American Journal of Software Engineering and Applications
2014; 3(6): 95-101
Published online December 23, 2014 (http://www.sciencepublishinggroup.com/j/ajsea)
doi: 10.11648/j.ajsea.20140306.15
ISSN: 2327-2473 (Print); ISSN: 2327-249X (Online)

An empirical study on the effectiveness of automated test case generation techniques

Bolanle F. Oladejo, Dimple T. Ogunbiyi

Department of Computer Science, University of Ibadan, Ibadan, Nigeria

Email address:
fb.oladejo@ui.edu.ng (B. F. Oladejo), ogunbiyidimple@gmail.com (D. T. Ogunbiyi)

To cite this article:
Bolanle F. Oladejo, Dimple T. Ogunbiyi. An Empirical Study on the Effectiveness of Automated Test Case Generation Techniques. *American Journal of Software Engineering and Applications.* Vol. 3, No. 6, 2014, pp. 95-101. doi: 10.11648/j.ajsea.20140306.15

Abstract: The advent of automated test case generation has helped to reduce the laborious task of generating test cases manually and is prominent in the software testing field of research and as a result, several techniques have been developed to aid the generation of test cases automatically. However, some major currently used automated test case generation techniques have not been empirically evaluated to ascertain their performances as many assumptions on technique performances are based on theoretical deductions. In this paper, we perform experiment on two major automated test case generation techniques (Concolic test case generation technique and the Combinatorial test case generation technique) and evaluate based on selected metrics (number of test cases generated, complexities of the selected programs, the percentage of test coverage and performance score). The results from the experiment show that the Combinatorial technique performed better than the Concolic technique. Hence, the Combinatorial test case generation technique was found to be more effective than the Concolic test case generation technique based on the selected metrics.

Keywords: Automated Test Case Generation Technique, Combinatorial, Concolic, Empirical Study, Software Testing, Software Metrics

1. Introduction

Software testing plays a very significant role in the software development process and serves as an important way to measure and improve software quality with its main intent on detecting defects in software. Software testing has been defined as a process of verifying and validating that a software program meets its business and technical requirements that guides its design and development and works as expected [2] therefore, it is a very important means of assessing software to determine its quality [11]. It is also heavily used to initiate, locate and remove software defects [13]. Software testing can be broken down into three fundamental procedures; design (generation) of test cases, execution of test cases and checking whether the output produced is correct based on the input given [10].

Test case generation is a process of creating or identifying test data which can satisfy a given testing criterion [8]. Test case generation is among the most labour-intensive tasks in software testing and its manual approach can take very long time to generate and execute test cases. Automated test case generation came into place to reduce the work load of testers [1] with the intent of generating quality test cases to execute programs. In recent years, several techniques have been developed to enhance automated test case generation and it is important for testers/researchers to be conversant with current approaches to generating test cases automatically. Furthermore, it is also important to perform experiments on automated test case generation techniques in order to appraise their performances. Reference [15] encouraged researchers to carry out repeated experiments on tools and techniques in order to give the software testers knowledge on their strengths, weaknesses, effectiveness and functionalities. Hence, this study is aimed at evaluating the effectiveness of two major automated test case generation techniques, with its objectives stated as; identifying which technique achieves the highest test coverage; identifying the effect of program complexities on the test coverage of techniques and identifying which technique is most effective in general.

In this paper, we evaluate the Combinatorial and Concolic test case generation techniques. In addition, we compare and evaluate the techniques based on the number of test cases

generated, complexities of the selected programs, percentage of test coverage and performance score for achieving each of the stated objectives. The results from the experiment shows that Combinatorial technique achieved a higher test coverage than the Concolic technique for the programs used. The results also show that the complexities of the programs used does not affect the Combinatorial technique in achieving high test coverage but affects the Concolic technique in achieving high test coverage.

The remaining part of this paper will present a brief description of the selected techniques in section 2, the related studies in section 3, the experimental procedure in section 4, the results in section 5 and the conclusion in section 6.

2. Description of Selected Techniques

This section presents a brief description of the operations of the techniques selected for evaluation. The techniques are; Concolic and Combinatorial techniques. The automated test case generation tools (referred to as test case generators) for each technique, were selected based on their features and functions. The techniques/tools are described in the subsections below.

