Automated Test Cases Generation From Requirements Specification

1st Mohammed Lafi
Department of Software Engineering
Faculty of Science and Information Technology
AL- Zaytoonah University of Jordan
Amman, Jordan
lafi@zuj.edu.jo

2nd Thamer Alrawashed
Department of Software Engineering
Faculty of Science and Information Technology
AL- Zaytoonah University of Jordan
Amman, Jordan
thamer.r@zuj.edu.jo

3rd Ahmad Munir Hammad
Department of Software Engineering
Faculty of Science and Information Technology
AL- Zaytoonah University of Jordan
Amman, Jordan
ahmad.hammad94@yahoo.com

2021 International Conference on Information Technology (ICIT) | 978-1-6654-2870-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICIT52682.2021.9491761

***Abstract*— One of the most significant phases of software development is software testing, because of its value in identifying mistakes and gaps in the early stages of program development. In the past, software testing was done manually, and this is a tiring and inaccurate process that carries with it errors and gaps and requires time, effort, and money. At present, testers tend to perform the testing process automatically, to save time, effort, and money, and to obtain accurate results. However, there are few research works on the aspect of generating test cases from requirements specification especially generate test cases from use case description. An Approach to Generate Test Cases from Use Case description is proposed in this paper that consists of several processes. The input is the use case description of the use case diagram, which is being used as a basis for the approach. In the next step, each software summary of UML use cases is utilized to extract the necessary information for the development of the control flow graph and NLP table. A control flow graph and NLP table will be generated based on a specific algorithm after that generate test paths based on a specific algorithm. Then, the test cases will be generated from the test paths and NLP table. The proposed approach will enhance the process of generation of test cases and increase the accuracy and efficiency.**

Keywords*— *Software Testing, Test Case Generation, NLP, Use Case Description.

I. INTRODUCTION

Software engineering (SE) is the disciplined employment of engineering, scientific, and mathematical principles and methods to produce software of high quality [4]. Another definition of software engineering is the detailed study of software to analyze, design, develop and maintain the software product. Another perspective for defining software engineering, applying a systematic, disciplined, and quantifiable approach to develop, operate and maintain software[12].

System testing is performed when software development is finished. The testing team begins testing the entire system's functionality. This is done to ensure that the entire application functions as expected by the customer. QA and testing teams may discover bugs or defects during this phase, which they report to developers. The development team fixes the issue and sends it back to QA for another round of testing. This process is repeated until the software is bug-free, stable, and meets the system's business requirements. The testing phase is one of the most significant phases of the life cycle of system development. There have been several suggested definitions of testing: The process of analyzing a software item to detect differences between existing and required conditions (bugs) and to evaluate the software item's features [5].

Testing is intended to demonstrate that a program does what it is supposed to do and to identify program flaws before it is put into use [20]. The process of manually or automatically exercising a system or system component to ensure that it meets specified requirements or to identify differences between expected and actual results [5]. Testing on a complete, integrated system to determine compliance with the system's specified requirements [6]. Software testing is the method of discovering bugs in a program's code lines that can be achieved through manual or automation testing [19]. Software testing is a method of running a program to identify errors [8]. Software testing is the method of checking whether the actual results of the software match the intended results set out in the criteria and specifications and ensuring that there is no fault in the software. Software testing aims to find software mistakes, faults, or defects and to ensure that meets the software requirements specifications.

The Unified Modeling Language (UML) is a structured modeling language consisting of an interconnected collection of diagrams designed to assist system and software developers to define, imagine, construct and record software system objects, as well as business modeling and other non-software systems [14]. The steps taken in a UML use case are defined by UML diagrams and demonstrate a business process or workflow between users and the system [25].

