Simplifying Control Flow Graphs for Reducing Complexity in Control Flow Testing

Myint Myitzu Aung[#1], Kay Thi Win[*2]

University of Computer Studies, Mandalay
Myanmar

Abstract — ***Control Flow Graph (CFG) is the graphical representation that contains all possible paths in the program execution. It is very essential part of the control flow testing but there are some challenges in generating the effective CFGs. These are how to reduce the complexity and isomorphism of the CFGs and how to check the feasibility of these CFGs. In this paper, these challenges are addressed by introducing the simplifying of CFGs. In the control flow testing, the execution order is determined which is the instruction or statement order of the program through a control structure and the tester needs to select a specific part of a large program to set the testing path. The CFGs of large programs are sliced to simplify and to reduce the complexity by using a program slicing technique, Tree Slicing which is also called Path-Sensitively Sliced CFG (PSS-CFG). It is a slicing and merging technique for different sub-trees under certain conditions. By using this technique, the complexity of these CFGs is reduced due to the simplifying of these graphs. Moreover, this paper shows that the code coverage of the program of simplified CFGs is also improved according to the experimental results of the benchmark dataset that is different from other recent researches because this system is tested in Java by converting the C dataset.***

Keywords — ***Control Flow Graph (CFG), Control Flow Testing, Path-Sensitively Sliced CFG.***

I. Introduction

Control flow testing is a testing technique and it is a type of white box testing. It can determine mainly for the execution order of instructions or statements included in the program through a control structure which is used to develop a test case for the program and the tester needs to select a specific part of a large program to set the testing path. Therefore control flow path plays as an essential role in control flow testing. To obtain the control flow path, the Control Flow Graph (CFG) of a program must be considered [1].

A control-flow graph (CFG) is a directed graph representation of a program and usually a sparse graph. CFGs include all possible control paths in a program. This makes CFG a great tool to obtain control-flow behavior of its process. Vertices in a CFG give the level of detail, such as instruction-level or basic block level that cannot be further divided. Edges in CFG represent control jumps and are classified into two types - forward and backward. Branch instructions, function calls, conditional and unconditional jumps account for forward edges. Virtual calls and indirect function calls are also considered as forward edges but their destinations are difficult to determine. Loops and returns generally account for backward edges. The integrity among duplicate processes that run on replica nodes needed to be verified with the information available in a CFG [2].

Similarity check among the logic of the programs can be performed by comparing their CFGs for isomorphism. CFG isomorphism is hard and time consuming which is a complex problem, sometimes known to be NP-complete. To reduce the complexity, CFGs can be reduced to subtrees or subgraphs before performing any coherence or integrity checks. A CFG can be converted to a subtree using various methods [1]. Among these methods, Tree Slicing is an effective technique which is used to slice and merge many different Symbolic Execution (SE) sub-trees under certain condition. To describe the degree of the source code that has been tested, different control flow code coverage measures are used. Two different approaches for code coverage are: semantic and syntactic approach. The semantic approach describes how syntactic coverage has to be changed to only all feasible information flows and the syntactic approach describes code coverage in a flow graph. Coverage analysis is to reduce faults and increase the quality of the product at the end [4]. In measuring the code coverage, the original input program may has many infeasible branches so that there is needed to reduce complexity and improve performance in the past. Thus, this paper is intended to overcome these problems.

In this paper, the novel idea including the key steps to reduce complexity and improve the code coverage is proposed. The main key steps are converting C program into Java, generating CFGs, simplifying CFGs by using program slicing, and measuring its coverage. Firstly, the input C program is converted into Java and generate control flow graph. Then it is generated into Symbolic Execution Tree to divide all possible true and false cases. In fact, the advantages of dividing both true and false cases are

- Validating that all the branches in the code are reached.
- Ensuring that no branches lead to any abnormality of the program's operation.
- Eliminating the problems that occur with statement coverage testing [13].

Thus, the symbolic execution tree (SETree) is generated. And then, this tree is transformed into Path-Sensitively Sliced CFG (PSS-CFG) by removing edges and sub-trees with the algorithm of splitting or merging as shown section 3. According to this PSS-CFG, simplified, reduced and sliced program is obtained. This fact leads to reduce complexity and improve coverage. Moreover, the system is novel experiment because it is tested with converted Java dataset although the original benchmark dataset is in C. Finally, the experimental result of reduced complexity and improved coverage is presented in section 5 which is due to the effects of PSS-CFG for simplifying CFGs. The detail design architecture of the proposed system is shown in section 4 and its related works is described in section 2.