2.1. Concolic Technique

The Concolic technique is a hybrid technique that combines Concrete execution (executes program using concrete inputs) with Symbolic execution (executes program using symbolic inputs). Concolic testing performs symbolic execution of a program along a concrete execution path. It executes a program starting with some specified or random concrete input and gathers symbolic constraints on inputs at conditional statements during the execution caused by the concrete input then it uses a constraint solver to create variants of the concrete input for the next execution of the program. This process will be repeated until all feasible execution paths are explored or a user-defined coverage criterion is met [16].

The Concolic test case generator used for the experiment is a publicly available tool named LIME Concolic Tester [7] and is available at http://www.tcs.hut.fi/Software/lime/.

2.2. Combinatorial (Pairwise) Technique

The Combinatorial technique generates test cases for a combination of parameters for programs. It places special emphasis on selecting a sample of input parameters covering a recommended subset of combinations of elements to be tested. For each pair of input parameters it will test all possible discrete combinations of those parameters, using chosen test vectors [9]. Pairwise testing is a prominent combinatorial strategy that reduces the number of test cases created. Pairwise testing strategy is defined as: Given a set of N independent test factors: f_1, f_2, ..., f_N, with each factor f_i having L_i possible levels: $f_i = \{l_{i,1}, ..., l_i, L_i\}$, a set of tests R is produced. Each test in R contains N test levels, one for each test factor f_i, and collectively all tests in R cover all possible pairs of test factor levels i.e. for each pair of factor levels $l_{i,p}$ and $l_{j,q}$, where $1 \leq p \leq L_i$, $1 \leq q \leq L_j$ and $i \neq j$ there exists at least one test in R that contains both $l_{i,p}$ and $l_{j,q}$ [3]. The Combinatorial test case generator used for the experiment is also a publicly available tool named Test Case Generator, developed by Bulmahn M. in 2007 and is available at http://www.download.microsoft.com/download\.

3. Related Studies

Several experimental studies have been carried out on various automated test case generation techniques. This section presents the methods and results of some similar work that have been carried out on automated test case generation. Reference [6] conducted experiment on four test data generation techniques (Random technique, IRM based Method, Korel method and GA based method). The results of the experiment show that the genetic algorithm (GA)-based test data generation performs the best. Reference [5] carried out an experiment comparing a total of 49 subjects split between writing tests manually and writing tests with the aid of an automated unit test generation tool, EVOSUITE. The purpose of this study was to investigate how the use of an automatic test generation tool, when used by testers, impacts the testing process compared to traditional manual testing. Their results indicated that while the use of automated test generation tools can improve structural coverage over manual testing, it does not appear to improve the ability of testers to detect current or future regression faults. Reference [7] compared the effectiveness of Concolic testing and random testing. The experiment shows that Concolic testing is able to find significantly more bugs than random testing in the testing domain. Reference [4] presented an empirical comparison of automated generation and classification techniques for object oriented unit testing. Pairs of test-generation techniques based on random generation or symbolic execution and test-classification techniques based on uncaught exceptions or operational models were compared. Their findings show that the techniques are complementary in revealing faults. Some other experimental studies conducted are on the evaluation of tools [17], [14].

This study extends existing empirical studies by testing the effectiveness of two techniques that have been widely used over the years for test case generation and test coverage improvement. We present in this study an experimental structure describing the activities involved in evaluating the techniques, this can also serve as a framework for further experiments or can be advanced.

4. Experimental Procedure

This section presents the methods and procedures used for the experiment. It covers the programs selected, experimental processes and the metrics used for evaluation. In addition, we present an experimental structure that simplifies the description of the experimental procedure used in evaluating the test case generation techniques selected as shown in “Fig. 1”.

We used three different java programs (test objects) based on some features such as; arrays, loops, branching statements method calls and complexity measure of the programs. The complexity of each program was measured using the cyclomatic complexity metric (see section 4.1.1.) and the test case generators were applied on the programs.

The Concolic test case generator was installed and executed on a Linux ubuntu environment. It was applied on the three programs. Each program was passed as input to the test case generator and the resulting test cases were generated and coverage measured automatically. However, during the course of carrying out the experiment, it was discovered that a limitation of the Concolic test case generator used is that it does not accept string parameters. Therefore, the test case generator was applied on only two of the programs out of the three selected programs as one of the programs accepts string inputs only.

"Fig. 2" shows a snapshot of the Concolic test case generator environment.

