditions can be satisfied, safe regression test selection techniques guarantee to reveal faults in program.

(d) Ad Hoc/Random Techniques: Developers often select test cases based on “hunches, when retest-all approach cannot be used and no test selection tool is available. When another simple approach is to randomly select a predetermined number of test cases from test suite.

(e) Retest-All Technique: The retest-all technique simply reuses all existing test cases. To test program, the technique effectively selects all test cases in test suite [13].

Table 2: Comparison of testing techniques.

Sr. No.	**Type**	**Testing Environment**	**Effective (fault detection)**	**Size of test pool**	**Testing technique**
1	Random Testing	Black	Least effective	Large	Specification based
2	Functional Testing	Black	Effective	Large	Specification based
3	Control Flow Testing	White	Effective	Medium	Code based
4	Data Flow Testing	White	Effective	Small	Code based
5	Mutation testing	White	Most effective	Small	Fault based
6	Regression testing	White/Black	Most effective	Based on program size	validation

4. Conclusion

To conclude our survey, we return to the problem that how to get a suitable set of test cases to test a software system and find out errors. But it is really not possible to find out all the errors in the program. Thus, the major question arises, which strategy we would adopt to test. For this purpose, we have taken and analyzed number of testing techniques. Finally, the results of the analysis have been presented. The major conclusions are that, our current testing technique knowledge is very limited and is based on impressions and perceptions

References

[1] A. P. Mathur, “Foundation of Software Testing”, Pearson/Addison Wesley, 2008.

[2] IEEE Standard 829-1998, “IEEE Standard for Software Test Documentation”, pp.1-52, IEEE Computer Society, 1998.

[3] Glenford J. Myers, “The Art of Software Testing, Second Edition” Published by John Wiley & Sons, Inc., Hoboken, New Jersey.

[4] D. Gelperin and B. Hetzel, “The Growth of Software Testing”, *Communications of the ACM*, Volume 31 Issue 6, June 1988, pp. 687-695[history of st]

[5] D. Richardson, O. O’Malley and C. Tittle, “Approaches to specification-based testing”, *ACM SIGSOFT Software Engineering Notes*, Volume 14 , Issue 9, 1989, pp. 86 – 96[Approaches to specification-based testing]

[6] S. Rapps and E. J. Weyuker, “Selecting Software Test Data Using Data Flow Information,” *IEEE Transactions on Software Engineering*¸ April 1985, pp. 367-375

[7] Harrold Mary Jean, and Gregg Rothermel. "Performing data flow testing on classes." *ACM SIGSOFT Software Engineering Notes*. Vol. 19. No. 5. ACM, 1994.[acm.pdf]

[8] Claessen Koen, and John Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs." *Acm sigplan notices* 46.4 (2011): 53-64.

[9] Vilkomir Sergiy A., Kalpesh Kapoor, and Jonathan P. Bowen. "Tolerance of control-flow testing criteria." *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*. IEEE, 2003.[control ieee]

[10] Ntafos Simeon C. "On comparisons of random, partition, and proportional partition testing." *Software Engineering, IEEE Transactions on* 27.10 (2001): 949-960.[comparison random]

[11] Madeyski Lech et al. "Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation." (2013): 1-1.[LR4]

[12] Jia Yue, and Mark Harman. "An analysis and survey of the development of mutation testing." *Software Engineering, IEEE Transactions on* 37.5 (2011): 649-678.

[13] Graves Todd L. et al. "An empirical study of regression test selection techniques." *Proceedings of the 20th international conference on Software engineering*. IEEE Computer Society, 1998.

[14] Juristo Natalia, Ana M. Moreno, and Sira Vegas. "Reviewing 25 years of testing technique experiments." *Empirical Software Engineering* 9.1-2 (2004): 7-44.

[15] J. A. Whittaker, “What is Software Testing? And Why Is It So Hard?” *IEEE Software*, January 2000, pp. 70-79[hard software testing]

[16] Duran, Joe W., and Simeon C. Ntafos. "An evaluation of random testing."*Software Engineering, IEEE Transactions on* 4 (1984): 438-444.

[17] Hamlet Dick, and Ross Taylor. "Partition testing does not inspire confidence (program testing)." *IEEE Transactions on Software Engineering* 16.12 (1990): 1402-1411.

[18] Li Nan, Upsorn Praphamontripong, and Jeff Offutt. "An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage." *Software Testing, Verification and Validation Workshops, 2009. ICSTW'09. International Conference on*. IEEE, 2009.[testcritcomp.pdf]

[19] Hamlet, Richard. "Random testing." *Encyclopedia of software Engineering*(1994).

[20] Luo, L. "Software Testing Techniques: Technology Maturation and Research Strategy." *Class Report for* (2001).

[21] Jovanavic, Irena. "Software testing methods and techniques." *IM Jovanovic is with the Inzenjering, Mat* 26 (2008).

[22] Bluemke, Ilona, and Karol Kulesza. "A Comparison of Dataflow and Mutation Testing of Java Methods." *Dependable Computer Systems*. Springer Berlin Heidelberg, 2011. 17-30.

[23] Yoo, Shin, and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." *Software Testing, Verification and Reliability* 22.2 (2012): 67-120.

[24] Biswas, Swarnendu, et al. "Regression Test Selection Techniques: A Survey."*Informatica (03505596)* 35.3 (2011).