An Empirical based Object Oriented Coverage Analysis Using XML

Sasanko Sekhar Gantayat, Srinivas Prasad, K. Koteswar Rao
Department of Computer Science & Engineering,
GMR Institute of Technology, Rajam, Andhra Pradesh, India.
sgantayat67@gmail.com, [1]srinivas.prasad@gmrit.org, koteswarrao.k@gmrit.org

Abstract: *Testing of Object-oriented software has a number of features that make it different from conventional software testing. With the increase in size and complexity of modern software products, the importance of testing is rapidly growing. In this paper, a new methodology is proposed to evaluate the code coverage, its effectiveness and compared its advantages over other traditional techniques.*

Keywords: Test Coverage, Object Oriented Coverage, Coverage Measure, Test Coverage Tools, XML.

1. Introduction to Test Coverage

Test Coverage is the process of identifying the extent to which different test suites are appropriate in accessing a full complement of the source code in testing various components of the system. To the management and deployment team, it provides assurance on the comprehensive nature of the tests and provides insight into the areas where testing is inadequate, or where the labor expended on releasing a product can be reduced due to ineffective testing. It would be worthwhile to investigate the possibility of testing based on coverage analysis approach and study its effectiveness.

It is the process which provides a measure of how well the test suite actually tests the software. The major aspect of coverage analysis are[6]:

- Finding areas of a program not exercised by a set of test suites
- Creating additional test cases to increase the coverage
- Determining a quantitative measure of coverage which is an indirect measure of quality
- Identifying redundant test cases that do not increase coverage

2. Basic Coverage Measures

There are many coverage measures. Brief description about some of the basic coverage measures are given below [1,3,4,5,6,7,16,18]:

Statement Coverage: This measure reports the uncovered statements as well as a percentage of statements that are covered. It might not detect the control faults. It tells whether the nodes in a control flow graph (CFG) are executed or not. It can not check whether all the branches are executed or not.

CFG: It is a graphical representation of a program, in which each node consists of a set of statements that can be executed sequentially and edges are labeled with conditional statements.

Branch Coverage: It explores whether all branches are executed or not. It checks for the edges of the CFG i.e., it checks for all the branches that are formed with if statement, for statement, while, do while, switch statement and exception handlers. It is also known as Decision coverage. It reports whether Boolean expressions tested in control structures evaluated to both true and false.

Decision coverage: It reports whether Boolean expressions tested in control structures (like if statement, while statement) are evaluated to both true and false. The entire Boolean expression is treated as one true or false predicate although it contains logical - and or logical-or operators. In addition, it includes switch-statement cases, exception handlers and interrupts handlers.

Condition Coverage: It reports the true or false outcome of each Boolean subexpression, separated by logical – operators if any. It measures the subexpressions independently of each other. This measure is similar to Decision coverage but has better sensitivity to the control flow.

Path Coverage: It reports whether each of the possible paths is executed or not. A path is a unique sequence of branches from the function entry to the exit. It is also known as Predicate coverage. But due to the loop structures, many variations of this measure exist. It has two major disadvantages – the first one is the number of paths increases exponentially to the number of branches, the second one is that many paths are not possible to exercise due to relationships of data.

Method Coverage: It gives information that each method is invoked or not. In Java there are some methods declared as abstract i.e., they do not have a body. Therefore, this has to be taken care by the test coverage analyzer.

Class Coverage: It gives information that at least a single method of a class is invoked or not. Again in Java, there are some classes, called Interface, which do not have any method definition. So the test coverage analyzer should take care of such classes.
Loop Coverage: It reports whether each loop body in the program was executed zero times, exactly once or more than once (consecutively). For do-while loop, it reports whether the body is executed exactly once or more than once. The advantage of this measure over another measure is that it reports whether while-loop and for-loop in the program are executed more than once.