II. Related Work

This paper is mainly focused on the concept of simplifying control flow graphs (CFGs) with the aims of reducing complexity and improving code coverage. Therefore the following related research papers are studied to compare and to obtain other knowledge.

S. Aditham and other authors presented linear-time algorithm in the purpose of equivalence of classes without examination of each path in a CFG. Their proposed algorithm is to construct event-flow graph (EFG) which is a compact derivative of the CFG. Every path in this graph is represented as an equivalence class of paths in the corresponding CFG. EFGs are defined as the set of events that are defined by the analyzed property. In the original CFG, equivalence classes are guaranteed to reserve all event traces. This reduction is close to 99% for CFGs with a large number of paths [1].

R. Gold presented graph reduction system which is useful for directed graphs and control flow graphs. The author applied these graphs reduction to software testing, the program control flow and the three graph types including control flow graphs, decision graphs and segment graphs. Nodes in control flow graphs represent as statements. Thus node coverage is means that the statement coverage. Analogously, edges represent branches so that edge coverage is defined as branch coverage. Therefore three different abstractions from programs are the types of defined graph. Their paper established for these graph types as a uniform and formal basis in the use of manual and tool supported control flow oriented test case generation and graphical representation of programs[3].

Y. Dubey, D. Singh, and A. Singh presented a method to reduce the number of test suites with the help of mining concepts thereby facilitating the mining from test cases. Researchers implemented their system with K-Means Algorithm to divide the input domain or test suite of sample module into the partitions of an expected number (K). Their system is test suite reduction system that is guided by the code coverage criteria defined in order to select a test suite that will give 100% code coverage [9].

I. Papadhopulli and E. Mece presented the combination of several possible criteria since they are intended to many different parts of unit under test. The authors used an automated test suits generator called EvoSuite with different configurations for six open source projects. It can investigate how the combination of coverage criteria influences the test suite length and the coverage achieved. According to their experiments, the overall coverage and mutation score are increased by the use of multiple criteria with the cost of increase in test suite length [10].

Satyam proposed an automated technique that appears as a promising technique to eliminate test time and effort. The author used a code transformation technique. The input is simple java program and it is transformed by using four algorithms. The author used Quine Mc-cluskey method and Petric methods to achieve the transformed program. After transforming the program, a tool called Cobertura provides the branch coverage of that transformed program. The measurement of branch coverage using this transformed program is higher than the coverage of original program [11]. The technique is to increase in branch coverage that is compared with traditional techniques.

III. Background

A. Control Flow Graph (CFG)

A control flow graph (CFG) is a representation which uses graph notation, of all paths traverse through a whole program during execution. CFGs are built for static analysis reason. Static analysis is the examination which does not include the execution of the program being analyzed but rather gaining information from other sources like reviews, documentation, automated tools or formal methods for analysis. From the evaluation, information like coverage or reachability can be obtained [6]. To generate Java Program Control Flow Graph, Dr.Garbage Tool is used in this system because it is a well-known tool for java CFG in eclipse.

Among the CFG generators, Dr. Garbage Tool is a popular technique in Java Community of Software Development Group. This project collects open source tools for java program of control flow analysis, including Java Bytecode Visualizer, Sourcecode Visualizer and control flow factory for visualization and generation of the various control flow graphs.

B. Simplifying Control Flow Graph

There are many different methods to simplify CFGs in which program slicing is used in this paper. Program Slicing is an essential testing technique because it can provide program understanding by dividing the program into smaller program codes depending on the various dependencies (method, data, control, call etc) between the statements. By applying the program slicing, all slices contain statements that appropriate to specific variables and ignores other statements. Program slicing approach is classified depending on the slicing direction and run-time environment. Depending on the slicing direction, there are two directions in slicing: forward or backward, and depending on the run time environment, slicing may be dynamic or static [8].