Several suggested methods include UML diagrams for the generation process of test cases in software testing. As the UML diagrams are generated to define the software requirements in the analysis process, developers can also benefit from them to generate the test cases easily. Before using them in generating the test cases, some of the UML diagrams need to be pre-processed. For example, if your information is not sufficient when using a UML diagram, it can be combined with other UML diagrams to obtain more information. There are several problems and challenges in creating test cases using UML use case description. First, it requires a long time and big efforts. In addition, such a system has a high probability of logical and human errors. UML is not adequate to describe the personalization of complex and comprehensive systems. (Use case diagrams, for example) as informal charts that can be differently interpreted. Therefore, it was important to establish the textual language for such components. In this case, explicitly to organize the use case definition, different textual models were used for software requirements i.e. Cockburn's use case description template [2].

Manual testing is a type of testing that is done by human tester. By following the conditions written in test cases, Quality Assurance (QA) specialists ensure that applications

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.

work properly. Manual testing is still important, despite its primitive nature, because certain functionality cannot be tested automatically. Manual testing may suffer from the following disadvantages: it takes a lot of time and effort, test cases have low coverage, low efficiency, and low quality. In addition, changes in the system require a change in the test cases.

Test Automation is a software testing process that uses special automated testing software tools to carry out a test case. Automated Testing saves time and money, manually testing is costly and time-consuming. Automated testing aims to simplify testing efforts, time, and cost. Then, the test was performed according to this; the results were reported and compared with the results of the previous tests.

Tests are done manually by a QA Analyst. It's used to find bugs in software that's still in development. In manual testing, the tester examines all of the application's or software's critical features. Without the use of any automation software testing tools, software testers execute test cases and generate test reports in this process. It is a traditional method for all types of testing that aids in the discovery of bugs in software systems. To complete the software testing process, it is usually done by an experienced tester.

Artificial Intelligence and NLP techniques can be used in test case generations [11,15], as well as other domains [21,26-28].

The use of test automation can increase the speed of test case creation and enhance the coverage of test cases. Test automation makes it easy and cost-effective to use regressing testing. Also, in the case of software change it easier to change test cases generated automatically.

This is why automation testing is preferable to manual testing because you can schedule your test cases to run at any time of day, remotely from any location, and analyze the test results using reports generated from your test suite execution.

Another significant advantage of automation testing is the reusability of test scripts. Writing effective test cases takes a lot of time and effort, and re-performing the same test cases over and over again can be mentally draining. As a result, automation testing is preferable to manual testing. When you automate testing, you write a test script once and can reuse it as many times as you want. Let's take a look at this from the standpoint of cross-browser testing. When performing cross-browser testing, you may need to write test scripts or test cases for each application you test on different operating systems or devices.

The automated test scripts are reusable; even if the device's OS version changes, you don't always need new scripts. It's possible to repeat the test without forgetting any of the steps. Test scripts can be reused as many times as needed with automated testing tools, saving both time and effort. Cross-browser compatibility across various browsers, hardware, operating systems, networks, and mobile devices is critical when developing a website. Browser compatibility testing necessitates the creation of a large number of test cases. Especially when you consider the number of browser versions on the market. Manual testing across hundreds of browser and OS combinations may cause the software release to be delayed.

Automated testing, on the other hand, allows testers to run tests on different browsers, operating systems, and devices at the same time

II. Background and Related Works

This section covers the literature examined and the key difference between the prior work and the proposed work suggested.

Paul and Tommy [11] through the framework architecture discussed that there are three layers of the structure consisting of beautiful web scrapping soup, ML for predicting test case for each web feature, and finally selenium for performing different test cases [11].

Santiago et al. [15] investigated a realistic method that leverages artificial intelligence (AI) and machine learning (ML) technologies to produce device tests directly from human testers based on learning testing actions. Santiago, et al. to construct test cases learned from human testers, the approach combines a trainable classifier, which perceives application status, a language for describing test flows, and a trainable test flow generation model. Preliminary findings from the implementation of a prototype of the strategy are encouraging and put us one step closer to bridging the gap between human and computer research [15].

