Software Testing: A Survey and Tutorial on White and Black-box Testing of C/C++ Programs

Muhammad Nouman, Usman Pervez and Osman Hasan
School of Electrical Engineering and Computer Science
National University of Sciences and Technology
Islamabad, Pakistan
Email:{muhammad.nouman,usman.pervez,osman.hasan}@seecs.nust.edu.pk

Kashif Saghar
Software Quality Assurance Department
Centers of Excellence in Sciences
and Applied Technologies
Islamabad, Pakistan
Email:kashif.saghar@gmail.com

***Abstract*—Over the past couple of decades, we have been witnessing an ever increasing dependency of humans on computers and consequently software. Some of these software are governing the working of very safety-critical domains, like medicine and military. This transition has brought up software testing to ensure that the software behave as intended. With the growing complexity of software, software testing has also revolutionized in the past couple of decades. This paper surveys the recent trends in software testing and provides comprehensive tutorial about black and white-box testing. Moreover, an analysis of different commercial tools in this domain is also presented.**

I. Introduction

Over the past couple of decades, dependance of humans on computers, hence software, has not only increased many folds but nowadays computers almost encompass every aspect of our life, ranging from domestic applications to safety-critical applications. For software applications, which require high quality and maximum reliability, software quality management is an essential task. A slight negligence in this aspect can even result in the loss of human lives. For example, a software bug in the Therac-25 cancer therapy machine was primarily responsible for three deaths and three severe injuries during the mid-80s. Similarly, the computer software that controls the aeronautic navigation of an aeroplane or the one which keeps the pilot updated of the surroundings by communicating from the flight service station requires a rigorous analysis before deployment in the domain. Besides putting life at risk, software failures can also lead to dire financial consequences while dealing with inter-bank transactions or stock-exchange markets. According to a study conducted by NIST in 2002 [27], it has been revealed that software bugs cause a huge loss to U.S economy. The total loss accumulates up to about $595 billion per annum.

Software development is an exercise to solve out real life problems. As with any problem solving technique, surety of solution is a vital part of whole process. *Software testing* and evaluation does this for software, as it is a composed of a series of processes which not only makes sure that a piece of code is doing exactly what it was supposed to do but also that it does not do anything that was unplanned. Software testing can be formally defined in a number of ways and some commonly used definitions are as follows [2]:

"Testing is the activity or process which shows or demonstrates that a program or system performs all intended functions correctly"

"Testing is the activity of establishing the necessary *confidence* that a program or system does what it is supposed to do, based on the set of requirements that the user has specified"

"Testing is the process of executing a program/system with the intent of finding errors"

"Testing is any activity aimed at evaluating an attribute or capability of a program or system and determining that it meets its required results"

It is commonly believed that the earlier a software bug is found the cheaper it is to fix it. It is reported that by using an efficient software testing tool at first place more than one third of the $595 billion per annum cost of software bugs cost could be evaded [27].

Software testing is quite closely tied to software quality, which is primarily composed of several attributes, depicted in an hierarchical fashion with their respective characterization in Figure 1 [1]. For example, one of the fundamental quality attributes of software is being *reliable*, which in turn has characterization of being *adequate* and *robust* at same time, where adequate software has, in turn, is further characterized by being *correct*, *complete* and *consistent*. These quality attributes are much more qualitative instead of being quantitative.

Besides reliability, other software quality attributes, as show in Figure 1, include *usability*, *efficiency*, *transportability*, *testability* and *maintainability*. However in practical scenarios it has been observed that efficiency and other quality attributes often conflict with each other. For example, a highly critical chip demands assembly level programming for higher efficiency but doing so would highly decrease its transportability to other platforms. Because of these reasons, each software development team, must first categorize the relative importance of quality attributes.

In this survey, we have explained software testing, verification and validation (V&V) techniques that are applied throughout the life cycle of software development. One of the main problems associated with verification is confinement of verification techniques to the last stages of development [10]. Such practices can result in heavy financial losses, as the cost of fixing a bug is highly dependant on the time it is caught. The authors in [6] point out the same mistake of delaying the activity of testing and verification process until last stages of software development, which may result in disastrous consequences. Thus, with highest quality and minimum possible