C. Tree Slicing of Program Slicing Technique

It is the technique of program slicing and also called Path-Sensitively Sliced CFG (PSS-CFG), a powerful tree slicing technique that used to slice and merge many different Symbolic Execution (SE) sub-trees under certain condition. To simplify the CFGs, it needs to become PSS-CFG. Therefore this proposed system uses two-steps algorithm. The first step is to generate SETree annotated with dependencies and the second step is to remove sub-tree and edges according to the given criteria to obtain the PSS-CFG. To perform the first step of generating the SETree annotated with dependencies, the following algorithms are used [7].

```
GENPSSCFG ($v \equiv \langle \ell, s, \Pi \rangle$)
1: if $\exists\, v' \equiv \langle \ell, s', \Pi' \rangle$ s.t. $v$ and $v'$ satisfy Eqn. 4
2:     then MERGE ($v, v'$)
3: else if $v$ is at a branch point then
4:     SPLIT($v$)
5: else
6:     SYMEXEC ($v$)
```

Fig. 1 Generating PSS-CFG

```
MERGE ($v, v'$)
1: $\overline{\Psi}_v := \overline{\Psi}_{v'}$
2: $\sigma_v := \sigma_{v'}$
3: $\mathcal{S} := \mathcal{S} \cup \text{merged}(v, v')$
```

Fig. 2 Merging

```
SPLIT ($v \equiv \langle \ell, s, \Pi \rangle$)
1: $\overline{\Psi}_v$ := true
2: foreach transition $\ell \xrightarrow{\text{assume(c)}} \ell'$ do
3:     if ($v$ is a loop header) then
4:         $v' \triangleq \langle \ell', \cdot, \text{invariant}(v) \wedge [\![c]\!]_s \rangle$
5:     else
6:         $v' \triangleq \langle \ell', s, \Pi \wedge [\![c]\!]_s \rangle$
7:     if $v'$ is infeasible state then
8:         $\mathcal{S} := \mathcal{S} \cup \text{inf\_edge}(v \xrightarrow{\text{assume(c)}} v')$
9:         $\overline{\Psi}_{v'}$ := false, $\sigma_{v'} := \emptyset$
10:    else
11:        $\mathcal{S} := \mathcal{S} \cup \text{edge}(v \xrightarrow{\text{assume(c)}} v')$
12:        GENPSSCFG ($v'$)
13:    $\overline{\Psi}_v := \overline{\Psi}_v \wedge \widehat{wlp}\,(\overline{\Psi}_{v'}, \text{assume(c)})$
14:    $\sigma_v := \sigma_v \sqcup \widehat{pre}\,(\sigma_{v'}, \text{assume(c)}, s)$
15:    if $\delta \equiv v \xrightarrow{\text{assume(c)}} v'$ satisfies Eqn. 3 then
16:        $\mathcal{S} := \mathcal{S} \cup \text{in\_slice}(v \xrightarrow{\text{assume(c)}} v')$
```

Fig. 3 Splitting

```
SYMEXEC ($v \equiv \langle \ell, s, \Pi \rangle$)
1: if $\nexists$ transition relation $\ell \xrightarrow{\text{x:=e}} \ell'$ then
2:     $\overline{\Psi}_v$ :=true, $\sigma_v := \mathcal{V}$
3: else
4:     $v' \triangleq \langle \ell', s[x \mapsto [\![e]\!]_s], \Pi \rangle$
5:     $\mathcal{S} := \mathcal{S} \cup \text{edge}(v \xrightarrow{\text{x:=e}} v')$
6:     if $v'$ is not a loop header
7:         GENPSSCFG ($v'$)
8:     $\overline{\Psi}_v := \widehat{wlp}\,(\overline{\Psi}_{v'}, \text{x:=e})$
9:     $\sigma_v := \widehat{pre}\,(\sigma_{v'}, \text{x:=e})$
10:    if $v \xrightarrow{\text{x:=e}} v'$ satisfies Eqn. 2 then
11:        $\mathcal{S} := \mathcal{S} \cup \text{in\_slice}(v \xrightarrow{\text{x:=e}} v')$
```

Fig. 4 Symbolic Execution

The second step is to get final PSS-CFG (Fig. 10) by using following transformation rules [7].

- Rule 1: The statement can be removed if the LHS of an assignment statement does not include in the dependency set.
- Rule 2: If a branch point has only one feasible path which arises from it, it can be replaced or removed.
- Rule 3(called "Tree Slicing"): If both the "then" and "else" cases contain no statement that is included in the slice, an entire branch is inappropriate to the target point and can be removed [14].

D. Slicer for Simplifying CFGs

For both forward and backward slicing, Indus-Kaveri is the most appropriate tool in this system. The purpose of this tool is to simplify program analysis, program understanding, and testing. Indus is an effective framework for slicing and analyzing concurrent java program, and Kaveri has a rich features and it is an Eclipse-based GUI front-end for Indus java slicer. It uses the Indus slicer to develop the program slice in Java and then shows the output results in the Eclipse editor. As an Indus works on Jample, the criteria in the Slicing Criteria Factory for slicing is specified as Jimple chunks. This working flow is not deviated to Control Flow Graph (CFG) because it is the main issue in slicing concurrent java programs. It is to determine the control flow and data flow between program points of different threads [5].

