# Test Case Generation Using Java Path Finder and Symbolic Path Finder

Arnaldo Marulitua Sinaga and Rosni Lumbantoruan

***Abstract*— Test suite plays an important role in software testing. Good test cases may improve the quality of software by detecting failures earlier. Automation test case generation can help to generate good test cases is needed. Symbolic Path Finder (SPF) is a model checker developed by the National Aeronautics and Space Administration (NASA) as an extension of Java Path Finder (JPF). SPF uses symbolic execution, model checking and constraint solver to produce test cases. This study is intended to perform SPF and to verify the effectiveness of the test case produced by the SPF. Mutation testing, a fault-based testing technique, is conducted to investigate the adequacy of produced test cases. The programs under test in this study are Median, Armstrong, Multiple, Nested If, and Simple Calculator. All those programs are written in the Java programming language. The test suite for each of the programs under test has been generated by applying SPF. The mutation score obtained for produced test suites of each program ae as follows: 74.82% for the Median Program, 77.78% for the Armstrong Program, 31.20% for the Multiple Program, 92.00% for the Nested If Program, and 55.82% for the Simple Calculator Program. These experimental results indicate that the test suite generated by the SPF is unable to detect all existing faults. From the observations, this problem is because SPF only applies decision coverage in determining path during the process of forming test cases. Potential improvement may be obtained by applying other types of coverage criteria.**

***Index Terms*— *Test case generation, Java Path Finder, Symbolic Path Finder, Mutation testing***

## I. INTRODUCTION

SOFTWARE testing is the process of identifying the accuracy, completeness, and quality of the built software. Software testing is part of software engineering that includes validation of each stage of the Software Development Life Cycle (SDLC) [1]. The purpose of software testing is to detect failures [2][3]. Tests are performed using test suites which are a set of test cases. Test cases consist of an assembly of observed inputs and outputs that will then compare the actual output with the expected output [4].

A test case can be generated manually or automatically. Generating a test case manually depends on the ability of the tester to locate a failure after the program is executed [4]. While generating a test case automatically uses a tool that can generate a more effective test case and can cover the entire program code [5], a test case is required to cover the entire code of the program in order to optimize the failure detection. The more program code covered, the more likely it is to find a fault [6][7][8].

One of the tools used to generate test cases automatically is the Symbolic Path Finder (SPF). SPF is an extension of the Java PathFinder (JPF) for symbolic execution. JPF is a checker model developed by the National Aeronautics and Space Administration (NASA) [9][10]. JPF executes Java programs and searches for all possible paths that can be executed on programs as well as verifies their authenticity, such as deadlocks and exceptions. SPF is a combination of symbolic execution, model checking, and constraint solving to generate test cases automatically. SPF uses an analysis engine from JPF, which is a checking model to examine the internal code structure of the program to find errors. After checking all possible paths, the SPF uses a constraint solver to generate input test cases according to the constraints of the path passed. The resulting output contains information about the resulting parameter input where each path on the program code has been executed [11].

This research plays a role in experimentally investigating the use of SPF in software testing. To analyze the effectiveness of using SPF in generating test cases, mutation testing is performed. Mutation testing modifies the original program by inserting a fault into the program and then performs a process of testing to evaluate the ability of the test case and find that fault. The code of the program containing the fault is called a mutant program. The results of the testing process for the mutant program are called actual outputs, whereas the results of testing of the original program are called expected output.

This paper consists of six sections. Section 2 explaining the studied methods, namely Java Path Finder and Symbolic Path Finder. Section 3 explains the conducted experiment. Section 4 explains the results obtained from the experiment. Section 5 explains the discussion of the obtained results. Section 6 explains the conclusions and potential further research.