3. Techniques of Test Coverage

The test coverage tools work by instrumenting the program i.e., by inserting the "probes" into the program. The different tools vary in the way this instrumentation phase is done. Of course, adding probes to the program will make it bigger and execution slower.

The different types of instrumentation are discussed here [7,10].

Source Level Instrumentation: Some tools add probes at the source level. They analyze the source code and add additional code that will record the program execution status. Such tools actually do not create new source file. They intercept the compiler after parsing but before code generation to insert the changes.
This type of instrumentation is dependent on programming language i.e., the provider specifies which languages it supports. But it is independent of operating environment (processor, OS).

Executable Instrumentation: In this case the probes are added to the executable file. The tools will create a new executable after analyzing the present executable file.

This type of instrumentation is independent of programming language. But it is dependent on operating environment i.e., the provider specifies which processor to support.

Runtime Instrumentation: In this case the probes are not added until the program is run. The probes exist only in the in-memory copy of the executable file but not inserted into the file. Therefore to combine, the coverage tool initiates the program execution directly or indirectly.
Some coverage tools, alternatively, add a small bit of instrumentation to the executable, which does not affect the size or performance of the executable.

This type of instrumentation is also independent of programming language and dependent on operating environment.

4. Architecture of Coverage Tool

In this section the architecture of the general coverage tools is discussed. These are organization of the tool and the instrumentation technique.

4.1. Organization of Tool

Coverage tools helps in checking that how thoroughly the testing has been done [13,14].

A coverage tool first identifies the elements or coverage items that can be counted. At component testing level, the coverage items could be lines of code or code statements or decision outcomes (e.g. the True or False exit from an IF statement). At component integration level, the coverage item may be a call to a function or module.

Instrumenting the Code is the process of identifying the coverage items at component test level. Then automatically or manually a set of tests is run through the instrumented code. The number of coverage items are counted using the coverage tool and executed by the test suite. A report is generated on the percentage of coverage items that have been tested, and also identify the items that have not tested.

The common features of coverage measurement tools are:
• To identify coverage items (instrumenting the code)
• To calculate the percentage of coverage items those were tested by a set of tests
• To report coverage items those have not been tested yet
• To generate stubs and drivers

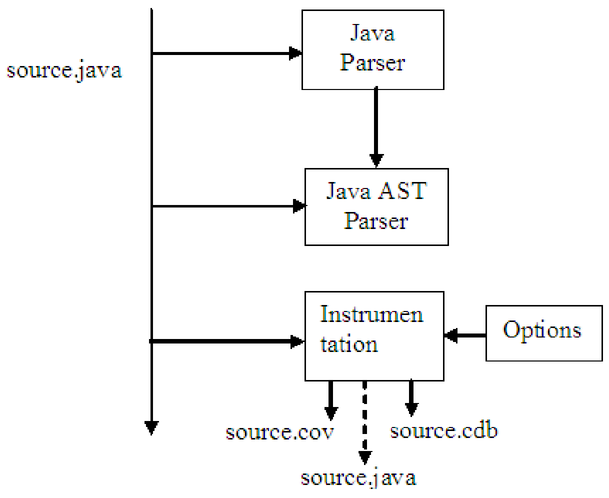

Fig-1. Basic Organization of Tool

There are numerous coverage tools available commercially. But here we will discuss a typical architecture which is common to much different instrumentations and program analysis tools. Although it is language independent, the approach is made with reference to Java language.

The internal organization of a coverage tool is depicted above. Here the Java source file (source.java) is parsed first and an Abstract Syntax Tree (AST) is generated. Then the AST is parsed during which code coverage instrumentation is inserted into the source file. According to the options chosen, several output files are generated. The *source.cov* is an instrumented version of the original source file. And the *source.cdb* contains a database of all the changes and instrumentation points made to the source file. Hence, the original source file can be constructed from cov and cdb files. In the other way, the original source file may be saved in another directory and the *source.cov* is renamed to *source.java.* Now this instrumented file may be compiled by a Java compiler and executed either in a unit test or as part of the original application. The required type of coverage can be mentioned in the options.