Grano et al. [3] suggested the branch coverage expectation (BCE) metric as the difficulty for a computer to generate test cases, based on two different test data generation tools in their study: Evo Suite, based on genetic algorithms, and Randoop, which implements a random testing approach [3].

Wang et al. [19] proposed the use case modeling for System Tests Generation (UMTG), a methodology that allows the construction of usable system test cases from natural language requirement specifications, optimizes the generation of system test cases dependent on use case specifications, and a domain model for the system being evaluated uses an NLP solution (i.e. semantic role labeling) OCL constraints that capture the pre and post conditions of use case steps automatically generated [19].

Wang, et al. [17] to extract the pre-and submit-conditions of the activities defined in the use case specification automatically, a method called OClgen proposed. The proposed method enables, at the same time as heading off much of the additional modeling attempt needed with the support of UMTG, the automated generation of test instances from use case speciation. Consequences from an industrial case study show that the method can produce more than 75 percent of the pre-and up-situations characterizing the activities mentioned in use case speciation systematically and efficiently [17].

Wang, et al. [18] addressed system check technology (UMTG) uses case modeling, a technique that produces executable gadgets, looks at use case speciation cases, and a domain version that provides a diagram and constraints of the category. UMTG used natural language processing (NLP), a confined method of use case speciation, to extract behavioral records from use cases and to produce check case automation [18].

In [1] an automated test case generator solution was suggested, which was used to minimize complexity and increase the percentage of test coverage. To convert use case information to a control flow diagram, move it to the

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.

Algorithm of Control Flow Diagram (ACFD) first stage to approach input use case explaining then. Finally, the test cases generated from the control flow diagram using the proposed test path generation method (PTGTP). To refine and assess the adequacy of such test cases, the authors used genetic algorithms [1].

The objective of the study proposed by Khalifa et al. [9] is to use a machine learning approach to create test cases from a use case diagram. There is a Meta-heuristic methodology used for the automation of the test case generation process. The quality and period of computing are the criteria used to test the proposed method's efficiency. The study results have shown that the efficiency of the methodology suggested is better than current techniques [9].

Jaffari et al. [7] suggested a model-based approach, based on its data flow component, using the operation model of the system under test as a test base. To completely optimize the test data generation process and produce test cases with greater consistency, the methodology is combined with a search-based optimization heuristic. Three open-source software frameworks were used in their experimental investigation to test and compare the proposed methodology with two alternative approaches. The experimental results show that the proposed procedure, which was 11.1 percent better than DFAAD and 38.4 percent better than EvoSuite, improved fault detection efficiency. However, in terms of statement and branch coverage, the techniques did not significantly differ. As the process complexity increases, the proposed technique has been capable of detecting more computation-related faults and appears to have improved fault detection capabilities [7].

Machalica et al. [10] created a general predictive test selection strategy in which selections for each update submitted to the consistent system developed a subset of tests to be exercised. Using simple machine learning methods, the strategy learned from a broad dataset of historical test results. The method lowers the overall infrastructure expense of code testing adjustments by a factor of two, thus ensuring that over 95 percent of person test errors and over 99.9% of defective modifications remain reported to developers again. In addition, the approach we present here accounts for the non-determinism, also recognized, of the test results flakiness as a measure [10].

Sutar et al. [16] suggested an approach to find the high potential regression. They created natural language processing test cases from the core validation set by choosing a test case based on its purpose to fit defects. The software created from this approach has helped them to reduce the regression cycle increased our item's explorative effectiveness and increased. This technique also opens the door to new ideas such as automatically creating test cases based on its awareness of the historical defects of the product, current test cases, and the creation of new features [16].