## II. STUDIED AUTOMATION TESTING METHODS

### *A. Automation Testing*

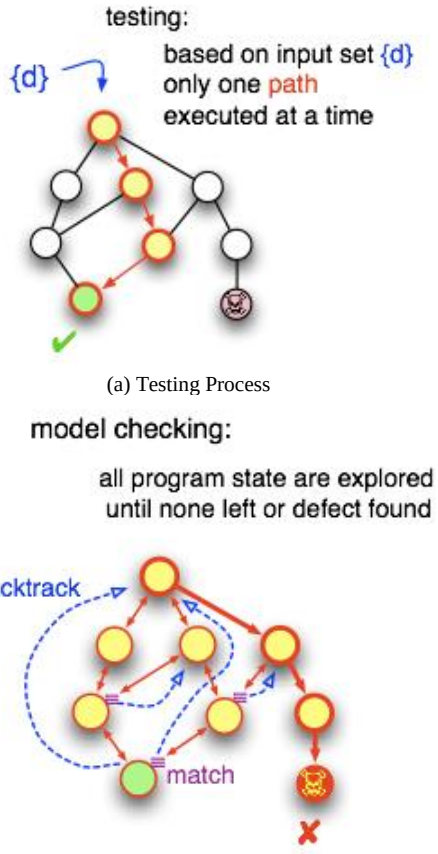
Automation testing is the process of writing and executing test cases using software. Automation testing can be done quickly and repeatedly. The software used to carry out automation testing in this research is Symbolic Path Finder (SPF) which is an extension of Java Path Finder (JPF).

A. M. Sinaga is with Faculty of Vocational Studies at Institut Teknologi Del (IT Del), Indonesia (e-mail: aldo@del.ac.id*).

R. Lumbantoruan is with Faculty of Informatics and Electric Engineering at Institut Teknologi Del (IT Del), Indonesia (e-mail: rosni@del.ac.id).

### B. *Java Path Finder*

The Java Path Finder (JPF) is a model checker developed by the National Aeronautics and Space Administration (NASA). JPF is an open-source research platform. In JPF, a Java program is given to be executed, and then a JPF file (with extension of .jpf) is generated to verify the program in order to detect failure in the program. JPF explores all paths in the program. When faced with the branch, the JPF checks if it has been through the same path or not. If yes, then it will go back to a previous point that it has never been through, called a backtrack. JPF is a model checker that distinguishes itself from testing in general [12]. The differences of JPF with general testing methods can be seen in Fig. 1 [10].




(a) Testing Process

(b) Model Checking Process

Fig. 1. JPF testing methods (a) testing process, (b) model checking process (Source: https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/testing_vs_model_checking)

According to Fig. 1, testing will execute a program based on the given input in which testing only executes one path on a program's control flow model at a time. This is called a concrete execution. Model checking explores all possible paths on the control flow models of a program in a backtrack until no more errors are found or called symbolic executions. It is performed to find more errors on the test program.

The input given to JPF is a class file (Java bytecode) of a test program and a configuration file to determine the execution mode that JPF performs to the program as well as the properties of the artifact needing to be produced. To generate the desired artifacts, JPF provides several usable extensions. The commonly used extensions of JPF are as follows [12]:

- Choice Generators are used to generate choices or choices on each branch formed by a test program.
- Instruction Factories are a set of semantic instructions consisting of sets of operations such as calling methods or variable access processes performed on test programs.
- Attribute Objects are metadata related to values and concrete objects. It means that JPF provides a mechanism to store operand values, local variable values, and metadata of a test program. This extension allows JPF to do a backtrack because when JPF performs a backtrack, the condition or information of the last attribute passed by JPF will be stored.
- Native Peers is an abstraction library that supports JPF executing programs on JVM.
- Listeners is the execution monitoring that JPF uses as a plugin for monitoring internal operations in JPF.