The primary function of the parser is to generate the AST for the entire source file. In addition to the language components, the AST contains the line and column references to the original source file for each node. This simplifies the identification of grammatical structures and variables that are to be instrumented and their locations. Hence, AST makes the instrumentation easier.

4.2. Instrumentation Technique

Generally, instrumentation is the technique of inserting probes into the source code which detect the coverage information about the tests run on the source code. So after insertion, the source code needs to be recompiled. But two factors are taken care about instrumentation – first, it should have the least impact on the execution time of the source code and second it should not affect the functionality of the unit. The most common instrumentation technique can be explained with a tool Advanced Java Coverage Tool (AJCT) as follows.

For each path determining relational expression in the source file, the relational expression is replaced by the following method:

jc.pc(relational expression, identifier)

where relational expression is the original expression of the source, the identifier is a unique integer value assigned by AJCT and *pc* is a method defined in a class *jc*. For example, the expression (p<q) would be replaced by jc.pc(p<q,16) if it is the 8th instrumentation point within the AST.

The purpose of this method is to increment the elements of the array maintained by AJCT. If the relational expression evaluates *true*, then the array element with index equal to identifier is incremented otherwise the array element with index equal to identifier+1 is incremented. So during post-execution analysis, the number of times the relational expression is evaluated can be determined from the array.

This approach allows Java virtual machine to evaluate the original relational expression and pass it as an argument to jc.pc and it returns the original Boolean result. This approach is used for conditional expressions like if, while, for etc. But, for non-branch related statements like switch statement or try-catch block, AJCT replaces the relational expression parameter with the Boolean value *true* and instruments the first location where an executable statement can be placed.

To obtain method coverage same technique can be applied. If the method contains conditional path determining expression, then the default relational expression instrumentation can indicate whether the method is reached or not. If the method contains non-loop conditional expression, then it can indicate the number of times the method was called. Otherwise, the AJCT uses jc.pc (true, identifier) as the first statement in the method and counts the number of times it was invoked.

For all cases, AJCT logs the value of the relational expression and also maintains a cumulative count of each true or false result. From such log files and the array maintained by AJCT, various coverage reports can be generated.

5. Survey of Coverage Tools

Here we have discussed few readily available coverage tools [3,8,13,17].

LDRA Testbed: It is a static analysis and code coverage tool suite for C and C++. It is a unique quality control tool that provides powerful source code analysis and testing facilities for the validation and verification of software applications. It is a fully integrated tool suite for static analysis and code coverage.

Static Analysis analyses the code and provides an understanding of the code structure and also it measures code coverage of statements, branches, test paths and conditions. It uses automatic instrumentation technique to measure the code coverage levels during test process and with the help of which it is able to detect the untested part of the code. Also, when an error occurs during testing, it presents the corresponding code area being executed in reports. This makes very helpful for the developer to focus on that specific area of code and save time.

Bullseye Coverage: It is a full-featured code coverage analyzer for C/C++ running on Microsoft and Unix operating systems. It quickly finds untested code and measures testing completeness. It also increases testing productivity by showing the regions of your source code that are not adequately tested. Bullseye Coverage enables us to create more reliable code and save time. It reports Function coverage which enables us to quickly know what major areas of the software are untested i.e., to get a quick overview. And also it reports condition/ decision coverage to know whether every control structure with every possible decision outcome as well as every possible condition outcome are checked i.e., to give high precision code. It uses source code instrumentation which is required for the best coverage analysis. Therefore, the code size increases by 1.4 times and the execution time increases by 1.2 times. It works with everything in C++ and C, including system-level and kernel mode.

Clover: It is a low cost code coverage tool for Java. Clover provides a method, branch and statement coverage for projects, packages, files and classes. Unlike tools that use bytecode instrumentation, it uses source code instrumentation and it produces the most accurate coverage measurement for the least runtime performance overhead.

