An Enhanced Multiview Test Case Generation Technique for Object-oriented Software using Class and Activity Diagrams

James Maina Mburu, Geoffrey Muchiri Muketha, Aaron Mogeni Oirere

Abstract: Software testing is one of the vital steps in software development life cycle. Test case generation is the first process in software testing which takes a lot of time, cost and effort to build an effective product from the start. Automatic test case generation is the best way to address this issue and model-based test case generation approach would be suitable for this automation process. One way to generate test cases automatically is by generating test cases from Unified Modeling Language (UML) models. The challenge with the existing test case generation techniques using UML models is that they provide a single view, meaning that the techniques capture a single aspect of the system, such as structural or behavioral but not both. In this paper, we have successfully developed a technique that automatically generates test cases which capture both structural and behavioral views of the system. These test cases can help to discover software faults early in the software development cycle. Finally, we conducted an experiment by comparing our technique with a manual process. The results show that the proposed technique can produce same test cases as manually writing test cases of the same system model but this technique saves a lot of time, effort and cost as well.

Keywords: Behavioral models, Structural models, Test case generation, UML.

I. INTRODUCTION

Software testing is one of the vital phases of Software Development Life Cycle (SDLC) and about 50% of the time taken to deliver a product is spent on testing. The important objective of testing is failure detection and as the systems are becoming huge and complex, the need to get an effective testing mechanism has become an essential part of the SDLC process [1]. In SDLC the testing process involves four steps namely: test case generation, test case selection, test case execution, and test case evaluation [1]. Test case generation is a very vital step in software testing, as it plays a key role on the efficiency and effectiveness of software test, however, it takes most of the effort in testing process [2]. Test cases can be produced automatically from source code or visual software model such as Unified Modeling Language (UML), Data Flow Diagram (DFD), or Entity Relationship Diagram (ERD) [3]. Using UML approach, test cases are created during analysis or design stage. Test case generation from design specifications has the added advantage of allowing test cases to be available before coding in the software development cycle [4]. An automatic test case generation from code is ineffective since some aspects of program behavior like state behavior is very hard to test based on code [5]. Even though each type of UML diagram offers a view of the software, there is a limitation to each type of model in generating test cases [1]. The limitation of each of the UML model is that they offer a single view of the system, either structural or behavioral [6]. For example, although the sequence diagram effectively captures messages between objects, it provides a behavioral view. On the other hand, the class diagram captures information about relationship between objects but only provides a structural view of the system [6]. A number of studies show that, there exist techniques for generation of test cases using UML structural models [7], [8], [9] and behavioral models [10], [11], [12]. These techniques generate test cases that depend on either structural or the behavioural view of the system based on UML model. However, there is no proof of test case generation technique that combines both behavioral and structural view of the system. Test cases generated from an individual UML diagram are not effective as the test case generated depend on a single view of a system [13]. The study also reveals that, there are existing test case generation techniques that are integrated with more than one UML model [3], [14], [15], but concentrate on combination of behavioral models hence test case generated still depend on single view of the system. There also techniques that use intermediate forms during test case generation [9], [10], [11] this makes automation difficult. Finally, there are test case generation techniques that are not automated [14], [15] therefore, a lot of effort, time and cost are taken to generate test cases. In this paper, we propose the design and implementation of an enhanced multiview test case generation technique (MUTCASGenerator). The effectiveness of the MUTCASGenerator was analysed through an experiment by comparing the technique with a manual process. The rest of the paper is organized as follows: section 2, describes the UML basic concepts and model-based testing. Section 3 describes survey on object-oriented test case generation techniques using UML structural models, behavioral models and UML combinational models. Section 4 covers the discussion of MUTCASGenerator requirements, procedure, design, test case design and algorithm. Section 5 covers the implementation of the MUTCASGenerator. Section 6 covers the experimental evaluation and finally Section 7 describes the conclusion and future work.

Revised Manuscript Received on October 20, 2020.

***** Correspondence Author
James Maina Mburu*, Department of Information Technology, Murang'a University of Technology, Muranga, Kenya. Email: jmburu48@gmail.com
Geoffrey Muchiri Muketha, Department of Computer Science, Murang'a University of Technology, Muranga, Kenya. Email: gmuchiri@mut.ac.ke
Aaron Mogeni Oirere, Department of Computer Science, Murang'a University of Technology, Muranga, Kenya. Email: amogeni@mut.ac.ke

II. BASIC CONCEPTS AND MODEL-BASED TESTING

UML is a commonly accepted set of notations for modeling object-oriented system and was released by Object Management Group (OMG) in 1997 [16]. UML is used for designing, modeling and documenting the object-oriented software systems. UML offers numerous diagrams to describe particular features of software artifacts. These diagrams can be categorized depending on whether they are intended to describe structural or behavioral aspects of systems [8].

The view of the system can be either structural view or behavioral view. The UML structural view describes the kinds of objects (or classes), that are vital to understand the working of a system and its implementation. Structural view also captures the associations among the classes (objects). The structural view includes object diagram, class diagram and composite structure diagrams. The behavioral view captures how objects intermingle with each other to realize the system behavior. The system behavior captures the dynamic behavior of the system. Behavioral view includes different diagrams such as Activity Diagram, State Chart Diagram, Sequence Diagram and Communication Diagram [6]. Classification of UML diagrams is presented in fig.1 [6].