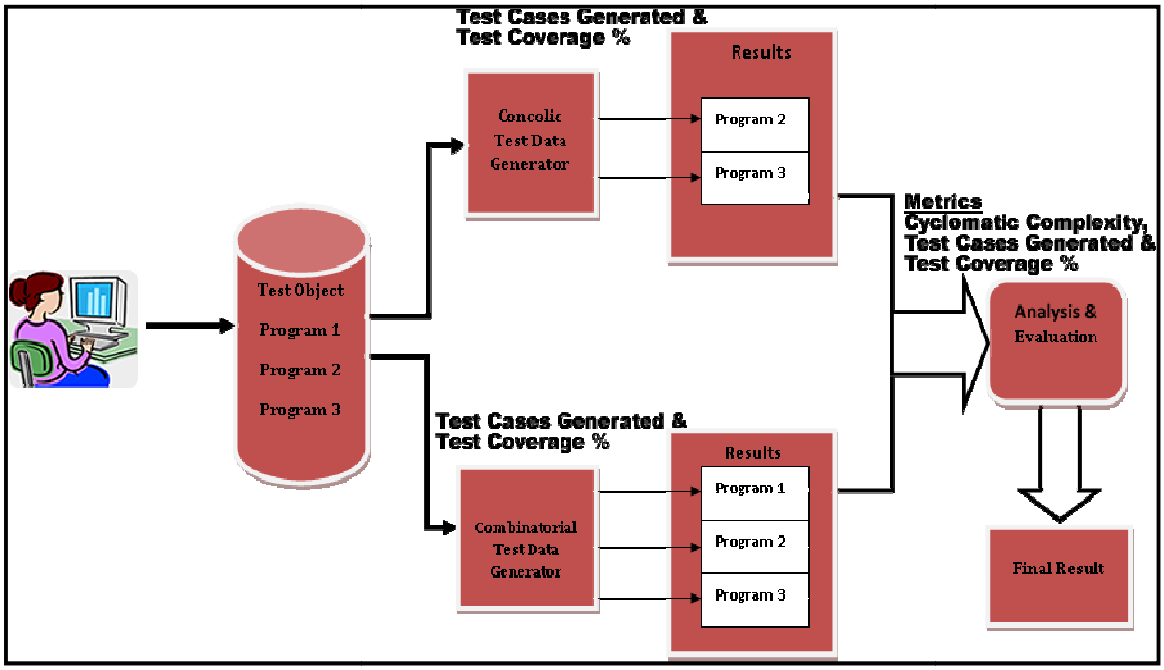

Figure 1. *Experimental Structure for the Evaluation of Selected Techniques.*

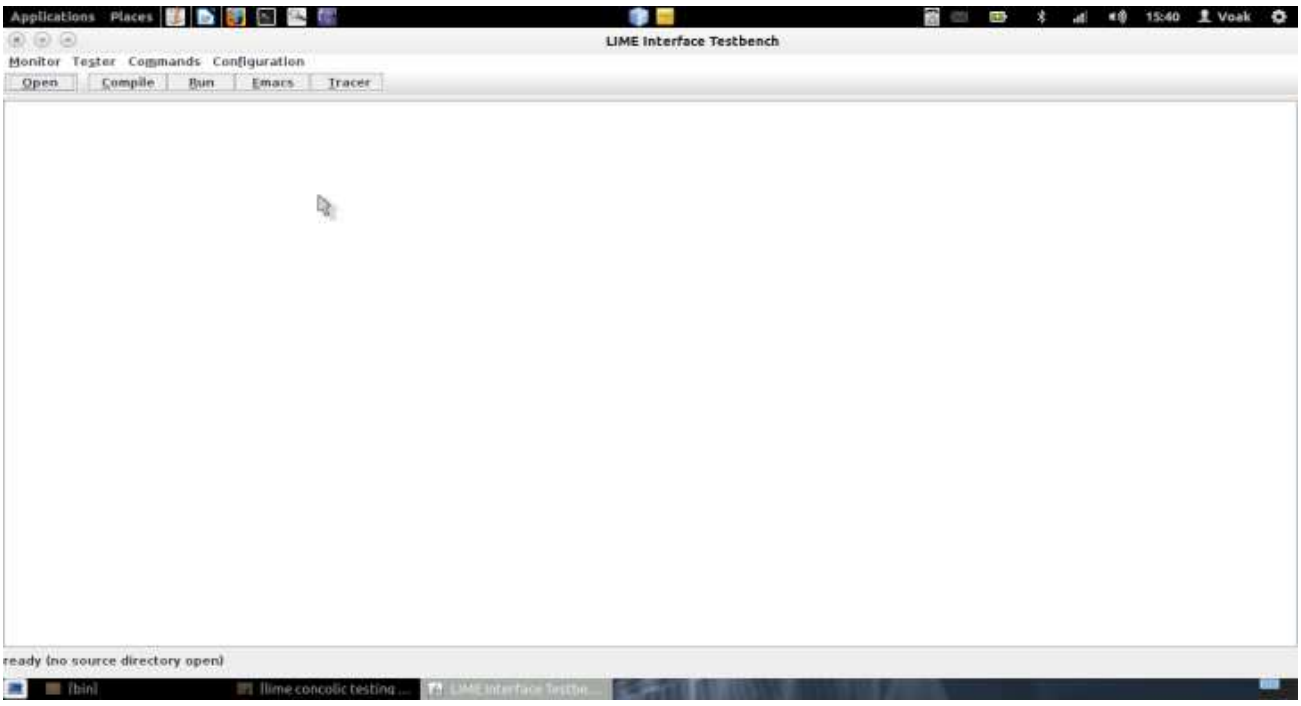

Figure 2. *The Concolic Test Case Generator Environment.*

The Combinatorial test case generator was installed and executed on a windows 7 environment. It was applied on three of the programs for individual results and applied on two of the programs for the compare results. The test cases were generated automatically from a list of user specified parameters, expected outcomes and rules for each of the programs. A combination depth of two (2) was selected because this study considers pairwise combinatorial strategy and the test coverage was determined from the test cases generated. “Fig. 3” shows a snapshot of the Combinatorial test case generator environment.

