Journal of Software Engineering and Applications, 2021, 14, 257-265
https://www.scirp.org/journal/jsea
ISSN Online: 1945-3124
ISSN Print: 1945-3116

# An Improved Approach for Generating Test Cases during Model-Based Testing Using Tree Traversal Algorithm

**Oluwatolani Achimugu[1*], Philip Achimugu[2], Chinonyelum Nwufoh[3], Sseggujja Husssein[4], Ridwan Kolapo[3], Tolulope Olufemi[3]**

[1]Department of Information and Communication Engineering, Air Force Institute of Technology, Kaduna, Nigeria
[2]Department of Computer Science, Air Force Institute of Technology, Kaduna, Nigeria
[3]Department of Computer Science, Lead City University, Ibadan, Nigeria
[4]Department of Computer Science, Islamic University in Uganda, Kampala, Uganda
Email: *o.achimugu@afit.edu.ng

**How to cite this paper:** Achimugu, O., Achimugu, P., Nwufoh, C., Husssein, S., Kolapo, R. and Olufemi, T. (2021) An Improved Approach for Generating Test Cases during Model-Based Testing Using Tree Traversal Algorithm. *Journal of Software Engineering and Applications*, **14**, 257-265.
https://doi.org/10.4236/jsea.2021.146015

**Received:** April 15, 2021
**Accepted:** June 18, 2021
**Published:** June 21, 2021

## Abstract

During the model-based software testing process, test cases are generated from modeled requirements to conduct acceptance testing. However, existing approaches generate erroneous test cases, lack full coverage criteria and prototype tools. Therefore, the aim of this research is to develop an approach capable of reducing erroneous test case generation based on full coverage criteria and a prototype tool. The method employed was to develop a parser to extract information from the XMI file of a modeling diagram where a tree is constructed and a traversal operation executed on the nodes and edges to generate test cases. The results obtained from the proposed approach showed that 97.35% of the generated test cases were precise and comprehensive enough to conduct testing because 99.01% of all the nodes and edges were fully covered during the traversal operations.

## Keywords

Requirements, Model, Testing, Tree, Traversal

## 1. Introduction

Testing can simply be described as the process of ensuring the behavior of a system meets the requirements or desires of users or stakeholders as specified in the contract documents [1]. It is also meant to ensure requirements conformance. These requirements are usually the expected runtime behaviour of a system under development which can be functional or non-functional. Hence, the major aim of a system testing effort is quality assurance [2].

The focus of this research is model-based testing with precise emphasis on UML diagrams. Model-based testing deals with the process of generating test cases from extracted information of the underlying model for the system under test (SUT). Consequently, an approach for extracting information or artefacts from the underlying models of a SUT to generate test cases is the focus of this research work. Test case generations are the foundation of any model-based testing activity [3] [4]. Therefore, no meaningful model-based testing activity can take place with vaguely generated test cases. Model-based testing can be initiated as soon as the requirements/design documents are ready. This is because, these documents provide the expected input and output of the system under development which together form what is known as test cases. Therefore, the importance of testing in software development cannot be over emphasized. It helps eradicate breaches of contract, trust, or agreement. This accounts for the reason why clients are beginning to request for testing results before accepting or deploying a system in its application domain.

Software testing as a discipline consists of many approaches. However, test case generation is a major activity that cuts across the existing software testing techniques because it provides the basis for conducting unit, system, integration or acceptance testing.

In model-based testing approach, the generation of test cases is derived from the underlying model used to represent user's requirements [5] [6] [7] [8]. These models are usually in the diagrammatic form either as a use case, sequence, activity, class, collaboration, deployment, statechart, or component diagrams from modeling languages like ArgoUML, Rational Rose, or Magic Draw among others. The rest of the paper is structured as follows: Section 2 describes the related work, section 3 presents the proposed approach and section 4 shows the experimental setup, results and discussion while Section 5 concludes the paper and suggests an area for future research.