As the code under test executes, code coverage systems collect information about which statements have been executed. This information is then used as the basis of reports. In addition to these basic mechanisms, coverage approaches vary on what forms of coverage information they collect. There are many forms of coverage beyond basic statement coverage including conditional coverage, method entry and path coverage.

Clover uses these measurements to produce a Total Coverage Percentage for each class, file, and package and for the project as a whole. The Total Coverage Percentage allows entities to be ranked in reports. The Total Coverage Percentage (TPC) is calculated as follows:

TPC = (BT + BF + SC + MC) / (2*B + S + M)
where
BT - *branches that evaluated to "true" at least once*
BF - *branches that evaluated to "false" at least once*
SC - *statements covered*
MC - *methods entered*
B - *total number of branches*
S - *total number of statements*
M - *total number of methods*

Clover is designed to measure code coverage in a way that fits with the current development environment and practices. Clover's IDE Plugins provide developers with a way to quickly measure code coverage without having to leave the IDE.

Dynamic Code Coverage: Without using any compile/link time instrumentation, it gathers coverage information by using runtime instrumentation. Each function/method, line, decision and branch is evaluated for execution and a detailed coverage file is generated in the process. Coverage files from multiple runs can be assembled to get a detailed Coverage Analysis of a particular process.

It can be used to gather information at any point in the software lifecycle. In development, it can be used for unit testing. In testing, it can be used to determine test suite effectiveness. In pre-production, Dynamic Code Coverage can determine which modules are of most interest to a particular customer. In production, it can be used to determine which features and modules are actually being used. And hence the various outputs can be summarized in a variety of ways in order to get different views on the depth of coverage. Platform – Solaris, Linux

SD's Java Test Coverage: Semantic Design supplies test (or code) coverage tools for arbitrary procedural languages. Such tools provide statistics and detail information about which parts of an application program have been executed (usually by a test suite). This information is useful to determine the readiness of software for actual use. The type of coverage information collected is branch coverage, which subsumes statement coverage.
SD's test coverage tools operate by inserting language-specific probes for each basic block in the source files of interest before compilation /execution. At execution time, the probes record which blocks get executed ("coverage data"). On completion of execution, the coverage data is typically written to a test coverage vector file. Finally, the test coverage data is displayed on top of source text for the system under test, enabling a test engineer to see what code has (not) been executed, and to see overall statistics on coverage data.
Platform - Probe installer operates on Win/NT, Win2K, WinXP Java applications under test and the coverage display tool can run on any Java2 platform.