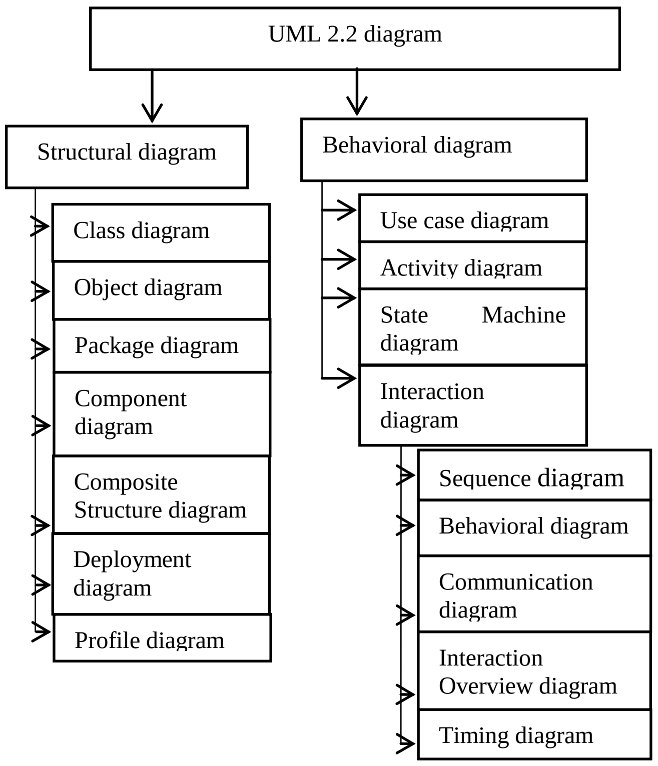


Fig. 1.Classification of UML diagrams

Model-based testing (MBT) is an automatic generation of concrete test cases from abstract formal models and their execution [17]. In MBT, test cases are created using the models, which allow the tester to appreciate the software in a better way and could acquire the test information with simple processing of models [1]. Models reserve vital information from requirement specifications and are the basis for final implementation [18]. MBT is generally done at the design phase and test cases derived helps in early detection of faults in the software. Model-based testing approach is presented in fig.2 [10].

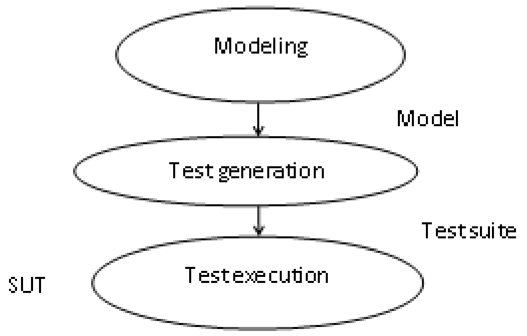


Fig. 2.Model-based testing testing

III. EXISTING TEST CASE GENERA TECHNIQUES USING UML MODELS

Test case generation is the process of creating test cases. It is the first phase of testing and is vital in creating an effective product [1]. The following section presents existing test case generation techniques using UML structural models, behavioral models and UML combinational models.

A. Test Case Generation Techniques using UML Structural Models

Prasanna, Chandran and Suberi [7] presented the class diagram by using data flow approach. Data variables and member methods in this approach were retrieved from class diagram and also used data flow technique to generate test cases. A directed flow graph was created which assists in expressing use pair approach. This approach is able to generate test cases automatically. Shanthi [8] Proposed an Automated test cases Generation for Object-oriented Software. The test cases are generated automatically from UML class diagram using Genetic algorithm and Binary Search Techniques. Information is extracted from UML class diagram and mapped to form a tree structure then genetic algorithm is applied to discover all patterns and finally depth first search technique is implemented to form a binary tree to represent the knowledge i.e. test cases. This approach generates novel automated test cases. Prasanna, Chandran and Suberi [9] proposed Automated Test Case for Object-oriented Systems using UML Object Diagrams. In their work, an object diagram of a cell phone system is drawn using Rational Rose software. Objects diagrams are stored as files for reference and then parsed to derive the graph. This graph is traversed to generate valid and invalid test cases. Test cases are generated automatically hence saving time, cost and effort.

B. Test Case Generation Techniques using UML Behavioral Models

Monim and Nor [10] proposed an automated test Case generation tool using UML activity diagram. The overall development processes start from developing UML activity diagram for the system under test. The model is converted to an intermediate model form (activity graph). The Depth-First Search algorithm is applied on the graph to produce test sequences used to generate test cases. This approach is able to generate test cases automatically hence saving, reducing human intervention and error free output.

Published By:
Blue Eyes Intelligence Engineering
and Sciences Publication

Singh and Preeti [11] proposed an automatic test generation for object-oriented system using activity diagram. In this approach activity graph is generated from the Activity diagram. Then an algorithm is used to traverse activity graph in order to extract all the possible test paths. This technique is also automated.

Fernando and Glaucia [12] suggested an automatic approach (EasyTest) to create test cases from UML activity diagrams using gray-box techniques. In their work they used activity dependency graph and activity dependency tree for sequencing the test path. They developed tool provides automatic support for the test case generation and applying test cases phases.

C.Test Case Generation Techniques using UML Combinational Models

Test case generation techniques using UML Combinational Models involves generation of test cases from more than one UML model or integration of several UML models.

