JITK (JURNAL ILMU PENGETAHUAN DAN TEKNOLOGI KOMPUTER)

VOL. 7. NO. 2 FEBRUARY 2022
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v7i2.2758

IMPLEMENTATION OF CONCOLIC UNIT TESTING IN TESTING BINARY SEARCH ALGORITHM USING JCUTE

Neneng Rachmalia Feta[1*]; Fitria[2]

Information Systems and Technology[1,2]
Institut Teknologi dan Bisnis BRI
http://bri-institute.ac.id/
nenengrachmaliafeta@gmail.com[1*], fitria.fitrib@gmail.com[2]

(*) Corresponding Author

Abstract— Testing is the process of implementing a program to find an error. A good test case is one of the tests that have the possibility of finding an undisclosed error. One of the existing types of testing is the Concolic Unit Testing Engine. In this research, testing is applied using the JCute tool, which is a tool to systematically and automatically test sequential C programs (including instructions) and Java programs together. This test is carried out on the search function of an element of data in the Binary Search Search Algorithm. However, to check whether concolic testing can detect bugs in the software practically through case studies. This research describes a case study of the application of a test tool to a Java application. Through this research, we tested the path coverage and Branches Covered. We can also find out the details of total branches covered; total functions invoked, percentage of branches covered, and the number of iterations. JCute can also find an interleaving of two sequences or circuits that results in an infinite loop.

***Keywords**: Testing, Concolic Unit Testing Engine, JCute, Binary Search.*

Abstrak— *Pengujian adalah suatu proses pelaksanaan suatu program dengan tujuan menemukan suatu kesalahan. Suatu kasus test yang baik adalah apabila test tersebut mempunyai kemungkinan menemukan sebuah kesalahan yang tidak terungkap. Salah satu dari jenis pengujian yang ada adalah Concolic Unit Testing Engine. Pada penelitian ini diterapkan pengujian dengan menggunakan tools JCute yang merupakan alat untuk secara sistematis dan otomatis menguji program C berurutan (termasuk petunjuk) dan program Java bersamaan. Pengujian ini dilakukan pada fungsi pencarian suatu elemen atau data pada Algoritma Pencarian Binary Search. Namun untuk memeriksa apakah pengujian concolic dapat mendeteksi bug pada perangkat lunak secara praktis melalui studi kasus. Penelitian ini menjelaskan studi kasus penerapan alat uji coba ke aplikasi Java. Melalui penelitian ini, kami melakukan pengujian terhadap path coverage dan Branches Covered. Kita juga dapat mengetahui detail total branches covered, total functions invoked, percentantage of branches covered dan number of iterations. JCute juga dapat menemukan sebuah interleaving dari dua urutan/rangkaian yang menghasilkan loop tak terbatas.*

***Kata Kunci**: Pengujian, Concolic Unit Testing Engine, JCute, Binary Search.*

INTRODUCTION

Testing is a series of activities that can be planned and carried out systematically. Activities related to testing include analyzing items and programs and features of software items. [1]. Software testing is an essential part of determining software quality, every software that has been built needs to be tested so that there are no logical process errors and to ensure the software is 100% correct according to needs, errors in the software built not because a programmer does not pay attention to the processes that occur or do not pay attention to the quality of the software he creates, errors can occur because of the complexity of the software logic and the broad scope of the software can be the reason for these errors.

The purpose of software testing is to detect differences between the output of the software and the expected conditions and find errors. By testing, the quality and trust in the functioning of the software will increase. Bortolino, in his research "Software Testing Research: Achievements, Challenges, Dreams", mentions several approaches that can be used in software testing, including model approaches, engineering approaches, search-based approaches for the generation of test inputs and attribute performance assessment approaches [2].

Binary Search Algorithm is implemented to assist software testing. This algorithm is a method of searching for data or elements in an array with data conditions in an ordered state. The binary search process can only be done on a set of data that has been sorted beforehand. The researcher implements the Binary Search Algorithm into the JCute software to perform computerized software testing. JCute (Java Concolic Unit Testing Engine) for C and Java is a tool for systematically and automatically testing sequential C programs (including hints) and concurrent Java programs.