IV. SYSTEM ARCHITECTURE

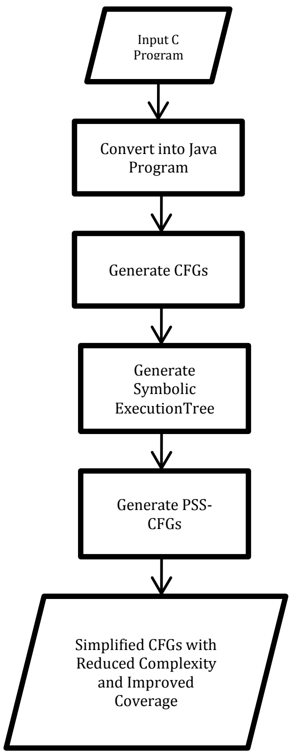

Input C Program
Convert into Java Program
Generate CFGs
Generate Symbolic ExecutionTree
Generate PSS-CFGs
Simplified CFGs with Reduced Complexity and Improved Coverage

Fig. 5 System Architecture

The architecture of the proposed system has four parts as shown in Figure 5. They are converting C program into Java, generating CFGs, generating symbolic execution tree (SETree) and generating PSS-CFGs (Simplified CFG).

Firstly, the input C program is converted into Java program. This Java program is generated into appropriate CFG and then the system use the two-steps algorithms to obtain PSS-CFG in which the first step is to generate SETree annotated with dependencies and the second step is to transform this tree by removing sub-tree and edges, to simplify CFG as shown in Figure 1, 2, 3 and 4.Finally, the output transformed program of simplified CFG (PSS-CFG) can be measured in complexity and code coverage. After simplifying the CFGs, the result is ensured that the complexity is reduced significantly and the code coverage is more improved than the original program.

A. Motivating Example

In this example, the steps of the performance of the technique on a small program to reduce complexity and improve coverage are described. The original input program and its Control Flow Graph (CFG) are shown in Figure 6 and 7.

```
if(read(c)) flag=1
  else flag=0
x=2
if(read(d)) y=4
  else y=5
if(flag) z=y+x
  else z=x+1
TARGET(z)
```

Fig. 6 The Original Input Program

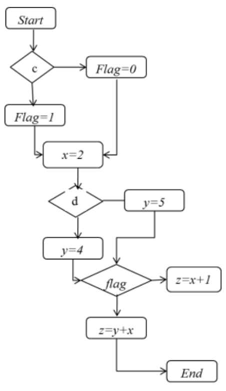


Fig. 7 The Control Flow Graph (CFG) of Original Program

The CFG of input program is needed to simplify in the purpose of reduction complexity and improving coverage. Therefore the proposed system divides all possible true and false cases as shown in Figure 8.

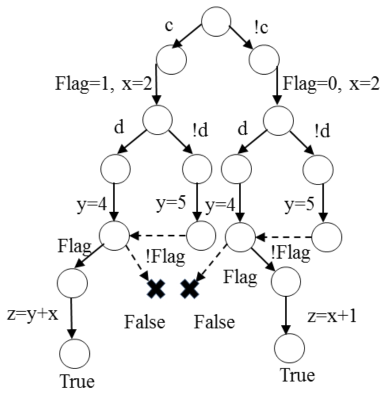
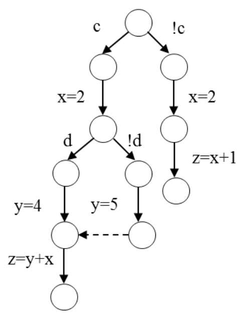


Fig. 8 Dividing all possible true and false cases(SETree)

The SE tree executes both the statement c is true and false firstly and then, flag=1 and x=2 are as usual. Upon success of the next branch, it splits into two: with d and !d. The context d executes y=4 and reaches the last branch. Again it splits into two: with flag and !flag, finally executing z=y+x before reaching the terminal point. At this point the path formula is: c^flag=1^x=2^d^y=4^flag^z=y+x1 which is satisfied, because this path is feasible to get the target variable z. But the path formula c^flag=1^x=2^d^y=4^!flag is not satisfied, thus the path is infeasible to be removed. The algorithm works in this fashion to be a PSS-CFG but path-sensitively makes the SETree in the number of branches. Therefore its size is checked by Merging (Fig. 2).



```
if(read(c)) {
  x=2
  if(read(d))
    y=4
  else
    y=5
  z=y+x
}
else {
  x=2
  z=x+1
}
```