## 2. Related Works

This section deals with the analysis of existing works in the area of model-based testing process. Accordingly; in [2] [3] [4], the authors utilized various models namely sequence, state and object diagrams to demonstrate UML-based testing process. Their approach was based on tree and graph at various points to represent the extracted artefacts which were traversed to generate test cases but full coverage was not achieved. In [9] [10], authors proposed techniques for generating test cases from activity/sequence diagrams and class/sequence diagrams respectively, but their techniques were not tested for scalability with complex models and inadequate coverage criteria as well as prototype implementation were issues raised in their researches. While [11] presented methods for generating test cases from behavioural diagrams such as sequence and activity but more coverage criteria were required to generate comprehensive test cases. An approach that focuses on optimizing test cases obtained from UML activity and state chart diagrams using Basic Genetic Algorithm (BGA) has been proposed [12]. For generating test cases, both diagrams were converted into their corresponding intermediate graphical forms namely, Activity Diagram Graph (ADG) and State Chart Diagram Graph (SCDG). Both graphs were then combined to form a single graph called, Activity State Chart Diagram Graph (ASCDG). Next, the ASCDG was optimized using BGA to generate the test cases. Limitation of this research has to do with inability to generate test data for large-scale and complex systems and the need to generate test cases based on more coverage criteria. A model-based test case generation approach using ATM and Library systems have been presented [13]. It was observed that, the use of class diagram, use cases and activity diagram has resulted in better coverage of test cases. However, this approach was not implemented and the need to combine the approach with formal specifications Object-Z and OCL is required.

## 3. Proposed Approach

The approach for generating test cases is depicted in **Figure 1**. A modified algorithm capable of extracting key information or artefacts from modeling diagrams is presented. The extracted information is transformed into a tree which is traversed to automatically generate test cases. Trees are special types of graphs which contain sets of Nodes denoted as $V$ and edges $E$ connecting these nodes with no cycles. The first node is usually considered to be the root node while subsequent nodes are known as sub-nodes. When using trees for generating test cases, an input is required. This input is the user's requirements expressed in any of the modeling diagrams. Different modeling tools store their information in various formats. For example, UML stores its information in MDL file while ArgoUML stores its information in XMI file. Therefore, the first task in test case generation is the development of a parser that is capable of extracting all relevant information from the underlying file of a modeling diagram. After parsing

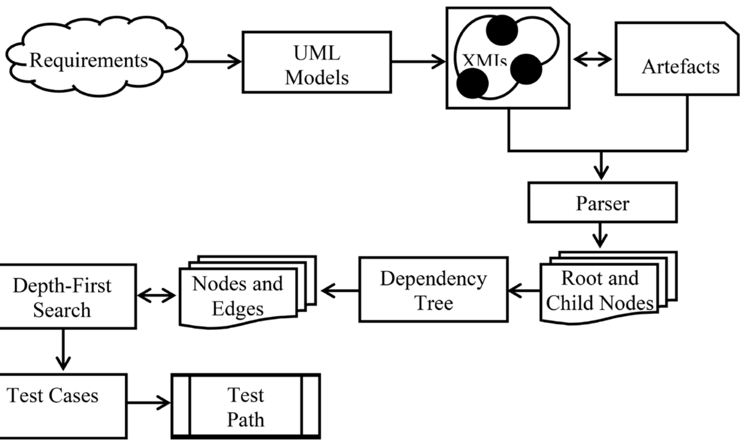


**Figure 1.** Proposed approach.

information from the diagram, an output is generated. The tree is constructed based on the information contained in the XMI file of a given diagram. Pseudo codes 3.1, 3.2 and 3.3 show the logic used in implementing the parser, tree and test case generators respectively. The tree consists of all requirements and their attributes represented using nodes and edges. The depth of a tree is determined based on the number of requirements contained in a diagram. If additional requirements are realized or minimized, the tree can be re-generated to reflect new changes. The tree is then traversed by reading the content of the file to generate test cases.

In developing components of the proposed approach, the improvements made are as follows:

- There is only one root node in any XMI file and all other nodes usually branch from the first node.
- The first child of the root node usually contains some information like date of creation of the XMI file amongst others.
- Every node in the XMI file must have at least one attribute and may have other child nodes.
- Every node has an assigned value, depending on the depth of that node. The root node has a value 1 and the node immediately after it, that is, its first child, has a value 2.
- Multiple nodes that are child nodes to a particular node will have the same value. Therefore, the deeper the node, the higher its value.