According to Sen. K and Agh. G [3], during execution, the algorithm collects a constraint on the symbolic value at each branch point (that is, a symbolic constraint). At the end of implementation, the algorithm has calculated the sequence of symbolic constraints corresponding to each branch point. We call this constraint conjunction a path constraint. Note that all input values that satisfy a given path constraint will explore the same execution path, provided we follow the same thread schedule. In addition to collecting symbolic constraints, the algorithm also calculates race conditions (data race and lock race) between various events in program execution, where informally, an event represents the execution of a statement in the program by a thread.

According to Kim. M, and Jang. Y [4]". Concolic Testing (concrete and symbolic) is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution path (testing on specific inputs).

Concrete testing explores possible paths in the same way as symbolic execution. [5][6]. Unit tests are built to map symbolic input to function parameters. This technique then aggregates the symbolic input value constraint and a set of path constraints and problem solvers. In addition, this technique collects input values with a constraint solver that produces test input values that can make high path coverage. When this technique resolves all constraints, it uses the tangible value of execution for the algorithm to continue. Moreover, since the algorithm performs a concrete implementation, all errors inferred by the technique are real.

The concolic test tool can also identify input and output variables used to generate test cases to determine input/output dependencies on the application [7]. And it can also get a high execution scope and is widely used in the industry for program testing [8].

JCute combines concrete and symbolic execution in a way that avoids overtesting as well as false warnings. The tool also introduces a race-flipping technique to efficiently test and check programs along with data input. Related to testing, the JCute Binary Search Algorithm can automatically search for possible input data for the test case design.

Based on the background of the problem as described previously, this research designs and builds software using a binary search algorithm and JCute as tools used to test software. And analyze software testing using the Concolic Unit Testing Engine approach, which can find undiscovered errors using JCute.

MATERIALS AND METHODS

The research methods applied in obtaining data and information that support this research are as follows :

Study of literature

Literature research is used as the basis for theoretical discussion by using data obtained from observations and evaluating the results of journal research, theories and views from books, internet searches and other sources in this study.

Software Testing Method

The basis of any software testing life cycle is first of all the knowledge about the specified system to be developed and all influencing factors. From the point of view of a model-centric approach to the problem, this knowledge base was created by first developing a concept that enables a lightweight integration of model information from all modeling domains (influencing factors) [9].

The method in writing this research consists of several steps, which we can see in Figure 1 below.

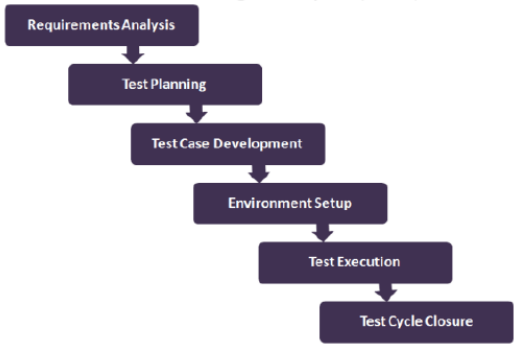


Figure 1. Software Testing Life Cycle [10] (STLC)

Requirements Analysis

In this first stage of the software testing cycle, the test team reviews each document and design requirements to determine what can be tested. By studying the needs, the test team understands the scope of the test. This phase may involve conversations with developers, designers, and stakeholders [11].

Accredited Rank 3 (Sinta 3) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti
No. 28/E/KPT/2019, September 26, 2019. Published by PPPM STMIK Nusa Mandiri

VOL. 7. NO. 2 FEBRUARY 2022
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v7i2.2758

Test Planning

What to test, how the test needs to be done, and who will try it. These are things that are determined during the testing planning stage. Once the requirements have been reviewed, it's time to plan a test project. A test plan document is created during this phase. This phase keeps everyone on the same page about how the test project will be approached.

Test Case Development

This stage aims to determine in detail the "how" to test. Test cases should be written to guide the tester through each test. If old test cases are used, make sure they are up to date. Many tests may require test data. Prepare the test data needed to run the test during this phase, so you don't have to spend time doing this during the trial.

Environment Setup

Ensure that the required test data is entered into the system and is ready for use. The test environment is the software and/or hardware configuration in which the test team performs testing. Without a ready-to-use test environment, you will run into a bottleneck.