An advanced technique is known as model-based research (MBT) has been revolutionized. Testing starts at the design step in the MBT method and is therefore helpful in the early detection of faults. To create test cases using MBT, the Unified Modeling Language (UML) was used. Using various UML diagrams, there has been a lot of study and suggestions made for MBT. However, the issue with these methods is that in all situations, a single UML diagram such as an action diagram or sequence diagram is not enough to produce test cases. There may be a scenario where it is possible to use several models to produce test cases. Discusses various current MBT techniques used during testing and describes the guidelines for selecting an appropriate UML diagram for test cases to produce. It also addresses how the models can be enhanced by using many UML models or by expanding the models [13].

Lakshminarayana and SureshKumar [22] proposed cuckoo Search and bee colony algorithm (CSBCA) \ to optimizing test cases and generating path convergence in the shortest possible time. Particle Swarm Optimization (PSO), Cuckoo Search (CS), Bee Colony Algorithm (BCA), and Firefly Algorithm were used to compare the performance of the proposed CSBCA to that of existing methods such as Particle Swarm Optimization (PSO), Cuckoo Search (CS), Bee Colony Algorithm (BCA), and Firefly Algorithm (FA). Test data and automated test cases is used to automatically generate test cases that are optimized by using an example of an ATM withdrawal operation. The fitness function is used to select test data values. The test cases were optimized using the proposed method, which required the fewest iterations and time. An experimental analysis revealed that the proposed CSBCA method generates path coverage in 16.4 seconds. In comparison to all other methods, the proposed CSBCA algorithm achieved a higher fitness function value of 0.7 to 1.0 in 65 percent of test cases/test data. In comparison to PSO, CS, FA, and BCA, the proposed CSBCA method produced better results. This method could be used to test the software in a bank ATM. This method will be implemented and tested in the creation of complex software test cases in the future [22].

Mishra et.al. [23] presented a method for path testing using a real-coded genetic algorithm that involves automatically generating test data and optimizing it to test the critical paths for software under test (SUT). For automatic test data generation, real encoding is used, and the best result is a representative test suite that achieves 100 percent path coverage. The proposed real-coded genetic algorithm for path coverage (RCGAPC) in this paper generates a set of inputs for testing a specific software and outperforms by producing effective and efficient results with fewer test data generation counts. The test data is mapped to the corresponding path using a one-to-one injective mapping scheme in the proposed approach, and the most critical path is covered during path testing of specific software. In terms of critical path coverage, it appears to be faster than traditional GA. The proposed method can reduce the number of test data generation required for SUT path testing and provide an optimized Test suite that covers the entire path for a specific software [23].

Based on the related work shown above, their proposed work differs from existing ones as follows. In all previous studies over different years, researchers tried to find effective methods and techniques for extracting test cases automatically from requirements specification through the code or UML. There are a few research works on the aspect of generating test cases from requirements specification especially generate test cases from use case description. Moreover, previous studies did not consider control flow and test adequacy when developing approaches-based testing, which is critical for any qualified test approach.

The main gap in all previous studies is in generate test paths and test cases directly without using modern technology that clarifies the test cases in a clear and detailed way. This study will solve this gap using the NLP to generate the test

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.

cases with more details. Table I shows a set of the most important tools and models proposed in recent years to perform a test automatically, extract test cases in different ways, and compare them.