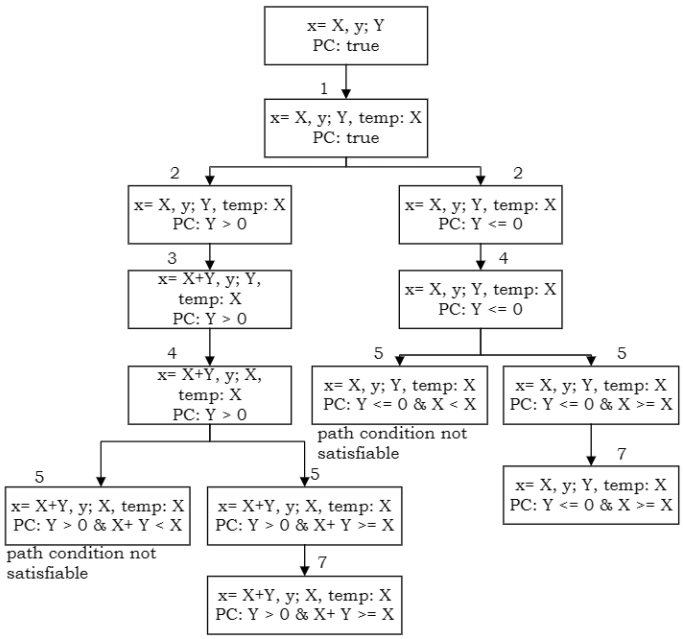
### C. *Symbolic Path Finder*

Symbolic Path Finder (SPF) is an open-source automatic test case generation tool [11][13]. SPF can handle input and operations from Booleans, integers, reals and complex data structures with a polymorphic class hierarchy [14]. SPF uses the analysis engine from JPF to examine the internal structure of program code to find errors. The output of the program code represents the numerical constraints of the input parameters in the program code. Then these constraints are used to produce concrete test cases as input parameters of the program code [11]. SPF not only uses symbolic execution mode, but SPF also supports combined execution modes, namely concrete execution and symbolic execution. The input parameters used for concrete execution are input parameters with concrete values that are given fixed values, for example constant values, and the input parameters used for symbolic execution are input parameters whose symbolic values correspond to the conditions of the path traversed [14].

Symbolic execution is a program analysis technique where input variables that have concrete values are replaced into symbolic variables [12][15]. The principle is that when determining a path that contains the symbolic value of all executed paths, the paths that have been traversed will be saved. In symbolic execution, Path Condition (PC) is known as a constraint condition for input on a branch, so that the PC is always updated according to the branch being traversed. The path traversed during symbolic execution is represented in the form of a symbolic execution tree [12]. An example of a simple program to form a symbolic execution tree from a program executed is using the symbolic execution technique illustrated in Fig. 2.

```
   int example(int x, int y) {
1:   int temp = x;
2:   if (y > 0)
3:     x = x + y;
4:   y = temp;
5:   if (x < y)
6:     assert (false);
7:   return x + y;
   }
```

(a) Code example for symbolic execution




(b) Symbolic execution of program

Fig. 2. Symbolic execution tree for example code (a) code example, (b) symbolic execution of program (Source: Evaluation of Java PathFinder Symbolic Execution Extension)

In Fig. 2(b) is a symbolic execution tree formed from the program in part a. First, the int x and int y parameter values are converted into symbolic values x to X and y to Y with the PC value true. Then the x value is given to the temp variable and the PC remains true. Then the branch is identified so that two branches are formed which are the value of the path condition and the negation of that path condition. On line 2, the specified PC condition is (Y > 0) then the negative (Y <= 0). After the PC (Y> 0) is passed, the line 3 is run (X = X + Y) and continues with (Y = temp) which means that Y becomes X. On line 3 and 4 the PC remains (Y > 0). The value generated at (X+Y < X) does not intersect, so the path is categorized as an infeasible path and no test case is generated on the path. After all path passes, the symbolic execution backtracks until the negation of the path on line 2 is PC value (Y <= 0). When executing this PC, line 3 is not executed so that X remains X value. Then on line 4, Y becomes X and on line 5, the same branch as the previous path is identified. The first path of the branch is also called an infeasible path [15].