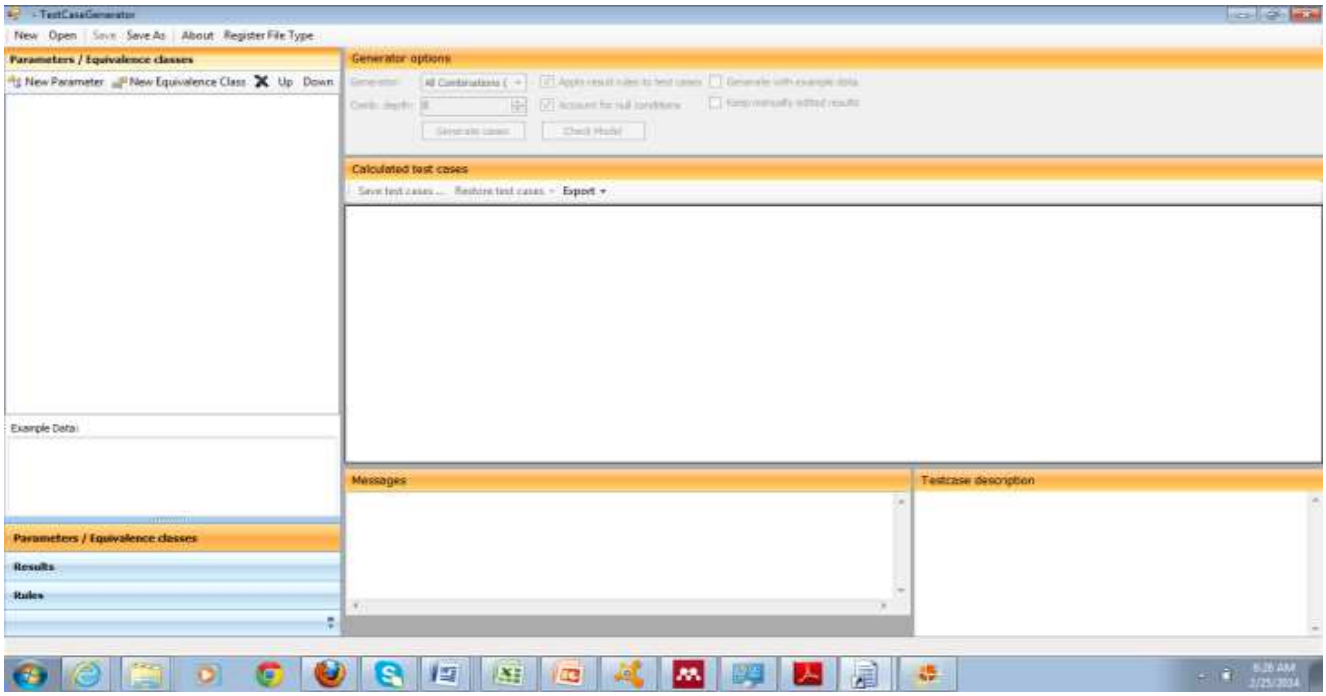

Figure 3. *The Combinatorial Test Case Generator Environment.*

4.1. Selected Metrics

This section presents the metrics used for comparison of the techniques. They were selected amongst other metrics in order to achieve the stated objectives of this study. Their descriptions are given in the following subsections:

4.1.1. Cyclomatic Complexity

We used the cyclomatic complexity metric to measure the complexity of each program used in this study. This complexity metric was selected because it quantitatively measures the logical capability of a program. The cyclomatic complexity was calculated from each program’s control flow graph. A control flow graph shows the flow of control of statements and decisions in a program. It consists of nodes (used to represent statements and decisions in a program) and edges. The complexity of each of the programs was measured using the McCabe’s cyclomatic complexity formula [12].

The formula is given as;

$$v(G)= E -N+ 2P$$

where:

v(G) = Cyclomatic Complexity

E = The number of edges of the graph

N = The number of nodes of the graph

P = The number of connected components

Table 1 shows the complexity values for the three programs with the range from a low complexity value to a high complexity value.

Table 1. *Cyclomatic Complexity Value for Selected Programs.*

Program	Cyclomatic Complexity
Program 1	3
Program 2	5
Program 3	25

The complexity values of the programs as shown in the table above ranges from the lowest complexity to the highest complexity. Some researchers have deduced that the complexity value of a program above ten (10) has a very high complexity. Furthermore, Reference [14], categorized the cyclomatic complexity value range of programs into three parts which include; LOW (complexity value range is 1- 4), MID (complexity value range is 5-10) and HIGH (complexity value range is above 10).

We used the cyclomatic complexity metric to test if the complexities of the programs would affect the test coverage of the automated test case generation techniques. An assumption is that the techniques should be able to achieve high coverage even with complex programs to prove that it is really effective.

4.1.2. Number of Test Cases Generated

The number of test cases generated for each program was gotten from the test case generators.

4.1.3. Test Coverage