Test Execution

Now that the tests are ready to run and the environment is set up, it's time to run the tests. Using test cases, the tester executes each test, compares the expected result with the actual outcome of each test and marks it as pass or fail or skip. If the test fails, the examiner must document what actually happened during the trial. This phase also involves bugs in the designated bug tracking system (defined in the planning phase)[12].

Test Cycle Closure

After all test cases have been run, the test manager must ensure that all required testing has been completed. This involves analyzing the defects found and other metrics such as how many test cases were passed/failed. This final stage in the software testing cycle may also include a project/testing process retrospective. This allows the team to learn from and improve future test projects.

RESULTS AND DISCUSSION

Requirements Analysis

The analysis is the first step in testing software. At this stage, the analysis process includes problem analysis software specification analysis. This analysis combines literature study data obtained from the data collection process and methods in testing. In this stage of testing, identify the requirements that can be tested. Activities that must be carried out in the needs analysis stage are as follows:

1. Analyze the system requirements specification from a testing point of view
2. Identify testing techniques and types of tests
3. Prioritize features that require focused testing
4. Analyze the feasibility of automation
5. Identify details about the test environment in which the actual testing will be carried out

Test Planning

The activities that will be carried out in the Test Planning stage are as follows :

1. Estimated testing effort
2. Selection of Testing Approach
3. Preparation of Test Plan, a Test Strategy document
4. Selection of Testing tools

Deliverables (Results) from the Test Planning stage are :

1. The most suitable Test Approach: Concolic Unit Testing
2. Test equipment to be used: JCute

Test Case Development

The activities that will be carried out in the Test Case Development stage are as follows :

1. Making test cases, namely binary search application cases
2. Test script creation if needed
3. Verify test cases and automation scripts
4. Test Data Creation in the test environment

Class BinarySearch.java

```
1.  public class BinarySearch {
2.      private int[] mData;
3.      private int mSize;
4.      public BinarySearch(int[] data) {
5.          this.mData = data;
6.          this.mSize = data.length;
7.      }
8.
9.      public int search(int key) {
10.         int low = 0;
11.         int high = this.mSize - 1;
12.         while(high >= low) {
13.             int middle = (low + high) / 2;
14.             if(this.mData[middle] == key) {
15.                 return middle;
16.             }
17.             if(this.mData[middle] < key) {
18.                 low = middle + 1;
19.             }
20.             if(this.mData[middle] > key) {
21.                 high = middle - 1;
22.             }
```

```
23.         }
24.
25.         return -1;
26.     }
27.}
```

Class MainConsole.java

```
public class MainConsole {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.out.println("Parameternya
belom");
            return;
        }
        int[] data = {1,2,3,4,5,6,7,8,9,10};
        BinarySearch binarySearch = new
BinarySearch(data);

        int search = Integer.parseInt(args[1]);
        int result = binarySearch.search(search);
        if(result != -1) {
        System.out.println("Yang dicari: " + result +
" ada pada index ke: " + result);
        } else {
            System.out.println("Ndak ado");
        }
    }
}
```

Environment Setup

The activities that must be carried out in the Test Environment Setup stage are as follows :

1. According to the requirements and Architectural documents, prepared a list of software and hardware
2. Setting up the test environment
3. Generating test data
4. Install build and test execution

Potongan Sourcode yang akan di Uji

```
1.  public int search(int key) {
2.  int low = 0;
3.  int high = this.mSize - 1;
4.
5.  while(high >= low) {
6.      int middle = (low + high) / 2;
7.      if(this.mData[middle] == key) {
8.          return middle;
9.      }
10.     if(this.mData[middle] < key) {
11.         low = middle + 1;
12.     }
13.     if(this.mData[middle] > key) {
14.         high = middle - 1;
15.     }
16.}
17.
18.return -1;
19.}
```

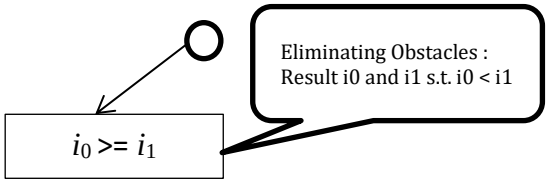
In the algorithm snippet above, the low variable value is inputted 0; then the high variable is calculated using the formula mSize – 1. Then we try to violate this statement: int middle = (low + high) / 2;. Then we assign a random value to the variable int low = 0; the value is 10 and the variable int high = this.mSize - 1; the value is 6. In Table 1 it is Randomly input concrete value Concolic Execution.