Meiliana [3] proposed an approach for Automatic Test Case Generation from Activity Diagram and Sequence Diagram using Depth Search Algorithm. In this approach, the activity diagram (AD) is converted into activity diagram graph (ADG) and sequence diagram (SD) is converted into sequence diagram graph (SDG). Then a graph called System Testing Graph (SYTG) is formed from by combining the activity diagram and sequence diagram graphs. The necessary information to form the test cases is pre-stored in this graph. The test cases generated present a behavioral view of the system.

Jagtap et al. [14] proposed a technique for generating test cases from UML use case and state chart diagrams. The design model was constructed using ArgoUML tool which support XMI file format. The shared model approach was used for test case generation. The same model was used for extracting artifacts as well as for test case generation. This approach is not automated therefore much time, effort and cost is spent to generate test cases.

Khurana and Chilla [15] proposed an approach for test case generation and optimization using both sequence diagram and state chart diagram. In this approach, a SUT is converted into a graph called System Testing Graph (SYTG) which is formed after integration of state chart graph and Sequence graph. Genetic algorithm is applied on this graph to generate and optimize the test cases automatically based on a coverage criterion and a fault model. This technique is not automated and requires to be integrated with one or more UML diagrams in order to handle all types of errors.

IV. MUTCASGENERATOR

In this paper, a new technique (MUTCASGenerator) is proposed in which an algorithm is applied to create all possible test cases. The technique is applied on the combination of class and activity diagrams. The section discusses requirements, technique flow chart and design and algorithm.

A.MUTCASGenerator Requirements

From the above review it has been observed that, existing test case generation techniques for object-oriented software using UML models reviewed in this paper were found to have several limitations. Firstly, several techniques handle only one UML diagram hence the test cases generated present single view. Secondly, there are existing techniques that use more than one UML diagram, but concentrate on the combination of behavioral models only hence the test cases generated still depend on one single view. Thirdly, these techniques uses intermediate forms during test case generation, this makes automation difficult. Finally, some of the existing techniques are not automated. Therefore, a lot of effort, time and cost are taken to generate test cases. Therefore, future studies should focus on test case generation techniques that capture both structural and behavioral view of the system and should generate test cases automatically without using any intermediate in order to increase the level of automation in test case generation domain.

Existing object-oriented test case generation techniques were compared on the basis of characteristics given below which are considered as researchers view point while developing a test case generation technique.

C1: UML diagrams used by the approach.
C2: Views provided by the test cases generated by the approach.
C3: Use of intermediate forms during test case generation
C4: Test case generation approach used i.e. manual or automated

By analyzing the existing techniques, researchers have instigated desired characteristics which need to be considered while developing test case generation technique;

1. Selection of UML diagrams need to be considered while developing test case generation technique in order in order to capture both structural and behavioral views of the system.
2. There is need to use less complex intermediate form or no intermediate form so as not make test automation difficult.
3. The process of generating test cases need to be automated in order to increase effectiveness

B. Steps to be followed for the MUTCASGenerator

Following are the series of steps that should be followed in order to generate the test cases.

1. Using visual paradigm tool construct a class diagram of a system
2. From the class diagram created, derive an activity diagram
3. Using visual paradigm tool convert diagram into XML file
4. Input XML file to a Parser
5. Generate test cases

All the test case scenario and end of the application can be attained using step 5.

C.MUTCASGenerator Procedure

Fig. 4 shows the flowchart for the technique showing steps to be followed

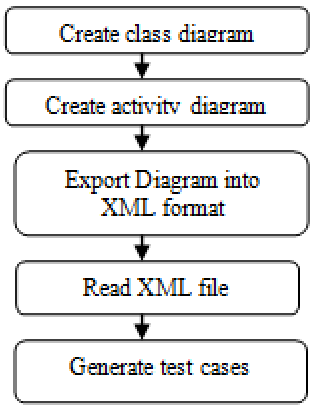


Fig. 3.Flowchart of MUTCASGenerator

D. Description of Methodology

The approach begins by creating a class diagram of a system using visual Paradigm tool by defining class name, attributes, methods and links between them. Then, activity diagram is derived from class diagram by enriching it with class attributes to name control flows of the activity diagram. The developed diagrams are then converted into XML format using visual Paradigm tool so as to store all information related to generated diagrams in the form of tags. A parser (MUTCASExtractor) is then used to read the XML file to extract test cases. Finally, an experimental evaluation is conducted to analyze the effectiveness of the proposed technique.

E. MUTCASGenerator Technique Design

This section presents the design of MUTCASGenerator using class and activity diagrams. While presenting this approach, the researcher has taken care of all points discussed above. This makes the proposed technique totally distinct from the previous approaches.

Fig.5 shows the design of MUTCASGenerator technique

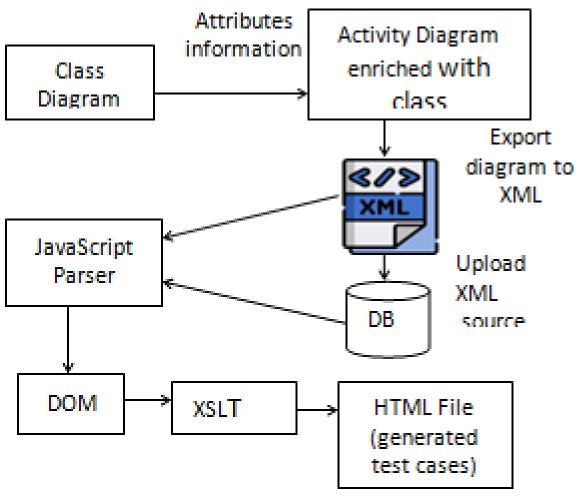


Fig. 4.Design of MUTCASGenerator

F. Test Case Design

Test cases are built using specifications and requirements document, i.e., what the system needs to perform. A test case is a triplet (I, S, O) where I is the data input to the system, S is the state or the condition of the system to which the data is input, and O is the expected output obtained from the system [5].