We used the test coverage metric to determine the degree to which the programs have been executed by the test cases generated. The branch coverage criterion was determined for the Concolic technique while the state space coverage criterion was determined for the Combinatorial technique [9]. The average percentage of test coverage by each test case generator was calculated and their performances were compared.

4.2. Threats to Validity

Our initial intent was to apply the test case generators on the three selected java programs partially because of the complexity range of the programs but in the comparison phase, the techniques were applied on only two of the three selected programs because of the limitation of the Concolic test case generator stated earlier. Furthermore, the test case generators were chosen amongst others because they meet our hardware requirements and program construct specifications. However, we believe that if the test case generators were applied on the third program, the result would still be the same or would be very similar to the present results. Also, if the third program was used for comparison of the techniques, the Combinatorial technique would have achieved an average test coverage of 89% which is still very reasonable and still makes it effective.

5. Results

The previous section gave a description of the methods and experimental procedures used in this study. This section presents and discusses the results gotten from the experiment performed on the automated test case generation techniques. We present the individual results for the techniques and the compared results. The compared results are presented based on the objectives of this study.

5.1. Individual Results Generated

Table 2 shows the individual results gotten for the Concolic and Combinatorial techniques. It includes the program names, the cyclomatic complexity value for each program, the number of test cases generated and the percentage of test coverage for each of the techniques.

Table 2. *Individual results for the two Techniques.*

Program	Cyclomatic Complexity	Concolic Technique		Combinatorial Technique	
		No. of Test Cases Generated	Test Coverage (%)	No. of Test Cases Generated	Test Coverage (%)
Program 1	3	-	-	18	67
Program 2	5	1	50	3	100
Program 3	25	400	41	8764	100
Total/Average Test Coverage	33	401	46	8785	89

The Concolic test case generator was applied on program 2 and Program 3 and generated a total number of four hundred and one (401) test cases and an average test coverage of forty six percent (46%) while the Combinatorial test case generator was applied on the three programs generating a total of eight thousand, seven hundred and eighty five (8785) test cases and an average test coverage of eighty nine percent (89%).