## III. THE EXPERIMENTS

### A. Program Under Test

The methodology in this study is an experimental method that is conducted to investigate the SPF tool in producing a test case. The study used five research objects written using the Java programming language and integer parameter input, namely Median Program, Armstrong Program, Multiple Program, Nested If Program, and Simple Calculator Program. The five programs were obtained from different sources where Median program was downloaded from the Software-Artifact Infrastructure Repository (SIR) [16], Armstrong program was downloaded from GeeksforGeeks [17], and Multiple programs, nested if program, and simple calculator program were programs created by researchers for research purposes.

### B. Experiments Analysis

To be able to generate test cases, SPF requires a class file (Java bytecode) and a configuration file with the extension .jpf. This file contains options that are used to verify the program and enabling it to generate test cases from the program. Some commonly used options in a configuration file are as follows:

- classpath is used as a path to the directory containing the compilation files of the Java program.
- sourcepath is used as a path to the directory containing the Java program being executed.
- target is used to indicate the package name and Java program being executed.
- symbolic.method is used to show the parameters of a method that is executed symbolically.
- symbolic.min_int and symbolic.max_int are used to provide test case value limits.
- SymbolicListener is used to display information of the path condition that is executed symbolically.
- SymbolicSequenceListener is used to display test cases.
- search.multiple_errors is used to prevent SPF from hindering the execution of a Java program if it encounters the first error.

The symbolic execution tree produced by SPF is used to determine the number of test cases generated in a program. Symbolic execution trees are formed based on the code structure of the program being tested. In this research, the symbolic execution tree is drawn manually for programs with few lines of program code, contrasting with programs with many lines of program code in which it is impossible to draw the symbolic execution tree manually. As a result, verification of the number of test cases produced in a program is carried out by conducting trials using several limit values. Limit value experiments were carried out using the symbolic.min_int and symbolic.max_int options.

The symbolic.min_int and symbolic.max_int options are used for programs with integer input parameters. These two options are used to determine the limit value or range of values for the candidates of test cases that will be generated. The symbolic.min_int option is used to provide a minimum value limit for the test case and symbolic.max_int is used to

provide a maximum value limit for the test case. These two options are additional options provided by SPF. In the experiments carried out, the symbolic.min_int and symbolic.max_int options were given in each study object configuration file to determine the number of test cases produced. To determine the maximum and minimum limit values, a series of experiments were carried out.

Several experiments carried out on the Median Program can be seen in Table I. In the Median Program, experiments were carried out on 18 candidates within the limit value range and 15 candidates were obtained, producing six test cases as in Table I. Based on these experiments, the maximum number of test cases in the Median Program is six. The limit value range chosen in this experiment is a minimum value of -1 and a maximum value of 1. Verification of this program can also be done based on the program's symbolic execution tree.

TABLE I
LIMIT VALUE FOR MEDIAN PROGRAM

| Min Value | Max Value | Number of TC |
|---|---|---|
| -50 | 50 | 6 |
| -40 | 40 | 6 |
| -30 | 30 | 6 |
| -20 | 20 | 6 |
| -10 | 10 | 6 |
| -5 | 5 | 6 |
| -1 | 0 | 5 |
| -1 | 1 | 6 |
| 0 | 0 | 1 |
| 0 | 1 | 5 |
| 0 | 5 | 6 |
| 0 | 2 | 6 |
| 0 | 10 | 6 |
| 0 | 20 | 6 |
| 0 | 30 | 6 |
| 0 | 40 | 6 |
| 0 | 50 | 6 |
| - | - | 6 |

The symbolic execution tree of the Median Program is presented in Fig. 3. It can be seen that the paths of the Median Program consist of six paths and all of these paths can be traversed. These are called feasible paths. The six paths in the Median Program are as follows:

- PC1: Y < Z ∧ X < Y
- PC2: Y < Z ∧ X >= Y ∧ X < Z
- PC3: Y < Z ∧ X >= Y ∧ X >= Z
- PC4: Y >= Z ∧
- PC5: Y >= Z ∧ X <= Y ∧ X > Z
- PC6: Y >= Z ∧ X <= Y ∧ X <= Z