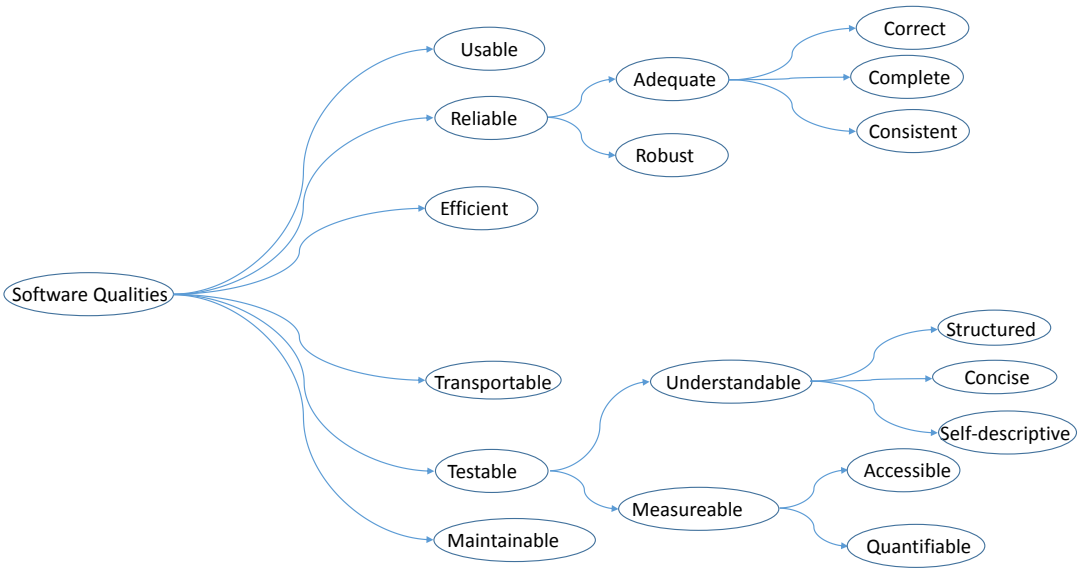

Fig. 1. Software Quality Assurance Attirbutes

cost as final goals, testing and verification must be applied throughout the life cycle of software development.

Software verification and validation mainly consist of processes, which confirm that the software is validating all of its specifications and complying with its intended purpose.

The *software verification* process can be described as a question,

Either we are building the product in the right way?

It checks whether the software and its associated processes are complying with the following rules:

- Does the software satisfy the requirements?
- Does the software conform to the practices and conventions?
- Before moving to next cycle, does it satisfy previous life cycle activity?

Similarly, the *software validation* process can also be described as a question,

Either we are building the product which was demanded?

It checks whether software and its associated processes are complying with the following rules:

- Does the software use the same resources which were allocated at each life cycle?
- Does the software solve the problem which it was supposed to solve?
- Does the software perform as per users expectations?

Various testing approach (e.g., citetesting,testing1,testing3) have been proposed but *White-box* and *Black-box* are considered to be the most widely used ones. The white-box testing allows deep analysis and identify bugs in the internal structure of the program and helps the user to optimize the program to the maximum extent. On the other hand, the black-box testing approach checks the dynamic behaviour of the program and is generally faster and easier to perform than the white-box testing. This paper is primarily surveys these testing approaches for C programs.

The main motivation for choosing C programming for this survey is its wide acceptance [18] due to its many unique features, including portability of the compilers, ready access to the hardware, a user friendly syntax with interactive environment, powerful and varied repertoire of operators, built-in standard libraries. C is also a foundational block for many other user friendly languages like C++, C#, java etc. Initially C was used only for system development work, but later on, due to its versatile benefits, its usage increased rapidly in other development as well, Examples include development of compilers for other languages, assemblers, text editors, language interpreters, network drivers, printing spoolers, data bases, modern applications development etc. Programming in C is found everywhere in the world such as, development of embedded systems, avionic software systems, medical software systems, communication systems, games, custom applications, commercial softwares etc. Due to the large scale applications of C programming, the question of its quality assurance and validation has always been an important one [20], [25].

The rest of the paper is organized as follows: In Section II, we describe all the verification activates that must be performed throughout the life cycle of software development. This is followed by the description of white and black-box testing of C/C++ programs along with their strengths, limitations and related work in Sections III and IV, respectively. In Section V, we have evaluated different White and Black box testers and described different pros and cons associated with these tools. Finally, Section VI concludes the paper.

TABLE I. VERIFICATION ACTIVATES THROUGHOUT SOFTWARE DEVELOPMENT

Software Development Stages	Verification activities
Requirement Analysis	1.Brainstorm and define the verification method. 2.Define the suitability of proposed approach. 3.Create the test data.
Design	1.Find out if design complies with the requirements. 2.Find out if design complies with standards. 3.Create structural test data.
Implementation	1.Determine if implementation is performing consistently with requirement and design. 2.Apply test data.
Operation and Maintenance	1.Re-verify. 2.Adequate re-development as per deviations.