A sample SD's test coverage report is presented below.

```
Semantic Design's TEST COVERAGE REPORT
Probe Reference File:
    C:\users\idbaxter\Parlanse\P0Compiler\p0c.prf

Test Coverage Vectors:
    C:\users\idbaxter\Parlanse\P0Compiler\RegressionTest\

%TestCoverage_2002_11_19_09_21_44_000.tcv

SUMMARY:
    Total Probes: 9481
    Total Files: 7
    4780 probes covered.     4701 probes uncovered.
    50.4% probes covered.    49.6% probes uncovered.
COVERAGE REPORT BY FILE:
    [1] C:\users\idbaxter\Parlanse\P0Compiler\P0Compiler.c   49.9%
                          uncovered  4587/9205
    [2] C:\users\idbaxter\Parlanse\P0Compiler\P0CriticalRanges.c
47.5%
                    uncovered  38/80
    [3] C:\users\idbaxter\Parlanse\P0Compiler\P0Cwin32.c   100.0%
                       uncovered  30/30
    [4] C:\users\idbaxter\Parlanse\P0Compiler\P0ExceptionRanges.c
17.7%
                     uncovered  9/51
    [5] C:\users\idbaxter\Parlanse\P0Compiler\P0File.C   100.0%
                       uncovered  0/0
    [6] C:\users\idbaxter\Parlanse\P0Compiler\P0SourceXRef.c   34.3%
                    uncovered  37/108
    [7] C:\users\idbaxter\Parlanse\P0Compiler\crc32.c   0.0%
                     uncovered  0/7
FILES COMPLETELY COVERED:
    [1] C:\users\idbaxter\Parlanse\P0Compiler\P0File.C   0 probes
    [2] C:\users\idbaxter\Parlanse\P0Compiler\crc32.c   7 probes

COVERAGE BY SUBSYSTEM/DIRECTORY:
    (1) C:\users\idbaxter\Parlanse\P0Compiler   49.6% uncovered
4701/9481
        [1] C:\users\idbaxter\Parlanse\P0Compiler\P0Compiler.c
        [2] C:\users\idbaxter\Parlanse\P0Compiler\P0CriticalRanges.c
        [3] C:\users\idbaxter\Parlanse\P0Compiler\P0Cwin32.c
        [4] C:\users\idbaxter\Parlanse\P0Compiler\P0ExceptionRanges.c
        [5] C:\users\idbaxter\Parlanse\P0Compiler\P0File.C
        [6] C:\users\idbaxter\Parlanse\P0Compiler\P0SourceXRef.c
        [7] C:\users\idbaxter\Parlanse\P0Compiler\crc32.c
```

EMMA: EMMA is an open-source toolkit for measuring and reporting Java code coverage. It is different from other tools for its unique feature support for large-scale enterprise software development while keeping individual developer's work fast and iterative.
It is a pure Java coverage tool based on bytecode instrumentation. It provides two options for instrumentation – offline or online. In the offline mode instrumentation is done explicitly and in the online mode it is done in the JVM. It supports line, method, class coverage but it doesn't support branch and path coverage. And the coverage outputs are consolidated at method, class, package and 'all classes' levels. It does not require accessing the source file, rather it instruments the individual .class files or entire .jar file. Since the runtime overhead of added instrumentation is small (5-20%), it is quite fast. Platform – any Java platform

CTC++: It is a powerful instrumentation-based test coverage and dynamic analysis tool for C and C++ code. It provides all coverage measures like, function coverage, decision coverage, statement coverage, condition coverage. It operates in following three steps:

- Use the CTC++ Preprocessor (*ctc*) utility for instrumenting and compiling the C or C++ source files of interest and for linking the instrumented program with the CTC++ run-time library. At this phase *ctc* maintains a symbol file, MON.sym by default, where it remembers the names of the instrumented files and what they contained.

- Execute the test runs with the instrumented program. When the instrumented code portions are executed, CTC++ collects the coverage and function timing history in memory. Normally at the end of the program, automatically by CTC++, the collected counters are written to a data file, MON.dat by default. If there were previous counters in the data file, they are summed up.

- Use the CTC++ Postprocessor (ctcpost) utility for putting one or more symbol files and data files together and produce the human readable textual reports. One of them, the Execution Profile Listing, can be further processed with ctc2html utility for getting and an easy-to-view hierarchical and color-coded HTML representation of the coverage information. With the ctc2excel utility the coverage data can be converted to a TSV (tab separated values) file, suitable input to Excel (or any spreadsheet application). Platform-Windows 2000/NT, HPUX, Solaris, Linux.

GlassJAR Toolkit: The type of coverage information collected by this tool is branch coverage, line coverage and method coverage. It presents coverage for individual test cases in addition to the grand totals. Since, it operates on bytecode, a tester need not to install a full development environment and source code. It supports any JVM compliant with the Java 2 standard. It supports J2EE, J2SE and J2ME i.e.; It can test for servlets, EJB, any stand alone application or J2ME midlets. The best part is the output report format is customizable. Platform - Windows NT/2000/XP, Solaris, HP-UX, Linux, others (any Java 2 platform)