The same analysis process is conducted to all programs under the test. The Armstrong program is a program with repetition code to ensure that the number of test cases is dependent on the number of iterations carried out. In the Armstrong Program, experiments were conducted on 13 candidate limit value ranges, and it was found that three candidates produced invalid test cases. An invalid test case is a test case consisting of a value exceeding the maximum limit of the integer value type. This test case is inoperable because it causes the program to error. Limit values that produced invalid test cases were eliminated from the selected limit value candidates because the test cases could not be used, 10 more candidates to be selected. The range of limit values chosen for use in experiments is a minimum value of 0 and a maximum value of 1000. Verification of this program can also be done based on the program's symbolic execution tree. The symbolic execution tree of the Armstrong Program has also been analysed.

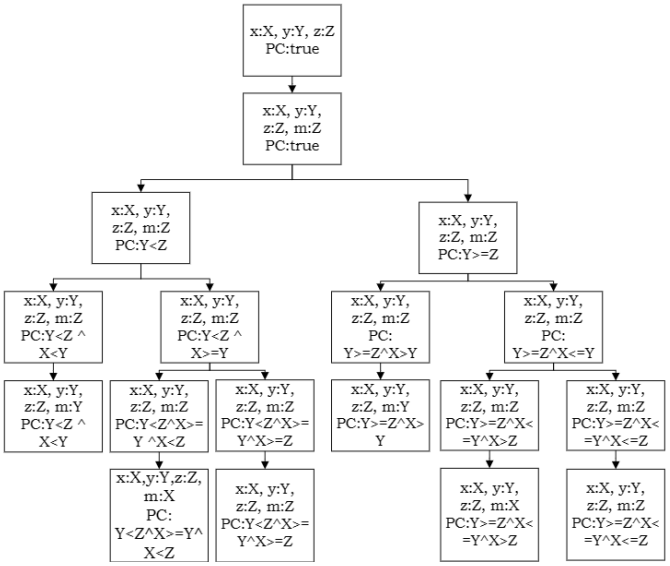



Fig. 3. Symbolic Execution Tree of Median Program

The paths that can be identified for the Armstrong Program with one iteration are three paths. The three paths are as follows:

- PC1: A = 0
- PC2: A !=0 ∧ (A/10) = 0
- PC3: A !=0 ∧ (A/10) != 0

The Multiple Program consists of three methods that are executed sequentially in the main driver program, namely the checkMonth method to return the name of the month from the input value, the checkEvenOdd method to check odd or even numbers and the checkNumber method to check positive or negative numbers. The number of test cases generated depends on the last method executed in the program, namely the checkNumber method, so the number of test cases for the other two methods will be the same as the number of test cases for the checkNumber method. Therefore, the number of test cases for each method in this program is three. In the Multiple Program, experiments were carried out on 28 candidate limit value ranges, and it was found that 14 candidates produced invalid test cases. Limit values that produced invalid test cases were eliminated from the selected candidate limit values because the test cases could not be used. so that the limit value

candidates become 14 candidates. The range of limit values selected to use is a minimum value of 0 and a maximum value of 40. For Multiple Programs, verification of the number of test cases can also be performed based on the symbolic execution tree. The symbolic execution tree for this program is described based on the method being executed**.**

Based on the experimental analysis for Nested If Program, the number of test cases for this program was 41. Since this program is a program that has many lines of program code, the symbolic execution tree was not drawn manually. In the Nested If Program, experiments were conducted on 27 candidate limit value ranges, and it was found that 15 candidates produced invalid test cases. Limit values that produced invalid test cases were eliminated from the selected candidate limit values because the test cases could not be used for experiments, leaving 12 candidates. The range of limit values selected for use in the experiment is a minimum value of 0 and a maximum value of 11.