II. VERIFICATION ACTIVITIES IN RESPECTIVE DEVELOPMENT CYCLES

Table I summarizes all the verification activates that must be performed throughout the life cycle of a software development. A successful software development project requires an elaborately defined product after each development cycle. For example, at the conclusion of requirement analysis, a clear definition and specification of what exactly is required, must be presented. A successful product at the end of each cycle ensures that the respective testing and verification activates can be performed, which in turn ensures a complete product at the end as per specifications.

A brief explanation of each development cycle is given below along with the verification steps carried along those cycles:

A. Verification activities in Requirement Analysis Stage

One of the most important steps in the development cycle is requirement analysis, which ensures the completeness, consistency and correctness of requirements. This step is vital for the successful completion of the project. Ideally, the verification actives must be initiated in the requirement analysis stage of software development. It includes the selection and formulation of test cases and test evaluation criteria. If the budget and time allows, a separate and independent test team should be formed. The test team should evaluative the project with proper test schedules after every achieved milestone.

B. Verification activities in Design Stage

The next stage of software development, i.e., design, requires the acquirement or development of support tools and development of test procedures. Such support tools ensure that the proposed design of software complies with the previous stage of requirement. Moreover as with the growth of software development, more and more effective set of test cases should be built or acquired to ensure the successful completion of the project.

C. Verification activities in Construction Stage

One of the important development stage with respect to the verification process is the construction phase. In this phase, real development of proposed designed according to the requirements takes place. There exist many test tools and techniques which ensure the successful verification in this critical stage of development. Most of the classical and manual techniques include the code walk-through and manual inspection. However, with the advancement in software development, and hence the complexity, these techniques have become obsolete now. Therefore, large software require some automated tools to facilitate test teams. White-box testing (Static analysis) techniques require the detection of errors by analyzing the code flow and critically analyzing the code to find the logical errors. Block-box testing (Dynamic analysis) of code is performed by actually running the software and performing coverage tests.

D. Verification activities in Operation and maintenance stage

More than 50% of a software development budget is spent on the operation and maintenance stage. As the software gets deployed more and more modifications deem necessary. The basic reason for such modification is to correct errors or to enhance the capability of software. Therefore, after every modification, testing should be initialized to verify the correct functioning of the program.

III. WHITE-BOX TESTING

White-box testing [24], usually known as static testing is a kind of testing in which the programming instructions, APIs, internal structure, design and implementation of the software is tested. Regardless of the behaviour and working of the software, the software is tested at the instruction level. Every single instruction is tested and bugs are identified. White-box testing includes different kinds of tests, such as API testing, code coverage, mutation test, pointers test, arithmetic/logical operations test, memory leakage test, array out of bound test.

To understand the working of a white-box tester consider the design methodology, depicted in Figure 2, for constructing a simple white-box testing tool. The tool is mainly composed of a GUI or command-line interface, which the user can use to provide the input code that is to be tested. Thereafter, the user is asked to identify the tests to be performed. The white-box testing tools are usually composed of an Automatic Code Analyser (ACA), which performs two major operations on the code, i.e., Abstract Syntax Tree (AST) [22] and Static Analyser [5]. AST basically extracts all the required information from the C program, such as header files, custom functions, main function, loops, inputs, outputs, instructions, operations and all other possible information. All the information is stored separately in different files and thus is further fed into the static analyser. Based on this information, the static analyser tests every single instructions, starting from the `main` function.

As an illustrative example, consider the case of the "*divide by zero*" test. The given C program is entered by the user and is fed into AST of the ACA. AST will first extract out all the information and thus the information will be fed into the static analyser. Static analyser will move down into the main function and search for the divide instructions. For all divide instructions, the static analyser will identify the denominators and find the range of those denominators through the information extracted by the AST. If in any case, the denominators exhibit value '0', the static analyser will generate "Test Failed" as an output. In case of "*array access out of bounds*" test, the static analyser will move down the main function and search for the arrays, loops and the instructions involving assign functions. Furthermore, the static analyser