Table-I: Showing desired test case output design

Test Caseno	Test Condition	Input	Expected Output	Test Case Status
1				
2				
3				

In the table (I), test condition is the variable or the left operand, input is the value or right operand, expected outcome is the result after evaluating a condition, test case status holds results after testing a variable i.e. returns either pass or fail . The test caseno keeps track of number of test cases evaluated.

G. Description of the Multiview Test Case Generation Algorithm (MUTCSGA)

Input to the algorithm is the XML file while the output to the algorithm is the set of test cases. The algorithm transverse the decision node in sequential order and as the decision node is transversed, the test variable is found which is an attribute from class diagram plus assigned integer value (input). The test variable is parsed into 3 constituents' i.e. the left operand, operator and the right operand. In this study the researcher considered the binary relational operators such as ==, !=, <, >, <=,>=.

How MUTCSGA works

If the predicate has >= operator, the algorithm outputs two test cases as (testcaseno, testcondtion, input, expectedoutput, testcasestatus). The other one is; (testcaseno, testcondition, input+1, expectedoutput, testcasestatus) and the testcaseno is likewise increased with a value. If the predicate has <= operator, the algorithm outputs two test cases as (testcaseno, testcondition, input, expectedoutput, testcasestatus). The other one is; (testcaseno, testcondtion, input-1, expected output, testcasestatus) and the testcaseno is equally increased with a value. If the predicate has >operator, the algorithm display one test case as (testcaseno, testcondition, input+1, expectedoutput, testcasestatus) and the testcaseno is correspondingly increased with a value. If the predicate has < operator, the algorithm output one test case as (testcaseno, testcondition, input-1, expected output, testcasestatus) and the testcaseno is also increased with a value. If predicate has != operator, the algorithm output one test case as (testcaseno, testcondition, input-1, expected output, testcasestatus). The testcaseno is equally incremented with a value. Other test case is (testcaseno, testcondition, testinput+1, expected output, testcasestatus). If the predicate has == operator, the algorithm output test cases as (testcaseno, testcondition, input, expectedoutput, testcasestatus). The test testcaseno is equally increased with a value.

V. IMPLEMENTATION

The design models were constructed using visual paradigm 16.0 tool. Visual paradigm was also used to export the models into an XML files format.

In this study, 50 models were designed. Javascript was used to develop a parser to read the XML file, XSLT was used to transform the xml to html file that contains generated test cases. PHP programming language was used to develop the interfaces and outputs of the technique. The following are steps involved in this phase

A.Creating Class Diagram

Visual Paradigm (Standard Edition) tool was used to create class diagrams. Class diagrams was first developed by defining Class name, attributes, methods and links between them. Fig.6 shows a case study of class diagram of Hotel Booking System [19].

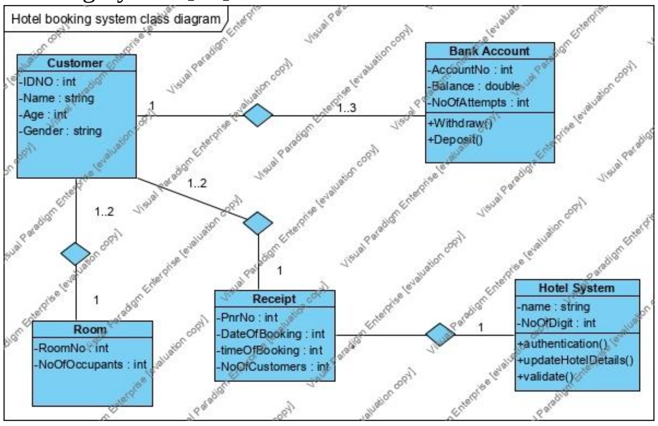


Fig. 5.Screenshot of Hotel Booking System class diagram

B.Creating Activity Diagram

The 'new diagram option' from generated class diagram was used to get a complete activity diagram. Class attributes were extracted from Class diagram and then used in naming control flows presented by activity diagram in a system. Fig.7 shows the activity diagram for Hotel booking system [19].