From the analysis carried out, the number of test cases produced by SPF for the Simple Calculator Program was five. In the Simple Calculator Program, experiments were carried out on 17 candidate limit value ranges, and it was found that seven candidates produced five test cases. Based on this analysis, the maximum number of test cases in the Simple Calculator Program was five. The range of limit values selected in experiments is a minimum value of -1 and a maximum value of 1.

### *C. Experiment Design*

The experiment's algorithm is described as follows:

a. The initial process carried out is to download and install jpf-core and jpf-symbc with the aim of being able to run the SPF tool used in this final assignment. The IDE used to run SPF is NetBeans version 8.1 and uses Java version "1.8.0_171".
b. The next stage is determining the object of study. The selected study object is the program used during the research on this final assignment. The object of study is selected by reading the program code to find out the internal structure of the program. The study objects used in this research were five programs.
c. After selecting a study object, a mutant program is generated for each study object. The mutant program was generated using an open-source Java project called µJava (muJava). muJava is a program that is utilized to generate mutants for Java programs automatically by altering operators in the program or mutation operators [18].
d. Next, test cases are generated using SPF from each original program of the study object. Verification of the effectiveness of this test case is carried out by mutation testing.
e. The test cases produced by SPF can be seen in the NetBeans IDE output display. For research purposes, where there is a testing process for mutant programs, the test cases are stored in a directory to make it easier to read the test cases.
f. Next, the original program and mutant program for each study object read the test cases stored in the storage directory.
g. After successfully reading the stored test case, the test case is executed on the original program and the mutant program. In the original program, to be able to execute the test case, modification on the main driver of the original program is required. The test cases are later executed on the mutant program in the same way as the test cases were executed on the original program. The execution results of the original program are recorded as expected output, and the execution results of the mutant program are recorded as actual output.
h. The next stage is to compare the output between the original program (expected output) and the mutant program (actual output). The output comparison is done automatically. The output of the original program and the output of the mutant program are stored in the form of a .txt file. Then, a comparison is carried out by checking the similarities of the contents of the output files. If the output between the original program and the mutant program is different, the test case is effective because it can detect faults and is classified as a killed mutant. If the outputs match, then it is classified as a live mutant. The aim of this stage is to calculate the mutation score for each study object. Mutation score determines the percentage of test case effectiveness produced by SPF in the program.
i. After obtaining the mutation score for each study object, conclusions stating the effectiveness of the test case are drawn.

## IV. EXPERIMENTAL RESULT

Experiments were carried out on five study objects. Then test cases were generated for each study object where the effectiveness of the resulting test cases was checked. The effectiveness of test cases was checked by applying mutation testing. The number of mutants produced for each study object varied. The number of test cases, number of mutants, number of killed mutants and live mutants for each study object are recorded in Table II.

TABLE II
MUTANTS FOR PROGRAM UNDER TEST

| Program | No of TC | No of Mutant | No of killed mutant | No of live mutant |
|---|---|---|---|---|
| Median | 6 | 147 | 110 | 37 |
| Armstrong | 6 | 135 | 105 | 30 |
| Multiple | 3 | 125 | 39 | 86 |
| Nested If | 41 | 200 | 184 | 16 |
| Simple Calculator | 5 | 206 | 115 | 91 |

After mutation testing is conducted to each of the programs under test, the mutation score was calculated by using the following formula [19][20]:

$$Mutation\ score\ = 100 * \frac{Total\ killed\ mutant}{Total\ mutant}$$

The mutation score resulted from the mutation testing for each of the programs under test is presented in Table III.

TABLE III
MUTATION SCORE RESULTS

| Program | Mutation Score (%) |
|---|---|
| Median | 74.82 |
| Armstrong | 77.78 |
| Multiple | 31.20 |
| Nested If | 92.00 |
| Simple Calculator | 55.82 |

In Table III, the mutation score value for each program is recorded. The Median Program mutation score reached 74.82%, the Armstrong Program 77.78%, the Multiple Program 31.20%, the Nested If Program 92.00% and the Simple Calculator Program 55.82%. Of the four programs above, the highest mutation score is obtained by the Nested If Program whereas the lowest is produced by the Multiple Program.