5.2. Compared Results

Only Program 2 and Program 3 were used for the comparison of the techniques. The results are presented based on the objectives of this study as follows.

Objective 1: Identify which technique achieves the highest test coverage

The test coverage is highly important in evaluating the techniques. A test case generation technique which achieves test coverage of 100% means that it has generated test cases which explored all the feasible paths of a program but does not mean that the program is free from defects. Table 3 and "Fig. 4", shows the results of the test coverage for the techniques.

Table 3. *Comparison of Test Coverage of Techniques.*

Program	Concolic Test Coverage (%)	Combinatorial Test Coverage (%)
Program 2	50	100
Program 3	41	100
Average Test Coverage	46	100

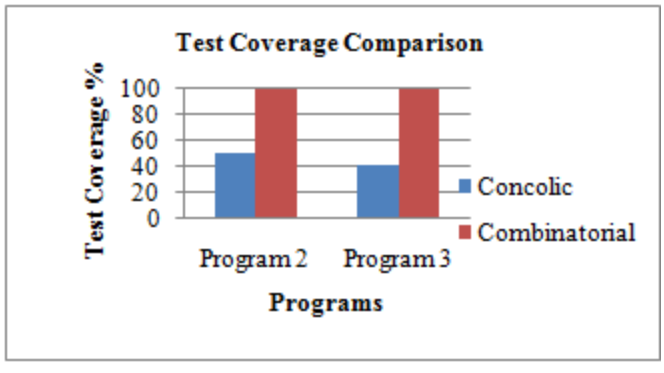

Figure 4. *Comparison of Test Coverage for Techniques.*

The chart above shows that the Combinatorial technique performs better than the Concolic technique in achieving a high test coverage because for the two programs used, it achieved a test coverage of 100% each. The Concolic technique achieved a lower coverage for the two programs compared to the Combinatorial technique. Hence, it can be inferred that the Combinatorial test case generation technique performs better than the Concolic test case generation technique in achieving high test coverage.

Objective 2: Identify the effect of program complexities on the test coverage of techniques

Knowing if the complexities of programs will affect the test coverage of techniques is a very essential factor to measure their effectiveness. For *n* programs with different levels of complexities, the techniques should be able to achieve a substantial amount of test coverage to prove their strength. The result for the comparison of the effect of program complexities on techniques is presented in Table 4 and a chart describing the comparison is shown in "Fig. 5".

Table 4. *Complexity Effect Comparison.*

Program	Cyclomatic Complexity	Concolic Test Coverage (%)	Combinatorial Test Coverage (%)
Program 2	5	50	100
Program 3	25	41	100
Total/Average Test Coverage	30	46	100

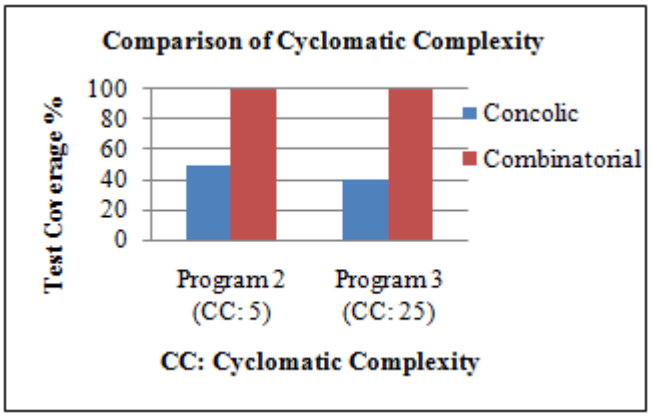

Figure 5. *The Effect of Program Complexities on both Techniques.*

The chart above compares the Concolic and the Combinatorial technique based on the effect of the complexity of two programs on the percentage of test coverage achieved. From the chart it can be stated that the Combinatorial technique performs better than the Concolic technique in achieving high test coverage for complex programs.