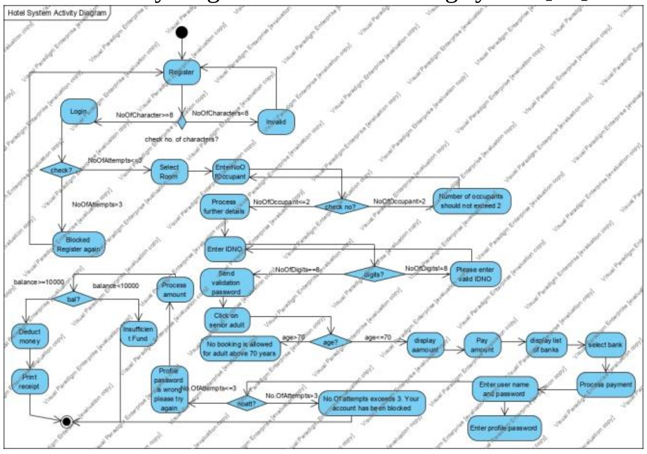
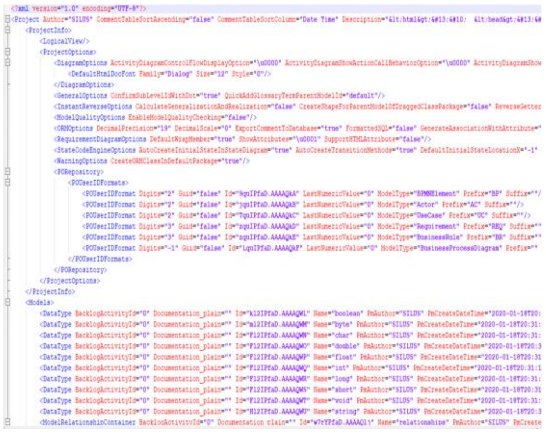


Fig. 6.Screenshot of Hotel Booking System Activity Diagram

C.Exporting Diagrams into XML Format

The developed diagrams were then exported into XML format using Visual Paradigm .Fig.8 shows a screen shot of XML file generated for Hotel booking system model.

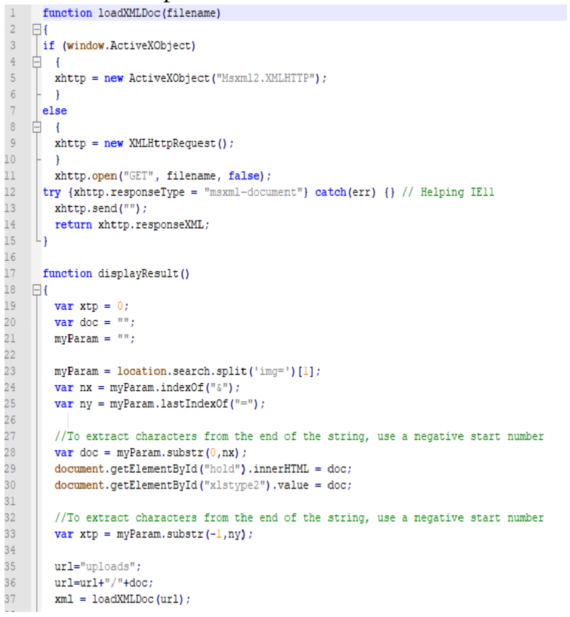


```
<?xml version="1.0" encoding="UTF-8"?>
<Project Author="SILUS" CommentTableSortAscending="false" CommentTableSortColumn="Date Time" Description="&lt;html&gt;&#13;&#10;  &lt;head&gt;&#13;&#
    <ProjectInfo>
        <LogicalView/>
        <ProjectOptions>
            <DiagramOptions ActivityDiagramControlFlowDisplayOption="\u0000" ActivityDiagramShowActionCallBehaviorOption="\u0000" ActivityDiagramShow
                <DefaultHtmlDocFont Family="Dialog" Size="12" Style="0"/>
            </DiagramOptions>
            <GeneralOptions ConfirmSubLevelIdWithDot="true" QuickAddGlossaryTermParentModelId="default"/>
            <InstantReverseOptions CalculateGeneralizationAndRealization="false" CreateShapeForParentModelOfDraggedClassPackage="false" ReverseGetter
            <ModelQualityOptions EnableModelQualityChecking="false"/>
            <ORMOptions DecimalPrecision="19" DecimalScale="0" ExportCommentToDatabase="true" FormattedSQL="false" GenerateAssociationWithAttribute="
            <RequirementDiagramOptions DefaultWrapMember="true" ShowAttributes="\u0001" SupportHTMLAttribute="false"/>
            <StateCodeEngineOptions AutoCreateInitialStateInStateDiagram="true" AutoCreateTransitionMethods="true" DefaultInitialStateLocationX="-1"
            <WarningOptions CreateORMClassInDefaultPackage="true"/>
            <PORepository>
                <POUserIDFormats>
                    <POUserIDFormat Digits="2" Guid="false" Id="9quIPfaD.AAAAQkA" LastNumericValue="0" ModelType="BPMNElement" Prefix="BP" Suffix=""/
                    <POUserIDFormat Digits="2" Guid="false" Id="jquIPfaD.AAAAQkB" LastNumericValue="0" ModelType="Actor" Prefix="AC" Suffix=""/>
                    <POUserIDFormat Digits="2" Guid="false" Id="TquIPfaD.AAAAQkC" LastNumericValue="0" ModelType="UseCase" Prefix="UC" Suffix=""/>
                    <POUserIDFormat Digits="3" Guid="false" Id="zquIPfaD.AAAAQkD" LastNumericValue="0" ModelType="Requirement" Prefix="REQ" Suffix=""
                    <POUserIDFormat Digits="3" Guid="false" Id="zquIPfaD.AAAAQkE" LastNumericValue="0" ModelType="BusinessRule" Prefix="BR" Suffix=""
                    <POUserIDFormat Digits="-1" Guid="false" Id="LquIPfaD.AAAAQkF" LastNumericValue="0" ModelType="BusinessProcessDiagram" Prefix=""
                </POUserIDFormats>
            </PORepository>
        </ProjectOptions>
    </ProjectInfo>
    <Models>
        <DataType BacklogActivityId="0" Documentation_plain="" Id="k12IPfaD.AAAAQWL" Name="boolean" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:
        <DataType BacklogActivityId="0" Documentation_plain="" Id="m12IPfaD.AAAAQWM" Name="byte" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31:
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWN" Name="char" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31:
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWO" Name="double" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:3
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWP" Name="float" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWQ" Name="int" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31:1
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWR" Name="long" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31:
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWS" Name="short" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31
        <DataType BacklogActivityId="0" Documentation_plain="" Id="F12IPfaD.AAAAQWT" Name="void" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:31:
        <DataType BacklogActivityId="0" Documentation_plain="" Id="H12IPfaD.AAAAQWU" Name="string" PmAuthor="SILUS" PmCreateDateTime="2020-01-18T20:3
        <ModelRelationshipContainer BacklogActivityId="0" Documentation_plain="" Id="w7rYPfaD.AAAQl}" Name="relationships" PmAuthor="SILUS" PmCreate
```