## V. DISCUSSION

Limit value is an additional option provided and is a range of values that will be used for the values of candidate test cases generated by SPF. Without using limit values, the values of the candidate test cases used are the minimum and maximum values of the integer type. In this experiment, a weakness in SPF was identified in generating test cases, proven by the existence of invalid test cases. An invalid test case is a test case that has a value exceeding the maximum limit of the integer type, meaning that the test case is inoperable. Invalid test cases are generated because the minimum limit value given is negative, such as the experiments carried out in the Armstrong, Multiple, and Nested If programs. The limit value chosen is the value that produces a valid test case. This causes not all program paths to be traversed due to the lack of test cases used to meet all paths in the program. The resulting test case value will depend on the limit value used. Therefore, selecting a different limit value will give different results.

From Table III, it was recorded that there was no testing on program under test that obtained a mutation score reaching 100%. From the results of the analysis carried out, the test cases produced by SPF meet the Decision Coverage (DC) criteria. A test case that meets the decision coverage condition is a test case that focuses on the condition of each branch in the program code being tested, which means that each branch must be executed at least once. Test cases that meet decision coverage are test cases that fulfill the decision on the branch and not on each clause, which forms the branch. In this case, decision coverage still has limitations, where there are still other coverage criteria that should be investigated, such as the Decision Condition Coverage (DCC) criteria. Examination using the Decision Condition Coverage criterion is a criterion where the test case not only meets the decision conditions of a branch, but also fulfills every clause that forms the branch. The experimental results indicate that the test suite produced by SPF was unsuccessful in detecting all existing faults because SPF only applied decision coverage in determining the path during the test case formation process.

To produce test cases, SPF creates a symbolic execution tree to determine the number of paths formed by a program. In this research, the symbolic execution tree is drawn manually for programs with few lines of program code such as the Median Program, Armstrong Program, and Multiple Program. Meanwhile, for programs with many lines of program code, such as the Nested If Program and the Simple Calculator Program. Verification using a manually drawn symbolic execution tree allows room for human error when drawing it, creating a different obtained path with the path generated by SPF. This can result in an invalid test case verification. In this research, verification of the number of test cases for a program was also obtained by carrying out a series of experiments on several ranges of limit values. The use of different minimum-maximum value ranges will affect the number of test cases and test case values produced in a program. Verification when test cases have reached the maximum number is carried out if the number of test cases produced remains the same despite the increase in the range values given. Human error may occur in a test case variation, where the range of minimum-maximum values given is incorrect, causing the results obtained to change and be invalid.

## VI. CONCLUSION

SPF forms a symbolic execution tree to generate test cases. The resulting test cases are test cases that meet the conditions at each branch of the symbolic execution tree. This means that the test cases produced by SPF are test cases that meet the decision coverage where each branch must be executed at least once. The weakness of the SPF tool that was discovered during experiments in this final project was that invalid test case values were obtained, namely values exceeding the maximum integer limit for several programs which used a negative minimum value limit for their candidate test cases. To verify the test cases produced by SPF, mutation testing is carried out. Based on the experiments carried out, it was found that the mutation score value for the study object under study did not reach 100%. This means that the test cases produced by SPF for the study object are insufficient because the test cases produced by SPF are test cases that meet the decision coverage criteria which still have limitations compared to other coverage criteria.

In this research, it was found that not all faults in mutant programs could be detected using test cases generated by SPF. Therefore, it is necessary to carry out more in-depth research on the test cases produced by SPF using examination of path coverage criteria. It is necessary to verify the path with a symbolic execution tree produced by a tool (automation) to

reduce the possibility of manual process errors. Determining the limit value for generating test cases needs to be investigated further. It is important to find a way to obtain the most optimal limit value.