Table 1. Randomly input concrete value

	Concrete	Symbolic
low	10	i_0
high	6	i_1

Enter the concrete value, which is the new input:
Low = 10 which is a concrete value, and i0 is a symbolic value
High = 6 which is a concrete value, and i1 is a symbolic value.

Then we execute the Execution Tree based on the equation below:



Where variable int middle = (low + high) / 2; . Then we try again to make a new input, this time the variable int low = 0; the value is 8 and the variable int high = this.mSize – 1; its value is 20, and the following statement corresponds to the equation int middle = (low + high) / 2;. We input for the new concrete value of concolic execution after we got for the low variable the concrete value is 8, and the high concrete value is 20, which can be seen in Table 2 for the concrete value and its symbolic value.

Table 2. Enter the new concrete value

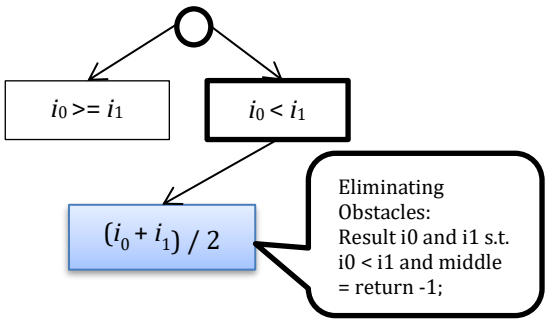
	Concrete	Symbolic
low	8	i_0
high	20	i_1

Enter the concrete value, which is the new input:
Low = 8 which is a concrete value, and i0 is a symbolic value
High = 20 which is a concrete value, and i1 is a symbolic value

Accredited Rank 3 (Sinta 3) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti
No. 28/E/KPT/2019, September 26, 2019. Published by PPPM STMIK Nusa Mandiri

VOL. 7. NO. 2 FEBRUARY 2022
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v7i2.2758

Then we can re-execute the Execution Tree based on the concrete and symbolic values in Table 2 to produce the following equation:



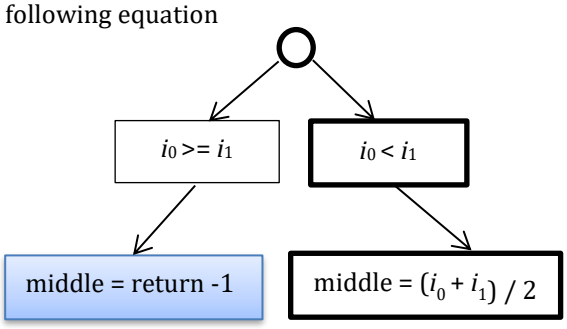
Then we try again to make a new input, but at this time, the variable int low = 0; has value 100 and variable int high = this.mSize - 1; the value is 4. The value of the two variables is obtained based on the following equation int middle = (low + high) / 2; In Table 3, we can see the low and high variable values that we got earlier, and we inputted them as new concrete values for the concolic execution.

Table 3. Enter the next new concrete value

	Concrete	Symbolic
low	100	i_0
high	4	i_1

Enter the concrete value, which is the new input:
Low = 100 which is a concrete value, and i0 is a symbolic value
High = 4 which is a concrete value, and i1 is a symbolic value

For the next experiment, we can execute the Execution Tree again based on the concrete and symbolic values in Table 3 so as to produce the following equation



Testing experiments carried outrun the program concretely and symbolically. Symbolic execution differs from traditional symbolic execution in that the algorithm follows a path that requires concrete execution. During execution, the algorithm collects a constraint on the symbolic value at each branch point (that is, a symbolic constraint). At the end of execution, the algorithm has calculated the sequence of symbolic conditions corresponding to each branch point. We can notice that all input values that satisfy a given path constraint will explore the same execution path, provided we follow the same thread schedule.