In simulating the token flow during model execution, the parser attempts to scan through all the nodes, sub-nodes of the XMI file where the information resides. Once the information is extracted, a dependency tree is generated and test cases are generated based on full coverage criteria (message and transition path).

Redundancy was avoided by ensuring that, each node and edge is visited once and properly marked as 'visited'. This is further enhanced by deleting redundant path during execution. For instance, if there exists a path from 1 to 2, *i.e.*, 1→2, and there also exist another path 1→3→4→2, then 1→2 is deleted. After this step, the auxiliary edges are added to make all end nodes point to the start node. The result path is the test case that satisfies full node and edge coverage criteria. This step is meant to avoid erroneous results. A depth first traversal (DFT) operation was implemented using a recursive algorithm with the aid of a stack data structure. Therefore, recursively applying the ordering rule causes traversals to occur starting from the root of the tree. The Pseudo code for the parser, dependency tree and test case generation processes are depicted in Pseudo code 3.1 - 3.3 respectively.

**Pseudo code 3.1: Parser (Artefacts Extraction Process)**

```
1. INPUT: XMI file;
2. Read the XMI file from the first node;
3. Read the current node;
4. IF the name of the current node is valid THEN;
```

```
5. Increase the value of the depthCounter by 1;
6. Add the name of the node to ElementsList;
7. IF the current node has at least one child node THEN;
8. FOR each child node, extract information;
9. Set current child node as current node;
10. END DO;
11. END IF;
12. ELSE THEN;
13. END.
```

**Pseudo code 3.2: Dependency Tree Generation**

```
1. INPUT: Extracted artefacts;
2. StartElement (String uri, String localName, String tagName, Attributes attr);
3. DefaultMutableTreeNode current = new;
4. DefaultMutableTreeNode(tagName);
5. Base.add(current);base = current;
6. For (int i = 0; i < attr.getLength(); i++) {;
7. DefaultMutableTreeNode currentAtt = new;
8. DefaultMutableTreeNode(attr.getLocalName(i) + "=" + attr.getValue(i));
9. Base.add(currentAtt);
10. EndElement(String namespace uri, String localName, String qName);
11. Base = (DefaultMutableTreeNode) base.getParent();
12. Main(String[] args) {;
13. TreeViewer = new XMLTreeViewer();
14. TreeViewer.xmlSetUp();
T15. TreeViewer.createUI().
```

**Pseudo code 3.3: Test Case Generation**

```
1. INPUT: Dependency Tree;
2. Create String currentTestCase;
3. Set depthCounter = item 1 on ElementList;
4. Create String ArrayList names testCase;
5. FOR each element in ElementList, DO;
6. IF next element is a child of the current element THEN;
7. Add current element's name to currentTestCase;
8. ELSE IF next element and current element are siblings (same level), THEN;
9. Add currentTestCase + current element's name to testCase;
10. Add currentTestCase + next element's name to testCase;
11. Skip the next element;
12. ELSE IF next element is a sibling to current element's parent, THEN;
13. Add currentTestCase + current element's name to testCase;
14. CurrentTestCase = testCase of current element's parent;
15. END ELSE IF;
16. END FOR;
17. Locate testCase with the highest weight (testCase which is the one whose
```

sum of children's weight is greatest; usually the one with highest number of child nodes);

18. Output result;
19. END.

## 4. Experimental Setup

Some diagrams were drawn in ArgoUML for a software application. These diagrams were saved in XMI file extensions and uploaded in the parser. The aim here is to see whether the parser is able to correctly extract the total numbers of artefacts as contained in the XMI file. Then, the total number of correctly extracted artefacts is compared to the total numbers of artefacts contained in the XMI source file to ascertain percentage level of accuracy. Similarly, the extracted artefacts are converted into a dependency tree. Finally, the tree is traversed to generate test cases. The generated test cases were analyzed in terms of test coverage criteria.