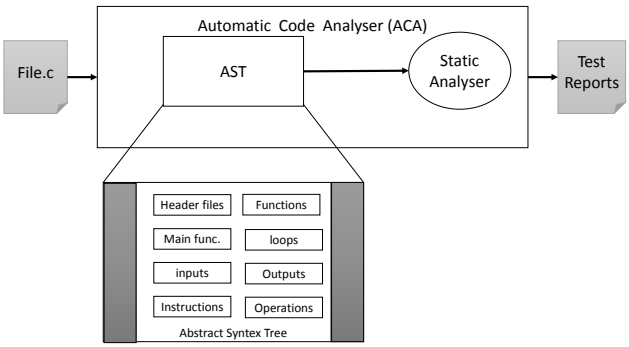

Fig. 2. Design Methodology for a White-box Tester

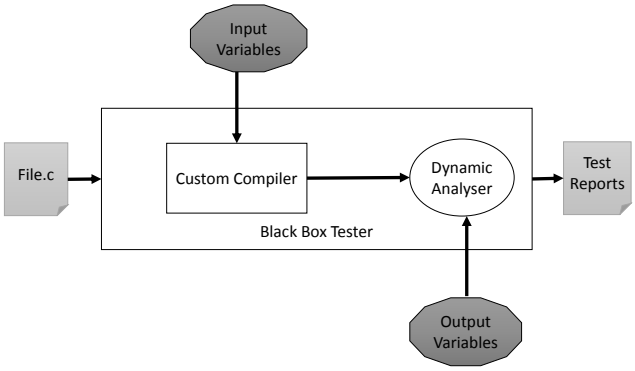

Fig. 3. Design Methodology for a Black-box Tester

will test the loops and assign instructions against the array ranges and thus if overflow occurs, the static analyser will generate "Test Failed" as an output. Similarly, other tests are performed according to the nature of the tests.

A lot of work has been done in the literature for the automatic testing of C programs using the white-box approach [24]. For Application Programming Interface (API) testing, a very brief, step by step method is presented in [16] which is basically the instruction-wise analysis of the C program. In [9] [1] [8], assertion testing is performed by declaring assert functions within the program. Code coverage testing is presented in [31] [14] [15] and in [11] [13], fuzz testing is performed by applying invalid inputs to the program and analysing that either the program crashes or not. In [26], regressing testing is performed by introducing recurring testing in the program.

IV. Black-box Testing

Black-box testing [4], usually known as dynamic testing, is a kind of testing in which the behaviour and working of the software is tested by checking the inputs and outputs of the software. In black-box testing, the internal structure/implementation of the software is of no interest. The inputs of the software are applied and corresponding outputs are obtained, and thus on the basis of those inputs and outputs, bugs are identified. Black-box testing includes equivalence partitioning, boundary value analysis, all-pair testing, fuzz testing, model base testing.

To understand the working of a black-box tester, consider the design methodology, depicted in Figure 3, for constructing a simple black-box testing tool. The tool is mainly composed of a GUI or command-line interface, which the user can use to provide the input code that is to be tested. Thereafter, the user is asked to identify the tests to be performed. The black-box testing tools are usually composed of a custom C language compiler that is able to efficiently read, understand, compile and execute all C programs. The C program entered by the user is fed into this compiler and after that, the user is asked for the input variables. The compiler then compiles the program and focuses on the variables-operations of those input variables. The user is also asked for the output variables and thus, this information together with the information obtained from the compiler, is given to a *dynamic analyser* [29]. Based on the input and output values, provided by the compiler, the dynamic analyser performs the identified black-box tests and generates the tests report as output.provides the black-box testing results.

As an illustrative example, consider the case of boundary value analysis. in this case, the C program to be tested is fed into the compiler along with the input variables. The compiler generates the minimum and maximum values of those input variables and executes the program to compute the minimum and maximum values of the output variables. The user also provides the output variables of interest and thus all the information is fed in the dynamic analyser. The dynamic analyser tests the inputs values against their corresponding output values and thus analyzes the faults in the program, such as run-time error and software crash, where these errors occur in case when the program is unable to handle the minimum and maximum values of the inputs or outputs. For all pair testing, the compiler generates all the possible values of the input and output variables and thus compiles the C program in the same way as described above for boundary value test. Other black-box tests can be performed in the same way and test reports can be generated as outputs.

A lot of work has been done in the literature for the automatic testing of C program using the black-box approach [4]. A boundary value analysis is presented in [12] [19] [28] for software by applying boundary values as inputs and analysing that either the software crashes or not. Equivalence partitioning is briefly discussed in [23] [17] [7]. The model based testing of a software through its state machine is described in [3] [30] [21].

V. Analysis and Evaluation

In this section, we have evaluated different white and black box testers. The evaluation is based on the different tests associated with each tester.