TABLE I. COMPARISON OF THE MOST IMPORTANT GENERATE TEST CASE APPROACHED

Name of Approach	Description / method used	Reference	Year of Publication
Web Scraper, Machine Learning (ML), And Selenium	Machine learning to predicted test cases	Paul, N, and Tommy, R	2018
Artificial Intelligence (AI) And Machine Learning (ML) Technologies.	Implementation of a prototype that used to generate test cases	Santiago, D et al	2018
Machine Learning[SVM], Branch Coverage, Evo Suite based on (Genetic algorithm), Randoop.	Generate test cases based on two tools: Evo site and randoob	Grano, G et al	2018
NLP [semantic role labeling], Use case specifications and a domain model.	Generate test cases from natural language requirements specification	Wang, C et al	2019
OClgen.	Extract the test cases based on precondition of the use case description	Wang, C et al	2018
Use case modeling for device checks technology (UMTG).	Used an NLP to generate test cases	Wang, C et al	2018
Proposed tool of generating test paths (PTGTP).	Generate test cases from the use case description	Alrawashed, T et al	2019
Machine learning approach	Generate test cases from use case diagram using machine learning	Khalifa et al	2019
A model-based approach	Generate test cases from the data flow component	Jaffari et al	2020
A general predictive test selection strategy	Machine learning tools to generate test cases	Machalica et al	2019
Natural language processing	Natural language processing test cases from the core validation	Sutar et al	2020
Model-based research (MBT)	Create test cases using MBT which is helpful in the early detection of faults	Rapolu	2018
Cuckoo Search and bee colony algorithm (CSBCA)	Generate test cases using Cuckoo Search and bee colony algorithm in the shortest possible time	Lakshminarayana and SureshKumar,	2019
A real-coded genetic algorithm	Generate test cases using a real-coded genetic algorithm	Mishra et.al	2019

III. THE PROPOSED APPROACH FOR DEVELOPING TEST CASES FROM USE CASES

The use case is a simplified overview of the software functionality that is required. Aside from their simplicity, use cases are usually depicted as a diagram. This diagram, however, assists in providing an overview of the use case, but it is only second in importance to the textual description, which provides some information about the actual use case [24]. As a result, textual refinement is required to gain a thorough understanding of such use cases. Cockburn [2] proposed a template that was well-suited for textual refining use cases.

The purpose of the use case description template is to create a comprehensive description of a use case by including all relevant details such as the name, goal, actors, pre-conditions, post-conditions, invariant, main success scenario, variations, extensions, and included use cases. A typical use case description may include the following: Pre-conditions which are the conditions that must hold for the use case to begin. Post-conditions are the conditions that must hold once the use case has completed and the primary flow which is the most frequent scenario or scenarios of the use case. Finally, Alternate and /or exception flows are the scenarios that are less frequent or other than nominal. The exception flows may reference extension points and generally represent flows that are not directly in support of the goals of the primary flow.

An Approach to Generate Test Cases from Use Case description is proposed in this paper that consists of several processes. The input is the use case description of the use case diagram, which is being used as a basis for the approach. In the next step, the use case description of the use case diagram is utilized to extract the necessary information for the development of the control flow graph and NLP table. A control flow graph and NLP table will be generated based on a specific algorithm after that the approach generated test paths based on a specific algorithm. Then, the test cases will be generated from the test paths and NLP table. Fig. 1 illustrates the general framework of the proposed approach.

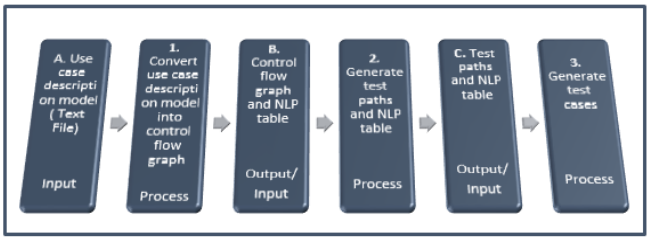

Fig. 1 General framework of the approach to generate test cases from use case description.

A. *Automatically generate control flow graph and NLP table*

The main goal of this stage is to propose an algorithm and tool for automatically creating a control flow graph from relevant details of a use case as described in the main success scenario and extension sections. These details include all scenarios for such a use case as well as the activities' dependencies. Another goal of this stage is to automatically generate an NLP table that will be used later to generate test paths. The input for this stage is the steps on the main scenario and the extensions section of the use case description and output is the control flow graph and NLP table.

A control flow graph is process-oriented that can display all of the paths that can be followed during the execution of a program. Which is consists of vertex's (nodes) that are represent the basic blocks and the edges (arrows) that are induced from the possible flow of the program. The basic block whose leader is the first statement is called the entry

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.

node or first node. The edges are represented by arrows that are responsible for a flow.