Fig. 7.Screenshot of generated XML file

D.Reading XML File

The XML file was parsed using a multiview test case extractor (MUTCASExtractor). MUTCASExtractor was used to extract the required information. Fig.9 below shows a screen shot JavaScript code for MUTCASExractor.



```
function loadXMLDoc(filename)
{
if (window.ActiveXObject)
  {
  xhttp = new ActiveXObject("Msxml2.XMLHTTP");
  }
else
  {
  xhttp = new XMLHttpRequest();
  }
  xhttp.open("GET", filename, false);
try {xhttp.responseType = "msxml-document"} catch(err) {} // Helping IE11
  xhttp.send("");
  return xhttp.responseXML;
}

function displayResult()
{
  var xtp = 0;
  var doc = "";
  myParam = "";

  myParam = location.search.split('img=')[1];
  var nx = myParam.indexOf("&");
  var ny = myParam.lastIndexOf("=");

  //To extract characters from the end of the string, use a negative start number
  var doc = myParam.substr(0,nx);
  document.getElementById("hold").innerHTML = doc;
  document.getElementById("xlstype2").value = doc;

  //To extract characters from the end of the string, use a negative start number
  var xtp = myParam.substr(-1,ny);

  url="uploads";
  url=url+"/"+doc;
  xml = loadXMLDoc(url);
```

Fig. 8.JavaScript code for MUTCASExtractor

The researcher developed an interface of the MUTCASGenerator, using PHP programming language to facilitate interaction with the users. Fig.10 shows a screenshot of the interface of the MUTCASGenerator.

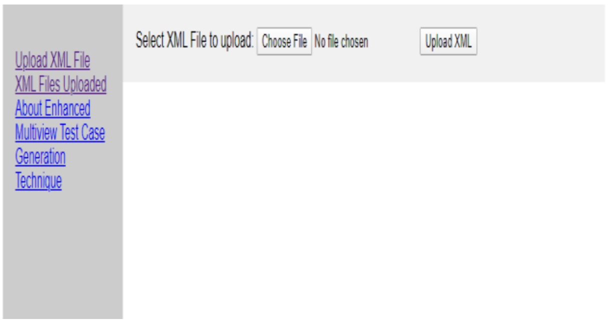
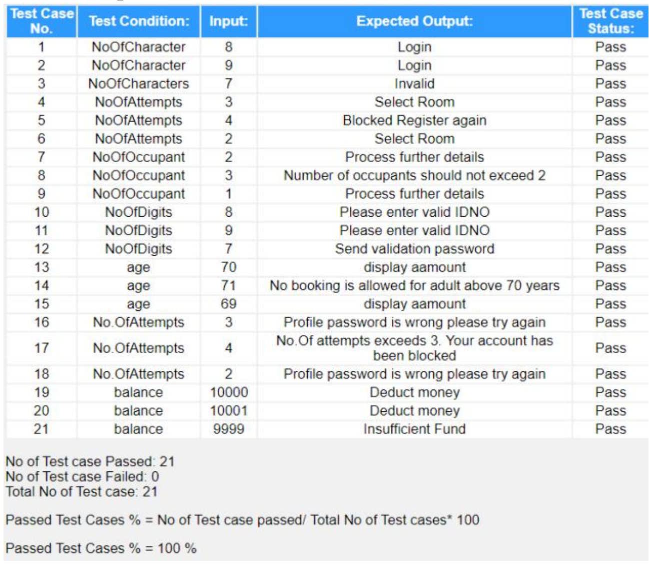

Select XML File to upload:
Choose File
No file chosen
Upload XML
Upload XML File
XML Files Uploaded
About Enhanced Multiview Test Case Generation Technique

Fig. 9.Screenshot of interface of the MUTCASGenerator

Fig.11 shows a screen shot of generated test cases after parsing XML code of Hotel booking system model. There are 3 test scenarios i.e. the first test case shows test case precisely at the extremes of input domain e.g. 8. The second shows test case precisely just above limits of input domain e.g. 9.The third shows the test case precisely just below the limits of input domain e.g. 7. Test condition holds attribute, input holds the value, expected outcome holds result after evaluating a condition, and test case status holds results after testing a variable i.e. returns either pass or fail. The test caseno keeps track of number of test cases evaluated.