### Results and Discussion

Evaluation of the proposed technique was conducted based on accuracy and coverage criteria. Accuracy for this experiment is taken as the number of artefacts correctly retrieved or generated divided by the total number of existing relevant artefacts while coverage criteria is given as the number of nodes and edges visited during the traversal operation divided by the total number of existing nodes and edges. The formula for calculating accuracy is depicted in Equation (1).

$$\text{Accuracy} = \frac{(\text{TP} + \text{TN})}{(\text{TP} + \text{TN} + \text{FP} + \text{FN})} \tag{1}$$

where:

- True Positive (TP): Correctly identified Node.
- False Positive (FP): Incorrectly identified Nodes.
- True Negative (TN): Correctly identified Edges.
- False Negative (FN): Incorrectly identified Edges.

The main idea behind the efficient extraction and generation processes lies in the optimization of the proposed pseudo codes which was aimed at reducing false classifications that culminates in erroneous extractions and generations respectively. As seen from the results, the proposed approach was accurately able to extract complete information from the XMI file (Table 1) which tallied with the number of elements in the source file. Furthermore, the percentage of coverage criteria was calculated and from the results displayed in Table 2, all the nodes and edges were fully visited during the traversal operation giving rise to 99.01% coverage. The coverage criteria results show that, test cases were correctly generated with respect to the number of visited nodes and edges. From the results displayed in Table 1 and Table 2, it is easy to conclude that, the extracted artefacts

Table 1. Results of the proposed approach.

| XMI file of Diagrams | Total Number of Relevant Artefacts | Number of Correctly Extracted Artefacts | Number of Correctly Generated Test cases | TP | FP | TN | FN | Accuracy (%) |
|---|---|---|---|---|---|---|---|---|
| Activity | 30 | 30 | 28 | 20 | 0 | 10 | 0 | 100.0 |
| Component | 28 | 28 | 26 | 18 | 0 | 7 | 0 | 100.0 |
| Class | 32 | 32 | 29 | 29 | 0 | 2 | 1 | 96.88 |
| Sequence | 39 | 39 | 37 | 34 | 0 | 4 | 1 | 97.43 |
| State chart | 46 | 41 | 45 | 38 | 1 | 6 | 1 | 95.65 |
| Use case | 34 | 34 | 29 | 25 | 0 | 7 | 2 | 94.12 |
| **Overall** | | | | | | | | **97.35** |

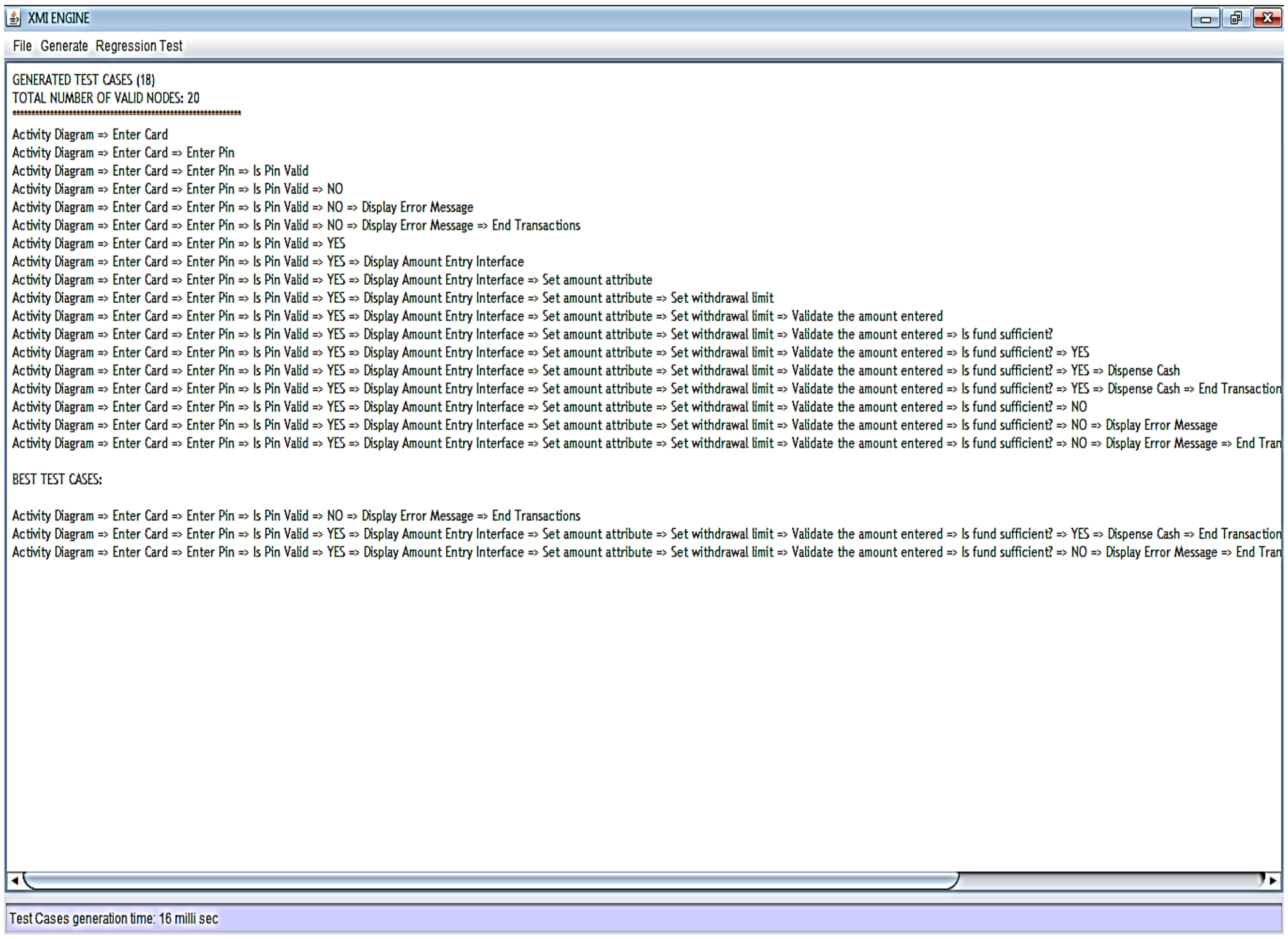
Table 2. Coverage criteria.