Fig. 9 New Parth Sensitively Sliced CFG and its transformed program

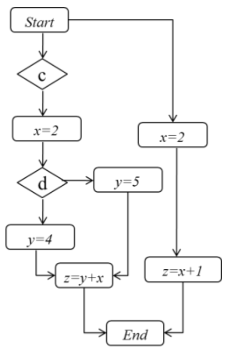


Fig. 10 Control Flow Graph of new PSS-CFG

By comparing cyclomatic complexity of the CFG of original input program (Fig. 7) (before simplifying) and the CFG of the final PSS-CFG (Fig. 10) (after simplifying), the complexity can be reduced significantly by using the following cyclomatic complexity metric.

$$V(G) = e - n + 2 \qquad (1)$$

Where, *V(G)* is the cyclomatic complexity value of control flow graph *G*, *e* is the number of edges and *n* is the number of nodes in that CFG.

For original input program before simplifying,

$$V(G) = 14 - 12 + 2 = 4$$

For transformed program after simplifying,

$$V(G) = 11 - 10 + 2 = 3$$

Thus, the complexity is reduced.

By comparing code coverage of the CFG of original input program (Fig. 7) (before simplifying) and the CFG of the final PSS-CFG (Fig. 10) (after simplifying), the coverage can also be improved significantly. The original coverage cov_b can be measured for this program (Fig. 7) by using commonly used original coverage metric.

For original input program before simplifying,

$$cov_b = \frac{br_{exe}}{br_{total}} * 100\% \qquad (2)$$

$$= \frac{2}{4} * 100\%$$

Therefore, the code coverage 50% is obtained.

For transformed program after simplifying, the final PSS-CFG (Fig. 10) can be measured for code coverage .

$$cov_b = \frac{br_{exe}}{br_{total}} * 100\%$$

$$= \frac{2}{3} * 100\%$$

The code coverage 66.67% is obtained and therefore the coverage is also improved.

V. Experimental Result

This paper is implemented for simplifying CFGs to reduce complexity and to improve coverage of java program of *ntdrivers-simplified* category of *SV-COMP 2013* benchmark dataset. Firstly, these are C programs and they are converted into java by using a tool, C++ to Java converter. And then the corresponding CFGs are generated. To generate SETree and PSS-CFG, *Dr.Garbage* tool is used in Eclipse platform. According to the GENPSSCFG Algorithm, *Indus Kaveri* slicing tool provides the transformed program with the appropriate criteria [12]. The comparing results of complexity of original and simplified CFGs of a program are as shown in Table 1. These values are obtained by visualizing and analyzing the source code with the help of *Sci Tool*, *Understand*. The final coverage testing of original input program and transformed program is improved with *EclEmma* coverage tool after simplifying the CFGs as shown in Table 2.

The experimental results of CFGs from the motivating example as described in section 4 are also shown in Fig. 11. These are the output of Dr.Garbage in Eclipse. The Figure 11(a) represents the CFG of original input program (Fig. 6) and Figure 11(b) is the PSS-CFG of transformed program after simplifying.

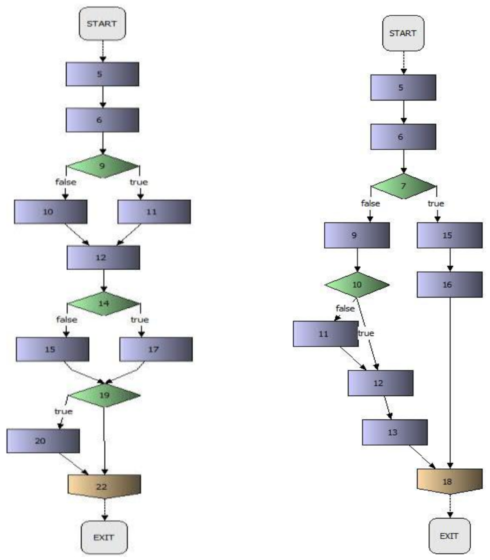


Fig. 11 CFG and PSS-CFG of example program before and after simplifying

TABLE I. Comparing Cyclomatic Complexity of Original Input Program and Sliced Transformed Program