The NLP table consists of two columns, the first column is the node number and the second column is the node step description. The proposed algorithm represents each statement (step) in the scenario with a node (N) (Vertex) and transitions between use case description sentences with edges (E) (arrows) to connect the nodes to produce the control flow graph. Since each node considers the next node to be its parent, and the current node to be the previous node's child.

If the node represents a conditional statement, it may have two children. Each step in the use case scenario is represented by a label with a textual stereotype on each node. The proposed approach splits the statement (step) into two or more statements once it contains a conditional statement. The first statement refers to the condition itself; the second to the statement that is executed if a logical expression of the condition is true; and the third to the statement that is executed if a logical expression of the condition is false.

The proposed algorithm creates a parent node with two children nodes in this case. The condition statement is represented by the parent node, while the children represent true and false statements. In addition, the approach in this stage creates the NLP table, which consists of a pair of step numbers and their corresponding description. The details of this algorithm (algorithm .1) are shown in Fig. 2.

Input:
S: Steps of the main scenario and
extension sections of the use case description.
Output:
Control flow graph (N: nodes, E: edges) and
NLP table (N: nodes, s: steps).
1: *R = N*1 {Let the root R of the graph be the first Node}
2: **for** each *S i* do
3: addVertex (*Ni= Si)*
4: Add *Ni, Si* to *NLP* table
5: **if** *S i* {contains a condition} then
6: add Vertex (N i+1 = S i+1)
7: add Vertex (N i+2 = S i+2)
8: addEdge (*Ni, Ni* + 1, *E i*)
9: addEdge (*Ni, Ni* + 2, *E i+i*)
10: **else**
11: addVertex (N i+1 = S i+)
12: addEdge (*Ni, Ni* + 1, *E i*)
13: **end if**
14: **end for**
Note: "addEdge (from Vertex, toVertex): add edge from fromVertex to toVertex
add Vertex (key, data): add vertex key with data "data" "

Fig. 2 Algorithm 1: Convert use case description into control flow graph and NLP table.

B. Automatically generate the test paths

In this stage, the approach generates the paths as follows. We can start with any node and generate a series of test cases. The test sequence is determined by the feasibility of a path from the current vertex to other vertices, and it then decides how to proceed, resulting in the best test sequence in the CFG diagram of the software under test. In this case, optimal means that all decision nodes have been traversed at least once. Each number in the NLP table corresponds to a node in a control flow graph, and the nodes, in turn, correspond to the statements in the main scenario of the use case description. Now, we can explain how an algorithm is used in basis path testing. Starting at the entry and working our way to the end node. Algorithm 2 takes a graph (Control Flow Graph) and the Leader (First Node) first statement and end (End Node) and NLP table as parameters as input. Then the algorithm returns a list of nodes (including the start and end nodes) comprising the path that can be traversed (visited) in the given CFG. If the first node equals the end node the algorithm returns the first path. When no path can be found (not visited), it returns null. The best path with the highest priority is chosen first, and then all of the paths in the control flow graph can be tested continuously. The algorithm is very useful in improving basis path testing. The details of this algorithm (algorithm.2) are shown in Fig. 3.

Input: Control flow graph and NLP table
Output: test paths

1: get_all_test_paths = get_test_paths (start, end, [], [], [])
2: get_test_paths(curr_node, dest_node, visited, path, testpaths)
3: node = nodes [curr_node]
4: visited. Append(curr_node)
5: path. Append(curr_node)
6: **if** *(curr_node ==dest_node)* **then**
7: testpaths. append ("path": list(path))
8: **end if**
9: **for** i in curr_node. getConnections () **do**
10: **if** *(i not in visited)* then
11: get_test_paths (I, des_node, visited, path, testpaths)
12: **end if**
13: **end for**
14: path.pop ()
15: visited.pop ()
16: **if** not path **then**
17: return testpaths
18: **end if**

Fig. 3. Algorithm (2): Convert control flow graph and NLP table into test paths.