| XMIs of software applications | Number of Nodes Visited | Number of Edges Visited | Coverage Criteria (%) |
|---|---|---|---|
| Activity | 20 | 10 | 100.0 |
| Component | 18 | 10 | 100.0 |
| Class | 25 | 7 | 100.0 |
| Sequence | 34 | 5 | 100.0 |
| State chart | 37 | 9 | 100.0 |
| Use case | 25 | 7 | 94.12 |
| **Overall** | | | **99.01** |

are entirely in agreement with the contents of the XMI file. The generated test cases were based on full-coverage criteria as defined for the proposed approach with reduced computation time and generation of best test path as seen in Figure 2. In other words, test cases were generated based on all the elements and descriptive links or attributes of the various XMI file. It is worthy to note that, the test cases generated by the proposed approach were comprehensive and has the capacity of enhancing acceptance test.

## 5. Conclusion and Future Work

In software development, models are used to visualize user's requirements which begin from planning to implementation stages of the system development life cycle phases. The modeling diagrams are also used to provide a glimpse of system functionalities to clients or users. They offer an overview of what is expected to be coded by the developer or programmer; enhances the documentation of system behaviour, operating procedures, and helps in clarifying the requirements to undergo the test. Model-based testing has caught the attention of most testing engineers because testing is performed based on the specified requirements that are articulated via a modeling diagram. A parser was developed to read XMI files



```
GENERATED TEST CASES (18)
TOTAL NUMBER OF VALID NODES: 20
*****************************************************************
Activity Diagram => Enter Card
Activity Diagram => Enter Card => Enter Pin
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => NO
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => NO => Display Error Message
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => NO => Display Error Message => End Transactions
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient?
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => YES
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => YES => Dispense Cash
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => YES => Dispense Cash => End Transaction
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => NO
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => NO => Display Error Message
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => NO => Display Error Message => End Tran

BEST TEST CASES:

Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => NO => Display Error Message => End Transactions
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => YES => Dispense Cash => End Transaction
Activity Diagram => Enter Card => Enter Pin => Is Pin Valid => YES => Display Amount Entry Interface => Set amount attribute => Set withdrawal limit => Validate the amount entered => Is fund sufficient? => NO => Display Error Message => End Tran

Test Cases generation time: 16 milli sec
```