In addition to collecting symbolic constraints, concolic unit testing also calculates race conditions (data race and lock race) between various events in program execution, where informally, an event represents the execution of a statement in the program by a thread. The first algorithm generates random input and a schedule, which determines the execution order. Then the algorithm does the following in a loop: it executes the code with the information and the resulting program. At the same time, the algorithm calculates the race conditions between various events and the symbolic constraints. It generates backtracks and generates a new schedule or new input, either by reordering the circumstances involved in the race or by breaking symbolic barriers, respectively, to explore all possible different execution paths using a deep first search strategy. Note that the algorithm performs a concrete execution, i.e., all the bugs it finds are actual.

Test Execution

After executing several test cases on the binary search algorithm by automatically generating inputs and schedules so that each program execution path is performed at once, we can see in Figure 2 that JCute develops a testing path of 16 paths. Incorrect paths are indicated by an asterisk (*) to view appropriate inputs and traces. The asterisk symbol (*) on path 10 will also cause program error, while on path 7, it is not wrong because there is no sign (*).

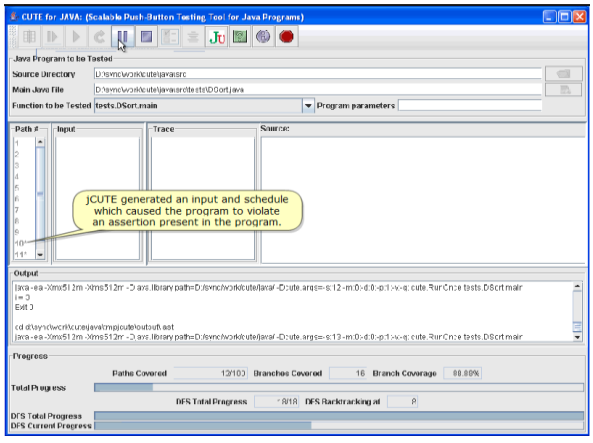


Figure 2. Incorrect traced program

Tests carried out on all existing paths when searching for paths manually will take a long time, but using JCute can be found automatically and thoroughly. It can determine error paths for certain data cases.

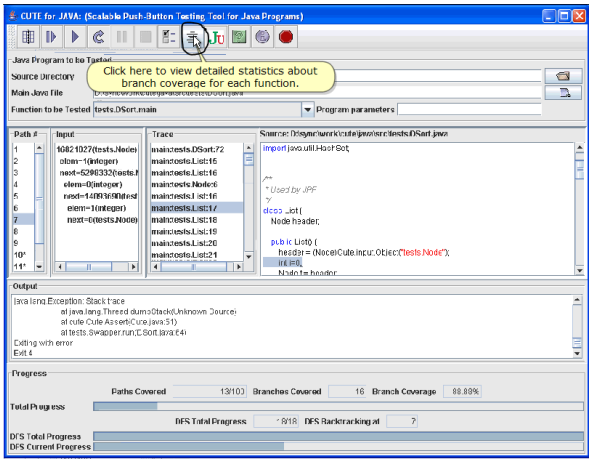


Figure 3. Branch coverage for each function

In Figure 3, it can be seen that JCute generates different inputs for each path and is also visualized in Figure 3 to the right of the code to be executed. It can be seen that the input variable is the branch we want to run. Trace performed as many as 72 TraceList.

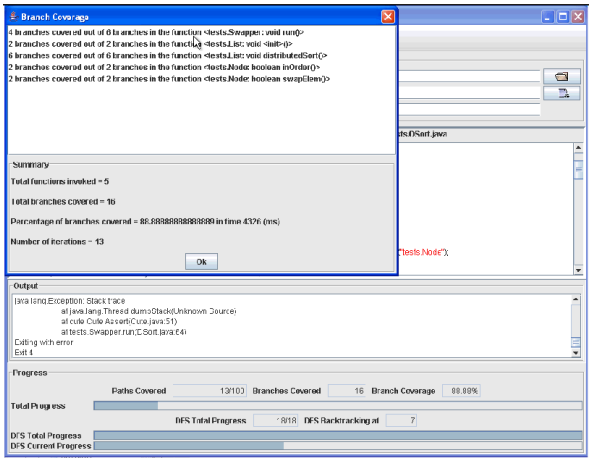


Figure 4. Details of branch coverage

Java programming with the input data provided, the results are shown in Figure 4 and more details are shown in Table 4 include 5 total functions invoked, 16 total branches coverage, 88.88 percentage of branches coverage, and 13 number of iterations.