## ACKNOWLEDGMENT

This research is fully supported by the Institut Teknologi Del, Indonesia in providing the necessary facilities**.**

## REFERENCES

[1] T. Parveen, S. Tilley and G. Gonzalez, "A Case Study in Test Management," *ACMSE* 2007, pp. 82-87, 2007.
[2] I. Hooda and R. S. Chhillar, "Software Test Process, Testing Types and Techniques," *International Journal of Computer Applications*, vol. 111, no. 13, pp. 10-14, February 2015.
[3] R. M. Sharma, "Tools and Techniques of Code Coverage Testing," *International Journal of Computer Engineering and Technology (IJCET)*, vol. 5, no. 9, pp. 165-171, 2014.
[4] P. Mahadik, D. Bhattacharyya and H.-J. Kim, "Techniques for Automated *Test Case*s Generation: A Review," *International Journal of Software Engineering and Its Applications*, vol. 10, no. 12, pp. 13-20, 2016.
[5] A. M. Sinaga, P. A. Wibowo, A. Silalahi and N. Yolanda, "Performance of Automation Testing Tools for Android Applications," *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 2018, pp. 534-539.
[6] S. Pathy, S. Panda, S. Baboo, "A Review of Code Coverage Analysis," *International Journal of Computer Computer Science & Engineering Technology (IJCSET)*, vol. 6, no.10, pp. 580-587, 2015.
[7] Z. Q. Zhou, A. Sinaga, W. Susilo, L. Zhao, K. Y. Cai, "A cost-effective software testing strategy employing online feedback information," *Information Sciences*, vol. 422, 2018, pp 318-335.
[8] A. M. Sinaga, A. S. Dharma, O. Hutajulu, A. Ginting, & G. Simanjuntak, "Dynamic Partitioning and Additional Branch Coverage for Test Case Selection," *Journal of Physics: Conference Series*, vol. 1175, 2019, pp. 012098
[9] "Symbolic Path Finder" [Online]. Available: https://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc. [Accessed: 26 February 2018].
[10] "Java Path Finder" [Online]. Available: https://babelfish.arc.nasa.gov/trac/jpf. [Accessed: 26 February 2018]
[11] S. Kunze, "Automated Test Case Generation for Function Block Diagrams Using Java Path Finder and Symbolic Execution," *Thesis for the Degree of Master of Science in Computer Science with specialisation in Software Engineering, Malardalen University*, School of Innovation Design and Engineering, Vasteras, Sweden.
[12] C. S. Pasareanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz and N. Rungta, "Symbolic Path Finder: integrating symbolic execution with model checking for Java bytecode analysis," *Autom. Softw. Eng.*, no. 20, pp. 391-425, 2013.
[13] D.-P. Nguyen, C.-T. Luu, A.-H. Truong and N. Radics, "Verifying implementation of UML sequence diagrams using Java Path Finder," *Second International Conference on Knowledge and Systems Engineering*, pp. 194-200, 2010.
[14] C. S. Pasareanu and N. Rungta, "Symbolic Path Finder: Symbolic Execution of Java Bytecode," *ACM*, 2010.
[15] K. Kähkönen, "Evaluation of Java Path Finder Symbolic Execution Extension," pp. 1 - 20, June 2007.
[16] "Software-artifact Infrastructure Repository" [Online]. Available: http://sir.unl.edu. [Accessed: 9 January 2018]
[17] "GeeksforGeeks" [Online]. Available: https://www.geeksforgeeks.org/. [Accessed: 01 March 2018]
[18] "muJava Home Page" [Online]. Available: https://cs.gmu.edu/~offutt/mujava/. [Accessed: 02 July 2018]
[19] M. Umar, "An Evaluation Of Mutation Operators For Equivalent Mutants," *Department of Computer Science, King's College, London.*, September 2006.
[20] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Journal Empirical Software Engineering*, vol. 14 , no. 3, pp. 341 - 369, June 2009.