6. Comparison of Tools

Coverage Tool	Company	Type of Coverage	Instrumentation Type	Language
Bullseye	Bullseye Testing Tech.	Function, Condition	Source Code	C, C++
Clover	Cenqua	Statemen, Method, Branch	Source Code	Java
Dynamic Code Coverage	Dynamic Memory Solutions.	Function, Decision, Branch	Run Time	Java
Java Test Coverage	Semantic Design	Branch	Source Code	Java
CTC++	Testwell	Function, Decision, Branch, Condition	Source Code	C, C++
EMMA	Source Forge	Statemen, Method, Class	Byte Code	Java
GlassJAR Toolkit	Tester's Edge	Statemen, Method, Branch	Source Code	Java
LDRA Testbed	LDRA	Statemen, Condition, Branch, Path	Source Code	C, C++

Table 1. Comparison of different Test Tools

7. New Approach for Coverage Analysis

We have implemented a new approach to realize a prototype tool. We have named our tool Model-based Mutation Tester (MMT).

From the model, we generate coverage matrix based on our proposed operators, and then automatically seed specific type of faults. When run the test suite to generate a test summary report of errors not detected by a test suite.

This approach has the following components:
1. Test Suite (to test and validate the test)
2. Test Executor (to execute and test the result with the expected result)
3. Test Oracle (to check the failure or success of a test)
4. Code (input code)
5. Result Analyzer
6. Log file (generates the report for each test case)
7. Test Summary report

Our proposed model is shown in figure 2. In our proposed methodology, automated tests build the test data, run the test and examine the result automatically. By using proper test suit, tests oracle and then log is generated. Result analyzer then analyses the log to generate the test summary report.

The models are syntactically and semantically analyzed using an XML parsers and Java, which are the required mutant program

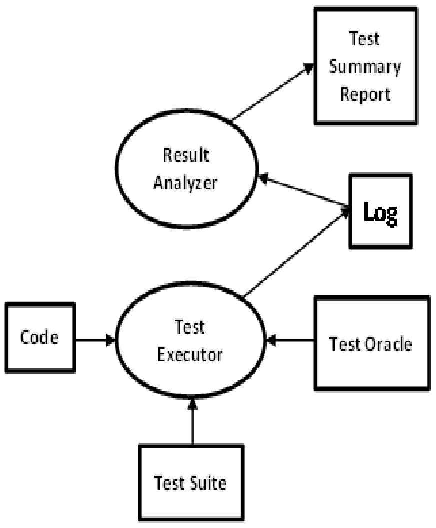

Fig 2: A Schematic Model of the new Approach

8. Experimental Result Analysis

Test Coverage is the process of identifying the extent to which different test suites are appropriate in accessing a full complement of the source code in testing various components of the system.

In this experiment, four project cases (Student, Web, Professional-L1, and Professional-L2) are studied using XML and proposed approach. This approach is best suited for the codes having minimum 2000 LOCs for each project. The summary of the test case coverage analysis is explained in the following table (Table 2) and the graph (Fig.3).

As shown in figure 3, in our approach of Test coverage for class and method enlarge comparatively then traditional testing approach.

In all the four cases, our result shows a better performance over other traditional test case coverage tools.

Project	LOC	System Level Coverage		Method Level Coverage	
		Our Approach	Traditional Approach	Our Approach	Traditional Approach
Student	2870	68	56	62	54
Web	2654	72	63	65	57
Professional-L1	2546	64	47	61	58
Professional-L2	2343	79	74	74	54