C. Automatically generate the test paths

Identifying independent paths in the control flow graph through which software execution flows is known as test path testing. This testing method's main goal is to ensure that all paths are covered and executed. To optimize the test cases using the test paths or basis path testing, four steps are followed: Construct the control flow graph then identify the independent paths finally design test cases from independent paths. In this stage, the approach generates the test cases from test paths and the NLP table. The number of the test case generated by the approach equal to the number of test paths and each test case will consist of several steps equals to the number of nodes of its corresponding test path, each number in the NLP table represents the node with its statements of the control flow graph. Algorithm 3 generates corresponding testing description scenarios from the test path and sets the conditions accordingly. The details of this algorithm shown in Fig. 4.

Input: Test paths and NLP Table
Output: Test case table.

1: Each test case will consist of number of steps equals to the number of nodes of its corresponding test path.
2: For each path in the test paths.
3: Generate its corresponding testing description scenarios.
4: For each step in the testing description scenario
Set the conditions accordingly.

Fig. 4 Algorithm (3): Automatically generate the test paths.

IV. CONCLUSION AND FUTURE WORK

An approach to generate test cases from use case description is proposed in this paper that consists of several

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.

processes. The input is the use case description of the use case diagram, which is being used as a basis for the approach. In the next step, each software summary of UML use cases is utilized to extract the necessary information for the development of the control flow graph and NLP table. A control flow graph and NLP table will be generated based on a specific algorithm after that generate test paths based on a specific algorithm. Then, the test cases will be generated from the test paths and NLP table.

In future works, the proposed approach can be applied to several case studies. Also, the results of the proposed approach can be validated using different coverage criteria.

REFERENCES

[1] T. A. Alrawashed, A. Almomani, A. Althunibat, and A. Tamimi, "An automated approach to generate test cases from use case description model," *Computer Modeling in Engineering &Sciences*, vol. 119, no. 3, pp. 409–42[, 2019.

[2] A. Cockburn, "Structuring use cases with goals," *Journal of object-oriented programming*, vol. 10, no. 5, pp. 56–62, 1997.

[3] G. Grano, T. V. Titov, S. Panichella, and H. C. Gall, "How high will it be? using machine learning models to predict branch coverage in automated testing," in 2018 *IEEE Workshop on Machine Learning Techniques for Software Quality Evaluation (MaLTeSQuE)*. IEEE, 2018, pp. 19–24.

[4] W. S. Humphrey, "The software engineering process: definition and scope," in *Proceedings of the 4th international software process workshop on Representing and enacting the software process*, 1988, pp. 82–83.

[5] IEEE 829-(1983) /IEEE Standard for Software Test Documentation. Available at:https://standards.ieee.org/standard/829-1983.html.

[6] I. S. C. Committee et al., "Ieee standard glossary of software engineering terminology (IEEE std 610.12-1990). Los Alamitos," CA: IEEE Computer Society, vol. 169, p. 132, 1990

[7] A. Jaffari, C.-J. Yoo, and J. Lee, "Automatic test data generation using the activity diagram and search-based technique," *Applied Sciences*, vol. 10, no. 10, p. 3397, 2020.

[8] I. Jovanovic, "Software testing methods and techniques," *The IPSI BgD Transactions on Internet Research*, vol. 30, 2006.

[9] E. M. Khalifa, D. Jawawi, and H. A. Jamil, "An efficient method to generate test cases from UML-use case diagram," *International Journal of Engineering Research and Technology*. ISSN 0974-3154, Volume 12, Number 7 (2019), pp. 1138-1145

[10] M. Machalica, A. Samylkin, M. Porth, and S. Chandra, "Predictive test selection," in 2019 *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 91–100.

[11] N. Paul and R. Tommy, "An approach of automated testing on the web-based platform using machine learning and selenium," in *2018 International Conference on Inventive Research in Computing Applications (ICIRCA)*. IEEE, 2018, pp. 851–856.