Table 4. Evaluation results of binary search testing using JCute.

Number Of Iterations	Total Functions Invoked	Total Branches Coverage	Percentage Of Brances Coverage
13	5	16	88.88
8	4	11	79.56
2	3	9	70.03

If the number of iterations is 2, the Percentage Of Branches Coverage only reaches 70.03. If the number Of Iterations is 8, the Percentage Of Branches Coverage reaches 79.56. If the number of iterations is 13, the Percentage Of Branches Coverage reaches 88.88.

CONCLUSION

In this study, the results obtained from analyzing the program source code for software testing using the Concolic Unit Testing Engine approach that the JCute application has been successfully used in testing the binary search algorithm. The binary search algorithm built in the Java programming language with the input data resulted in 5 total invoked functions, 16 total branches coverage, 88.88 percentage of branches coverage, and 13 iterations. The obtained percentage of branches coverage is above 70 percent. And the total function invoked is more than 0. The test driver also calls the algorithm by entering the concrete value and the number of iterations generated according to the function and concrete value entered. Testing the binary search algorithm in the Java programming language has been successfully carried out using JCute automatically.

REFERENCE

[1] B. B. Agarwal, S. P. Tayal, and M. Gupta, *Software Engineering and Testing*. 2010.

[2] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” *FoSE 2007 Futur. Softw. Eng.*, no. September, pp. 85–103, 2007, doi: 10.1109/FOSE.2007.25.

[3] K. Sen and G. Agha, “CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools,” *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, vol. 4144 LNCS, pp. 419–423, 2006, doi: 10.1007/11817963_38.

[4] M. Kim, Y. Kim, and Y. Jang, “Industrial application of concolic testing on embedded software: Case studies,” *Proc. - IEEE 5th Int. Conf. Softw. Testing, Verif. Validation, ICST 2012*, pp. 390–399, 2012, doi: 10.1109/ICST.2012.119.

[5] S. Godboley, A. Dutta, A. Das, and D. P. Mohapatra, “Measuring MC/DC at design phase using UML sequence diagram and concolic testing,” *2016 IEEE Annu. India Conf. INDICON 2016*, pp. 0–5, 2017, doi: 10.1109/INDICON.2016.7839079.

[6] S. Godboley, D. P. Mohapatra, A. Das, and R. Mall, “An improved distributed concolic testing approach,” 2017, doi: 10.1002/spe.2405.

Accredited Rank 3 (Sinta 3) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No. 28/E/KPT/2019, September 26, 2019. Published by PPPM STMIK Nusa Mandiri

VOL. 7. NO. 2 FEBRUARY 2022
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v7i2.2758

[7] M. E. Ruse and S. Basu, "Detecting cross-site scripting vulnerability using concolic testing," *Proc. 2013 10th Int. Conf. Inf. Technol. New Gener. ITNG 2013*, pp. 633–638, 2013, doi: 10.1109/ITNG.2013.97.

[8] R. Ahmadi, K. Jahed, and J. Dingel, "MCUTE: A model-level concolic unit testing engine for UML state machines," 2019, doi: 10.1109/ASE.2019.00132.

[9] R. Pröll, "Towards a model-centric software Testing life cycle For early and consistent testing Activities," University of Augsburg, 2021.

[10] I. Bhatti, J. A. Siddiqi, A. Moiz, and Z. A. Memon, "Towards Ad hoc testing technique effectiveness in software testing life cycle," 2019, doi: 10.1109/ICOMET.2019.8673390.

[11] Guddi Singh, "A study on software testing life cycle in software engineering," *Int. J. Manag. IT*, vol. Vol 9, no. No 2, 2018, [Online]. Available: https://globusjournal.com/wp-content/uploads/2018/12/9227Guddi.pdf.

[12] N. Honest, "Role of Testing in Software Development Life Cycle," *Int. J. Comput. Sci. Eng.*, 2019, doi: 10.26438/ijcse/v7i5.886889.

VOL. 7. NO. 2 FEBRUARY 2022
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480 /jitk.v7i2.2758