Table 2: Summary of Test Coverage on Object Oriented Code

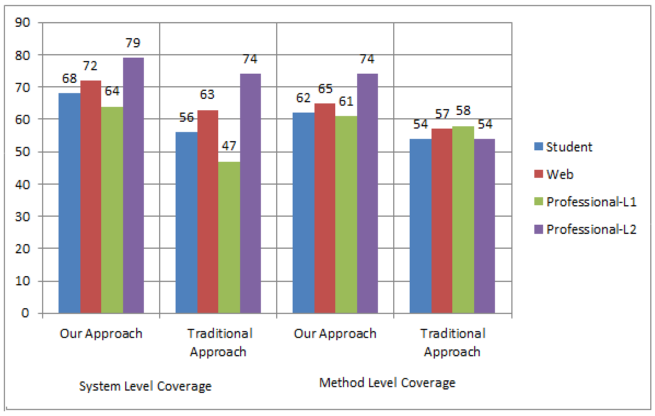

Fig-3: Comparative Performance Results

9. Conclusion & Future Work

In this paper, a new method is considered for test case coverage analysis of object oriented programs on Java. From this it is observed that the higher percentage of code coverage gives the higher experimental accuracy of test case coverage. It is also realizable if reference codes more. So Benchmarks must be carefully designed to include all code relevant to a full application. This methodology and the experiment can be modified to different object oriented programs like python, Visual C++, etc.

References

1. Rajib Mall, *Fundamentals of Software Engineering*. PHI Learning Pvt. Ltd., 2009.
2. Andrew Haigh, *Object Oriented Analysis & Design,* Tata McGraw Hill, 2001.
3. Code Coverage analysis at www.bullseye.com
4. P.Jalote, *An integrated approach to Software Engineering*, Springer Science & Business Media, 2012.
5. Jerry Gao, Raquel Wspinoza, *Testing coverage analysis for software component validation*, Proceedings of 29th International Computer Software & Applications. Conference (COMPSAC'05), 2005.
6. Test Coverage at www.patersontech.com
7. Raghu Lingampally, Pankaj Jalote, *A Multipurpose Code Coverage Tool for Java*, Proceedings of 40th Annual Hawai International Conference on System Sciences (HICSS'07), 2007.
8. Code Coverage from en.wikipedia.org
9. Khalid Alemerien and Kenneth Magel, *Examining the Effectiveness of Testing Coverage Tools: An Empirical Study*, International Journal of Software Engineering and Its Applications Vol.8, No.5, Pp.139-162, 2014.
10. T. W. Williams, R. Kapur, M. R. Mercer, J. P. Mucha, *Code Coverage, What does it mean in terms of quality?,* Proceedings Annual Reliability and Maintainability Symposium, IEEE Conference Publications, 2001.
11. M. J. Harrold, J. A. Jones, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. *Regression Test Selection for Java Software*, Proceedings of the 16th ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications *(OOPSLA '01)*, 2001.
12. J. A. Jones and M. J. Harrold. *Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage*, IEEE Transactions on Software Engineering, Vol. 29, Issue.3, March 2003.
13. M. Lyu, J. Horgan, and S. London. *A Coverage Analysis Tool for the Effectiveness of Software Testing*, IEEE Transactions on Reliability, Vol. 43, Issue. 4, Pp. 527 - 535, 1994.
14. Abdelilah Sakti, Gilles Pesant, and Yann-Gael Gueheneuc, *Instance Generator and Problem Representation to Improve Object Oriented Code Coverage,* IEEE Transactions on Software Engineering, Vol. 41, No.. 3, MARCH 2015.
15. Chun-Chia Wang, Wen C. Pai, Timothy K. Shih, *An Automated Object-Oriented Testing for C++,* Inheritance Hierarchy, IEEE, 1997.
16. Alan W. Williams, Robert L. Robert, *A Measure for Component Interaction Test Coverage*, ACS/IEEE International Conference on Computer Systems and Applications, pages 304– 311, June 2001.
17. D. Nageswara Rao, M. V. Srinath, P. Hiranmani bala, *Reliable Code Coverage Technique in Software Testing*, Proceedings of the IEEE International Conference on Pattern Recognition, Informatics and Mobile Engineering, February 21-22, 2013.
18. McCabe, T. J., *A Complexity Measure*, IEEE Transactions on Software Engineering. 1976.