In white box testing, we are usually interested in Application Programming Interfaces (API) testing, code coverage and static testing. Table II presents a comparison between different commercially available white-box testing tools based on these features.API makes the development quite easy. Just like any other interface to the product, APIs must be thoroughly tested and evaluated before their launching. As described earlier, APIs are the basic building blocks of the many software applications, so even a minor bug in these

TABLE II. SOME COMMERCIALLY AVAILABLE WHITE BOX AUTOMATIC TESTING TOOLS

Sr. #	Tool Name	API testing	Code Coverage	Static testing
1	Vector Cast		✓	✓
2	Cantata			✓
3	Cpp Check			✓
4	Embunit			✓
5	Parasoft	✓	✓	✓
6	Reactis		✓	
7	Testwell C		✓	
8	C Coverage Analyser		✓	
9	Testwell Cmt++		✓	✓
10	Ldra Testbed			✓
11	Aq Time		✓	
12	Bullseye Coverage		✓	
13	Tessy		✓	✓
14	Testwell CTA++			✓

APIs can cause the whole application to crash and debugging, while they are already integrated, is quite a tedious task. Moreover an improper API can cause security loopholes in the software application, allowing access to unauthentic users. Code coverage under a specific test, keeps record of the parts of code which got executed. The higher the code coverage, the better it is as it shows that maximum code got executed during a particular test. Static testing generally includes reviews, walk-throughs and inspections. In static testing, tools generally check the structure of the source code. Moreover it also check the syntax and data flow.

In black box testing, we are usually interested in all-pair testing, model based testing, runtime error testing and load testing. Table III presents a comparison between different commercially available black-box testing tools.

All pair or pair wise testing is a testing approach in which all possible combinations of inputs are tested through the software. For example, we have a simple function taking 5 input arguments, then in this kind of testing all possible combinations of 5 arguments are passed to the function and the tester tries to find those combinations of values against which the software crashes. In model based testing, test cases are derived from the model of system under test. These tests are then executed against the system under test. A runtime error can occur during the execution of program and causes the application to crash. In load testing, software under test is put under the maximum load and its response is measured accordingly. It helps in identifying the maximum capacity as well as the bottleneck of the system.

VI. CONCLUSION

This paper provides a survey and tutorial of the widely used techniques used for testing C programs during the life cycle of software development. Traditionally, the code is manually inspected and examined to catch potential errors. These traditional ways, including walk-throughs, reviews and manual inspection, can be employed to all software development steps. But these methods require serious commitment and become infeasible while dealing with large and complex codes. The automated white and black-box testing approaches have the potential to overcome the limitations of the traditional approaches. White-box testing, also known as static testing, is a kind of testing in which the programming instructions, APIs, internal structure, design and implementation of the software is tested. The paper provides a summary of the existing white-box tester along with a comparison between their supported features. Parasoft seems to be the best tool, providing all the tests related to white-box testing. Black-box testing, also known as dynamic testing, is a kind of testing in which the behaviour and working of the software is tested by checking the inputs and outputs of the software and the internal structure of software is of no interest. We also provided a comparison of different black-box testing tools, along with the different tests supported by each tool. Reactis seems to be the best tool, providing the maximum tests related to black-box testing. Black-box testing is simple, easy to perform, and fast but it cannot identify bugs in the internal structure. White-box testing, on the other hand, is capable of identifying bugs in the internal structure but it is time consuming and cannot test the behaviour of the software. Therefore, for thorough and complete testing of the software, white box testing must be performed in conjunction with black box testing.

REFERENCES

[1] W Richards Adrion, Martha A Branstad, and John C Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys (CSUR)*, 14(2):159–192, 1982.

[2] K. Akingbehin. Towards destructive software testing. In *Computer and Information Science, 2006 and 2006 1st IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse. ICIS-COMSAR 2006. 5th IEEE/ACIS International Conference on*, pages 374–377, July 2006.

[3] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee White. State generation and automated class testing. *Software Testing Verification and Reliability*, 10(3):149–170, 2000.

[4] Boris Beizer. *Black-box testing: techniques for functional testing of software and systems*. John Wiley & Sons, Inc., 1995.

[5] Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérome Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *ACM SIGPLAN Notices*, volume 38, pages 196–207. ACM, 2003.

[6] B. W. Boehm. *Seven basic prmci ples of software engmeering," in Soft ware engineering techniques, Infotech State of the Art Report, Infotech, Lon don*. 1977.

[7] Shu-lai CHU and Wei-li LI. Design and implementation of testing case of black box based on equivalence partitioning. *Computer Knowledge and Technology*, 2:028, 2012.