	Cyclomatic Complexity Value	
	Original Input Program (Before Simplifying)	***Sliced Transformed Program (After Simplifying)***
kbfiltr	31	4
diskperf	25	17
ssh server	101	11
ssh client	90	2
floppy	35	15
cdaudio	53	22

TABLE II. Comparing Code Coverage of Original Input Program and Sliced Transformed Program

	Code Coverage Value	
	Original Input Program (Before Simplifying)	***Sliced Transformed Program (After Simplifying)***
kbfiltr	38.1%	52.4%

	Code Coverage Value	
	Original Input Program (Before Simplifying)	***Sliced Transformed Program (After Simplifying)***
diskperf	16.8%	58.9%
ssh server	23.7%	59%
ssh client	33.1%	85%
floppy	14.8%	51.2%
cdaudio	12.6%	69.2%

VI. CONCLUSION

In control flow testing, it is important to determine the control structure and flow path. Therefore the control flow graph is essential part of this technique. But, CFGs may be complex and it leads to increase the complexity and reduce coverage performance because the program internal structure may be complex with infeasible branches. The difficulty of infeasible branch conditions and the weakness of estimation that derives without possibility of identifying all infeasible branches limit the effectiveness of various approaches. Therefore, this paper focuses on program slicing to simplify the CFGs for more effective in improving performance of measuring coverage and reduce the complexity. Even if the merging will take place in program slicing, the CFG isomorphism will take place and it can still reduce the complexity. By comparing the original program and PSS-CFG before and after simplifying, this system ensures that the coverage performance is more improved and the complexity is reduced significantly.

REFERENCES

[1] S. Aditham, N. Ranganathan, "A Novel Control-flow based Intrusion Detection Technique for Big Data Systems." arXiv preprint arXiv:1611.07649, 2016 - arxiv.org

[2] Amighi, Afshin, et al., "Provably correct control flow graphs from Javabytecode programs with exceptions." International Journal on SoftwareTools for Technology Transfer (2015): 1-32.

[3] R. Gold, "Control Flow Graphs and Code Coverage", Int. J. Appl. Math. Comput. Sci., 2010, Vol. 20, No. 4, pp.739–749. DOI: 10.2478/v10006-010-0056-9

[4] V. Elodie, "White box coverage and control flow graphs", June 21, 2011.

[5] V. Ranganath and J. Hatcliff, "Slicing concurrent java programs using indus and kaveri", Inernational Journal on Software Tools for Technology Transfer.

[6] H. Watson, J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric", NIST Special Publication 500-235, September,1996.

[7] J. Jaffar, V. Murali, "A Path-Sensitively Sliced Control Flow Graph", *FSE* '14, November 16-22, 2014, San Hong Kong, China. Copyright 2014 ACM.

[8] J. Arora, "Static Program Slicing- An Efficient Approach for Prioritization of Test Cases for Regression Testing', International Journal of Computer Applications (0975 – 8887) Volume 135 – No.13, February 2016, pp.18-23.

[9] Y. Dubey, D. Singh, and A. Singh, "Amalgamation of Automated Test Case Generation Techniques with Data Mining Techniques: A Survey", International Journal of Computer Applications (0975 – 8887) Volume 134 – No.5, January 2016.

[10] I. Papadhopulli and E. Mece, "Coverage Criteria for Search Based Automatic Unit Testing of Java Programs", International Journal of Computer Science and Software Engineering (IJCSSE), Volume 4, Issue 10, October 2015.

[11] K. Satyam, "Enhancement of branch coverage using java program code transformer", National Institute of Technology Rourkela, Orissa, India. May, 2015.

[12] G. Jayaraman, V. Ranganath, and J. Hatcliff, "Kaveri: delivering the indus java program slicer to eclipse".

[13] M. Myitzu Aung, K. Thi Win, "Improving Branch Coverage for White-box Testing", 27th International Conference on Computer Theory and Applications (ICCTA 2017).

[14] Adekola, O.D, Idowu, S.A, Okolie, S.O, Joshua, J.V, Akinsanya, A.O, Eze, M.O, EbiesuwaSeun "Software Maintainability and Reusability using Cohesion Metrics". International Journal of Computer Trends and Technology (IJCTT) V54(2):63-73, December 2017. ISSN:2231-2803. www.ijcttjournal.org. Published by Seventh Sense Research Group.