**Figure 2.** Prototype tool implementation.

or extract the information contained in the file to generate test cases. This technique basically utilizes an Application Programming Interface (API) to form an in-memory tree representation of the XMI tags that provides all the elements, transitions, entities, relationships, and sequences of events that are traversed in the process. The parser was implemented with Java programming language. The parsed XMI files serve as input for generating test cases. Furthermore, adequate test coverage criteria were utilized during test case generation. As a result, the entire test paths involved from top to bottom of the nodes of the tree are visited exactly once. The reliability of the generated test cases depends on the completeness of the information extracted from the nodes and edges which was achieved in this research. The nodes store information such as events between two or more objects or entities, the sending object and receiving object as well as the descriptions of attributes. In the future, it will be necessary to test the proposed approach for scalability with models of ultra-large-scale systems.

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Li, X., He, T. and Xiong, J. (2013) Extenics-Based Test Case Generation for UML Activity Diagram. *Procedia Computer Science*, **17**, 1186-1193. https://doi.org/10.1016/j.procs.2013.05.151

[2] Jagtap, S., Gawade, V., Pawar, R., Shendge, S. and Avhad, P. (2016) Generate Test Cases From UML Use Case and State Chart Diagrams. *International Research Journal of Engineering and Technology*, **3**, 873-881.

[3] Shanthi, A.V.K. and Kumar, D.G.M. (2011) Automated Test Cases Generation for Object Oriented Software. *Indian Journal of Computer Science and Engineering*, **2**, 543-546.

[4] Sawant, V. and Shah, K. (2011) Construction of Test Cases from UML Models. In: Shah, K., Lakshmi Gorty, V.R. and Phirke, A., Eds., *Technology Systems and Management*, Springer, Berlin, Heidelberg, 61-68. https://doi.org/10.1007/978-3-642-20209-4_9

[5] Pachauri, A. (2013) Automated Test Data Generation for Branch Testing Using Genetic Algorithm: An Improved Approach Using Branch Ordering, Memory and Elitism. *Journal of Systems and Software*, **86**, 1191-1208. https://doi.org/10.1016/j.jss.2012.11.045

[6] Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W. and Zhu, H. (2013) An Orchestrated Survey on Automated Software Test Case Generation. *The Journal of Systems and Software*, **86**, 1978-2001. https://doi.org/10.1016/j.jss.2013.02.061

[7] Kaur, G. and Bawa, S. (2013) A Survey of Requirement Prioritization Methods. *International Journal of Engineering, Research and Technology*, **2**, 958-962.

[8] Kaur, A. and Vig, V. (2018) Automatic Test Case Generation through Collaboration Diagram: A Case Study. *International Journal of Systems Assurance Engineering and Management*, **9**, 362-376. https://doi.org/10.1007/s13198-017-0675-8

[9] Septian, I., Alianto, R.S., and Gaol, F.L. (2017) Automated Test Case Generation from UML Activity Diagram and Sequence Diagram Using Depth First Search Algorithm. *Procedia Computer Science*, **116**, 629-637. https://doi.org/10.1016/j.procs.2017.10.029

[10] Shah, S.A.A., Shahzad, R.K., Bukhari, S.S.A. and Humayun, M. (2016) Automated Test Case Generation Using UML Class & Sequence Diagram. *British Journal of Applied Science and Technology*, **15**, 1-12. https://doi.org/10.9734/BJAST/2016/24860

[11] Swain, R., Panthi, V., Behera, P.K. and Mohapatra, D.P. (2012) Automatic Test Case Generation from UML State Chart Diagram. *International Journal of Computer Applications*, **42**, 26-36. https://doi.org/10.5120/5705-7756

[12] Sahoo, R.K., Derbali, M., Jerbi, H., Van Thang, D., Kumar, P. and Sahoo, S. (2021) Test Case Generation from UML-Diagrams Using Genetic Algorithm. *CMC-Computers Materials & Continua*, **67**, 2321-2336. https://doi.org/10.32604/cmc.2021.013014

[13] Arora, P.K. and Bhatia, R. (2018) Agent-Based Regression Test Case Generation using Class Diagram, Use Cases and Activity Diagram. *Procedia Computer Science*, **125**, 747-753. https://doi.org/10.1016/j.procs.2017.12.096