[12] J. Radatz, "other. IEEE standard glossary of software engineering terminology," *IEEE-SA Standards Board*, 1990.

[13] R. K. Rapolu, "Selection of UML models for test case generation: A discussion on techniques to generate test cases," 2018.

[14] F. G. C. Ribeiro, C. E. Pereira, A. Rettberg, and M. S. Soares, "Model-based requirements specification of real-time systems with UML, SysML, and Marte," *Software & Systems Modeling*, vol. 17, no. 1, pp. 343–361, 2018.

[15] D. Santiago, T. King, and P. Clarke, "AI-driven test generation: Machines learning from human testers," in 36 th Annual Pacific NW Software Quality Conference. *http://uploads*. pnsqc. org/2018/papers/119-Santiago-AI% 20Driven% 20Test% 20Generation. pdf, 2018.

[16] S. Sutar, R. Kumar, S. Pai, and B. Shwetha, "Regression test cases selection using natural language processing," in *2020 International Conference on Intelligent Engineering and Management (ICIEM)*. IEEE, 2020, pp. 301–305.

[17] C. Wang, F. Pastore, and L. Briand, "Automated generation of constraints from use case specifications to support system testing," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation(ICST)*. IEEE, 2018, pp. 23–33.

[18] C. Wang, F. Pastore, A. Goknil, L. Briand, and Z. Iqbal, "Automatic generation of system test cases from use case specifications," in *Proceedings of the 2015 international symposium on software testing and analysis*, 2015, pp. 385–396.

[19] C. Wang, F. Pastore, A. Goknil, and L. Briand, "Automatic generation of acceptance test cases from use case specifications: an NLP-based approach," *IEEE Transactions on Software Engineering*, 2020.

[20] A. Lawanna, "The theory of software testing," AU Journal of Technology, vol. 16, no. 1, pp.35–40, 2014.

[21] T. Kanan *et al.*, "A review of natural language processing and machine learning tools used to analyze Arabic social media," in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, 2019.

[22] P. Lakshminarayana, and T. V. SureshKumar, "Automatic Generation and Optimization of Test case using Hybrid Cuckoo Search and Bee Colony Algorithm," *Journal of Intelligent Systems,* no. 1, pp. 59-72, 2019.

[23] DB. Mishra, R. Mishra, KN. Das, and AA.Acharya, "Test case generation and optimization for critical path testing using genetic algorithm,". *In Soft computing for problem-solving 2019*. Springer, Singapore, 2019, pp. 67-80.

[24] M. Riebisch, K. Böllert, D. Streitferdt, and I. Philippow, (2002, June). "Extending feature diagrams with UML multiplicities," In *6th World Conference on Integrated Design & Process Technology (IDPT2002)* , Vol. 23, pp. 1-7,2002.

[25] G. Booch, 1999. "UML in action," *Communications of the ACM*, *42*(10), pp.26-28.

[26] B. Hawashin, F. Fotouhi, and T. M. Truta, "A privacy preserving efficient protocol for semantic similarity join using long string attributes," in *Proceedings of the 4th International Workshop on Privacy and Anonymity in the Information Society - PAIS '11*, 2011.

[27] B. Hawashin and A. Mansour, "An efficient agent-based system to extract interests of user groups," in *Proceedings of the World Congress on Engineering and Computer Science*, 2016, vol. 1.

[28] M. Elbes, E. Almaita, T. Alrawashdeh, T. Kanan, S. AlZu'bi, and B. Hawashin, "An indoor localization approach based on deep learning for indoor location-based services," in *2019 IEEE Jordan International Joint Conference on Electrical Engineering and Information Technology (JEEIT)*, 2019.

Authorized licensed use limited to: Masaryk University Brno. Downloaded on July 16,2025 at 19:07:36 UTC from IEEE Xplore. Restrictions apply.