[8] Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NASA Formal Methods*, pages 120–125. Springer, 2012.

[9] Doron Drusinsky and Man-Tak Shing. Verifying distributed protocols using msc-assertions, run-time monitoring, and automatic test generation. In *Rapid System Prototyping, 2007. RSP 2007. 18th IEEE/IFIP International Workshop on*, pages 82–88. IEEE, 2007.

[10] Corey Sandler Glenford J. Myers, Tom Badgett. *The Art Of Software Testing*. 2012.

TABLE III. SOME COMMERCIALLY AVAILABLE BLACK BOX AUTOMATIC TESTING TOOL

Sr. #	Tool Name	All-PairTesting	ModelBasedTesting	Run Time Error Detection	Load Testing	Manual Testing	MemoryError	DetectionRegression
1	Parasoft			✓	✓	✓	✓	
2	Reactis	✓		✓	✓		✓	
3	BoundsChecker			✓			✓	
4	Testwell							✓
5	AQ Time						✓	
6	Tessy							✓
7	Test Cast		✓					
8	Matelo		✓					

[11] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[12] Daniel Hoffman, Paul Strooper, and Lee White. Boundary values and automated component testing. *Software Testing, Verification and Reliability*, 9(1):3–26, 1999.

[13] Christian Holler, Kim Herzig, and Andreas Zeller. Fuzzing with code fragments. In *USENIX Security Symposium*, pages 445–458, 2012.

[14] J. R. Horgan and S. London. Data flow coverage and the c language. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, TAV4, pages 87–97, New York, NY, USA, 1991. ACM.

[15] Joseph R. Horgan, Saul London, and Michael R Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994.

[16] Alan Jorgensen and James A Whittaker. An api testing method. In *Proceedings of the International Conference on Software Testing Analysis & Review (STAREAST 2000)*, 2000.

[17] N Juristo, Sira Vegas, Martín Solari, Silvia Abrahao, and Isabel Ramos. Comparing the effectiveness of equivalence partitioning, branch testing and code reading by stepwise abstraction applied by subjects. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on*, pages 330–339. IEEE, 2012.

[18] Brian W Kernighan, Dennis M Ritchie, and Per Ejeklint. *The C programming language*, volume 2. prentice-Hall Englewood Cliffs, 1988.

[19] Priyanka Kulkarni and Yashada Joglekar. Generating and analyzing test cases from software requirements using nlp and hadoop. 2014.

[20] Huimei Liu and Huayu Xu. Research on code analysis and instrumentation in software test [j]. *Computer Engineering*, 1:028, 2007.

[21] Malte Lochau, Sebastian Oster, Ursula Goltz, and Andy Schürr. Model-based pairwise testing for feature interaction coverage in software product line engineering. *Software Quality Journal*, 20(3-4):567–604, 2012.

[22] Iulian Neamtiu, Jeffrey S Foster, and Michael Hicks. Understanding source code evolution using abstract syntax tree matching. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 1–5. ACM, 2005.

[23] A Jefferson Offutt and Alisa Irvine. Testing object-oriented software using the category-partition method. In *17th International Conference on Technology of Object-Oriented Languages and Systems*, pages 293–304, 1995.

[24] Thomas Ostrand. White-box testing. *Encyclopedia of Software Engineering*, 2002.

[25] Chandrashekar Rajaraman and Michael R Lyu. Reliability and maintainability related software coupling metrics in c++ programs. In *Software Reliability Engineering, 1992. Proceedings., Third International Symposium on*, pages 303–311. IEEE, 1992.

[26] Gregg Rothermel, Mary Jean Harrold, and Jeinay Dedhia. Regression test selection for c++ software. *Software Testing Verification and Reliability*, 10(2):77–109, 2000.

[27] Social RTI Health and Economics Research Research Triangle Park. The economic impacts of inadequate infrastructure for software testing final report, May 2002.

[28] Patrick J Schroeder and Bogdan Korel. *Black-box test reduction using input-output analysis*, volume 25. ACM, 2000.

[29] Anastasis A Sofokleous and Andreas S Andreou. Automatic, evolutionary test data generation for dynamic software testing. *Journal of Systems and Software*, 81(11):1883–1898, 2008.

[30] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5):297–312, 2012.

[31] W Eric Wong, Tatiana Sugeta, J Jenny Li, and José C Maldonado. Coverage testing software architectural design in sdl. *Computer Networks*, 42(3):359–374, 2003.