Objective 3: Identify which technique is most effective in general

We identified the technique that is most effective in general. By the word 'general' we mean the technique that performs best in meeting the objectives presented previously i.e. the technique that achieves the highest test coverage and the technique that program complexities have little or no effect on. We allocated a score to each objective and the total score was given as three (3), one for each objective. The Combinatorial technique scored the highest value of 3 because it performed better in all the objectives than the Concolic technique. The Concolic technique scored 1 for attaining at least level of test coverage. Table 4 shows the summary of the evaluation and it includes; the overall result for each technique based on the three metrics used in the study and the score for each technique.

Table 5. *Summary of Evaluation.*

Technique	Complexity Effect	Total Number of Test Cases Generated	Average Test Coverage (%)	Score
Concolic	Has Effect	401	46	1
Combinatorial	No Effect	8767	100	3

From the table above, it is possible that if the number of test cases generated by the Concolic test case generator increases then the average test coverage achieved would increase.

6. Conclusion

We have been able to evaluate two major automated test case generation techniques (Concolic and Combinatorial Techniques) through experiment. The results from the experiment show that the Combinatorial test case generation technique performed better than the Concolic test case generation technique and is thereby a more effective technique based on the evaluation criteria used. Hence, future works should be directed towards conducting further empirical studies on the Combinatorial technique with other major techniques and a large number of programs to validate its effectiveness.

References

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey on automated Software test case generation," Antonia Bertolino, J. Jenny Li and Hong Zhu, Editor/Orchestrators, Journal of Systems and Software 2013.

[2] J. E. Bentley, "Software testing fundamentals-concepts, roles, and terminology," Corporate Data Management and Governance, Wachovia Bank, 201 S. College Street, NC-1025, Charlotte NC 28210, 2001.

[3] J. Czerwonka, "Pairwise testing in real World: practical extensions to test case generators," Microsoft Corporation, One Microsoft way Redmond, WA 98052, 2006.

[4] M. d'Amorim, C. Pacheco, T. Xie, D. Marinov, and M. D. Ernst, "An empirical comparison of automated generation and classification techniques for object-oriented unit testing," Department of Computer Science, University of Illinois, Urbana-Champaign, IL, U.S.A., 2006.

[5] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and P. Padberg, "Does automated White-Box test generation really help Software Testers,?" Department of Computer Science, University of Sheffield, United Kingdom, 2013.

[6] S. Han and Y. Kwon, "An empirical evaluation of test data generation techniques." Journal of Computing Science and Engineering, vol. 2, No. 3, September, 2008.

[7] K. Kahkonen, R. Kindermann, K. Heljanko and I. Niemela, "Experimental comparison of Concolic and Random Testing for Java Card Applets," Department of Information and Computer Science Aalto University, P.O. Box 15400, FI-00076 AALTO, Finland, 2010.

[8] B. Korel, "Automated Software test data generation," IEEE Transactions on Software Engineering, Vol. 16, No. 8, 1990.

[9] D. R. Kuhn, R. N. Kacker, and Y. Lei, "Practical Combinatorial Testing,". National Institute of Standards and Technology (NIST), U.S. Government Printing Office, Washington, U.S.A., 2010.

[10] K. Lakhotia, P. McMinn, and M. Harman, "Automated test data generation for coverage: haven't we solved this problem yet?," King's College, CREST centre, London,WC2R 2LS, U.K., 2009.

[11] L. Luo, "Software Testing Techniques," Institute for Software Research International, Carnegie Mellon University, Pittsburgh, PA15232, U.S.A., 2001.

[12] T. J. McCabe, "A complexity measure," IEEE Transactions on Software Engineering, Vol. Se-2, No., 4, 1976.

[13] J. Pan, "Software Testing, Dependable Embedded Systems," Electrical and Computer Engineering Department, Carnegie Mellon University, 1999.

[14] X. Qu, and B. Robinson, "A case study of Concolic Testing tools and their limitations," ABB Corporate Research 940 main campus drive, Raleigh, NC, U.S.A., 2010.

[15] M. Roper, J. Miller, A. Brooks, and M. Wood, "Towards the experimental evaluation of Software testing techniques," EuroSTAR '94, pp 44/1-44/10October 10-13, 1994, Brussels.

[16] K. Sen, "Concolic testing and constraint satisfaction," Proceedings, 14th International Conference on Theory and Applications of Satisfiability Testing (SAT'11), 2011.

[17] S. Wang, and J. Offutt, "Comparison of unit-level automated test generation tools," Software Engineering, George Mason University, Fairfax, VA 22030, USA, 2008.