Test Case No.	Test Condition:	Input:	Expected Output:	Test Case Status:
1	NoOfCharacter	8	Login	Pass
2	NoOfCharacter	9	Login	Pass
3	NoOfCharacters	7	Invalid	Pass
4	NoOfAttempts	3	Select Room	Pass
5	NoOfAttempts	4	Blocked Register again	Pass
6	NoOfAttempts	2	Select Room	Pass
7	NoOfOccupant	2	Process further details	Pass
8	NoOfOccupant	3	Number of occupants should not exceed 2	Pass
9	NoOfOccupant	1	Process further details	Pass
10	NoOfDigits	8	Please enter valid IDNO	Pass
11	NoOfDigits	9	Please enter valid IDNO	Pass
12	NoOfDigits	7	Send validation password	Pass
13	age	70	display aamount	Pass
14	age	71	No booking is allowed for adult above 70 years	Pass
15	age	69	display aamount	Pass
16	No.OfAttempts	3	Profile password is wrong please try again	Pass
17	No.OfAttempts	4	No.Of attempts exceeds 3. Your account has been blocked	Pass
18	No.OfAttempts	2	Profile password is wrong please try again	Pass
19	balance	10000	Deduct money	Pass
20	balance	10001	Deduct money	Pass
21	balance	9999	Insufficient Fund	Pass

No of Test case Passed: 21
No of Test case Failed: 0
Total No of Test case: 21

Passed Test Cases % = No of Test case passed/ Total No of Test cases* 100

Passed Test Cases % = 100 %

Fig. 10. Screenshot of generated test cases

VI. EXPERIMENTAL EVALUATION

The goal of this phase was to dynamically analyze effectiveness of MUTCASGenerator by comparing it with a manual process known as manual test case generation technique (MATCASGenerator).The rationale was to find out which technique is actually effective in terms of; number of test cases generated, time taken to generate test cases, effort and cost of test generating cases. The experiment was done with the 4th year Bsc Software Engineering, Bsc IT & Bsc Computer Science students of various universities in Kenya.

A. Planning

The experiment subjects were selected on the basis that the subject has a good background in system analysis and design, coding and testing especially for object-oriented software.

Wohlin [20] suggest that there are two types of variable in a controlled experiment i.e. dependent and independent variables. In this experiment, there is only one independent variable which is test case generation approach used to generate test cases, with two treatments i.e. MUTCASGenerator and MATCASGenerator. The dependent variables were the test case generation effort (TE) and cost.

Test case generation effort (TE) is given by;

$$\mathrm{TE} = \frac{\text{Number of test cases}}{\text{Time taken to generate test cases}}$$

The cost was calculated by giving each subject “x” value as standard cost per second. For calculation, the researcher has given a standard value which is 0.5 units / second [21].

$$\text{Cost (C)} = \text{Time taken} * \text{Wage (x)}$$

The following shows measures collected in the experiment by each approach;

- Number of test cases generated
- Time taken to generate test cases

The results obtained after the experiment will be compared to the following hypothesis.

- Null hypothesis (H_{0TE}) = There is no significant difference in test case generation effort between MUTCASGenerator and MATCASGenerator
- Alternative hypothesis (H_{1TE}) = There is significant difference in test case generation effort between MUTCASGenerator and MATCASGenerator
- Null hypothesis (H_{0C}) = There is no significant difference in test case generation cost between MUTCASGenerator and MATCASGenerator
- Alternative hypothesis (H_{1C}) = There is significant difference in test case generation cost between MUTCASGenerator and MUTCASGenerator

B. Experimental design

The research followed empirical guidelines from Juristo and Moreno [22]. The study used between subjects experimental design and students were taken as the experimental subjects. There were 40 subjects, where each subject was randomly assigned to three different models.

C. Experiment execution

Preparations were made and training was conducted to the subjects via YouTube link (https://youtu.be/mz4N5gHIOCo) before actual execution. The subjects received models and test case templates via the email. Test case templates were used to document the test model. The experiment was conducted on 9th and 10th June, 2020 in an individual room at subjects’ home.

D. Results

The data collected was first subjected to descriptive statistics and then analyzed using inferential statistics which included linear regression and Z-test (two-tailed test).The results were presented by use of tables.

E. Results of test effort comparisons

The preliminary discussion of the results is based on the two building blocks of test effort construction, which are the number of test cases and time taken to produce those test cases. Then, the analysis is completed by comparing the results of test effort required by each test case generation approach.

F. Analysis of number test cases generated

Table-II shows that both techniques (MUTCASGenerator and MATCASGenerator) generated the same total number of test cases i.e. each produced sum of 795 test cases from 50 models.

Table -II: Showing results of total number of test cases generated by each approach

Technique	Statistic	Sum	Mean	Std. Deviation	Variance
		Statistic	*Statistic*	*Statistic*	*Statistic*
MUTCAS Generator	No Of Test cases	795	15.9	5.726	32.786
MATCAS Generator	No Of Test cases	795	15.9	5.726	32.786

G. Analysis of Time taken to generate test cases

The table-III shows that MUTCASGenerator is taking a total of 216.05 seconds to generate 795 test cases while MATCASGenerator is taking 23,760.00 seconds to generate 795 test cases, hence the MUTCASGenerator is faster in generating test cases than MATCASGenerator.

Table-III: Time taken to generate test cases by each approach

Technique	Statistic	Sum	Mean	Std. Deviation	Variance
		Statistic	*Statistic*	*Statistic*	*Statistic*
MUTCAS Generator	Time taken(s)	216.05	4.32	0.77	0.6
MATCAS Generator	Time taken(s)	23760	475.2	173.91	30245.89

H. Linear regression test on effort against number of test cases and time taken

Linear regression was applied to try to predict the values of effort based on number of test cases and time consumed.

The results from table-IV shows that In MUTCASGenerator, the variables have an Adjusted R Squire of 0.803 while in MATCASGenerator, the variables have an Adjusted R Squire of 0.815 .This means that in MATCASGenerator, the two predicators offer better explanation of about 82% of the variation in the dependent variable, effort as compared to variables in MUTCASGenerator which can explain about 80% of the variation of dependent variable effort

Table-IV: Showing correlation between effort and time and number of test cases

Model Summary				
Technique	*R*	*R Square*	*Adjusted R Square*	*Std. Error of the Estimate*
MUTCASGenerator	0.900[a]	0.811	0.803	0.2985
MATCASGenerator	0.907[a]	0.822	0.815	0.00407
a. Predictors: (Constant), Time taken(s), NoOfTest cases				

I. Analysis of variance between effort and time and number of test cases

Table-V is used to whether there is a statistically significant relationship between the effort and the combination of number of test cases and time taken i.e. The F-ratio in the ANOVA (table-V) test whether the regression model is a good fit of the data. The table shows that independent variables (Number of test cases and time taken) statistically significant predict the dependent variable (Effort), F (2, 47) = 100.765, p (0.000<.05) i.e. the regression model is a good fit of the data in both approaches.

Table-V: Showing ANOVA between effort and time and number of test cases

Technique		Sum of Squares	Df	Mean Square	F	Sig.
	Regression	17.957	2	8.978	100.765	0.000[b]
MUTCASGenerator	Residual	4.188	47	0.089		
	Total	22.145	49			
	Regression	0.004	2	0.002	108.842	0.000[b]
MUTCASGenerator	Residual	0.001	47	0		
	Total	0.004	49			
a. Dependent Variable: Effort						
b. Predictors: (Constant), Time taken(s), NoOfTest cases						

In the coefficient table-VI, the t-values and corresponding p-values are in the "t" and "sig" columns shows that number of test cases p(0.000)<0.05 and time taken p (.000)<0.05 are significant. In other words, number of test cases and time taken variables adds a substantial contribution in explaining dependent variable, effort in each approach.

Table-VI: Showing correlation coefficient between effort and time and number of test case

Coefficients[a]						
Model	*Statistics*	*Unstandardized Coefficients*		*Standardized Coefficients*	*T*	*Sig.*
		B	*Std. Error*	*Beta*		
	(Constant)	3.23	0.3		10.759	0
	NoOfTest cases	0.158	0.015	1.347	10.589	0
MUTCASGenerator	Time taken(s)	-0.494	0.111	-0.568	-4.462	0
	(Constant)	0.04	0.002		21.506	0
MATCASGenerator	NoOfTest cases	0.002	0	0.96	11.922	0
	Time taken(s)	-6.27E-05	0	-1.155	-14.338	0

a.Dependent Variable: Effor

J. Results of test effort

The table-VII shows that MUTCASGenerator is capable of generating 3.609 test cases per second on average which is the highest. Whereas MATCASGenerator is capable of producing 0.035 test cases on average. The results might indicate that MUTCASGenerator is more effective in terms of effort. However, the statistical tests performed in the next section statistically show the significant of the differences.

Table-VII: Showing Results of test effort

Technique	Statistic	Sum	Mean	Variance (n)	Std(n)
MUTCASGenerator	Effort	180.451	3.609	0.443	0.666
MUTCASGenerator	Effort	1.763	0.035	0	0.009

K. Z-test/ Two-tailed test on test case generation effort

In order to statistically demonstrate the test effort differences between the test case generation approaches, z-test for two independent samples was applied to the results to test the formulated hypothesis (H_{0TE}). The level of significance for the hypothesis test was also set to $\alpha = 0.05$. As can be observed from Table-VIII, the results suggest rejecting H_{0TE} in favor of H_{1TE} at the 0.05 significance level (since p-value < α, that is 0.0001 < 0.05). The analysis led to a conclusion that there is a statistically significant difference in test case generation effort between MUTCASGenerator and MATCASGenerator. This means that MUTCASGenerator consumes less effort as compared to MATCASGenerator.

Table-VIII: Z-test for test case generation effort

z(Observed value)	37.586
|z| (Critical value)	1.96
p-value (Two-tailed)	< 0.0001
Alpha	0.05

L. Results of cost comparisons

The results from table-IX show that MUTCASGenerator has a cost of 2.1605 units/seconds on average while MATCASGenerator has a cost of 237.6 units/seconds. From the observation MUTCASGenerator is less costly than MATCASGenerator.

Table-IX: Results of test cost comparison

Technique	**Statistic**	**Sum**	**Mean**	**Variance (n)**	**Standard deviation (n)**
MUTCASGenerator	Cost	108.027	2.161	0.146	0.382
MATCASGenerator	Cost	11880.000	237.600	7410.240	86.083

M. Hypothesis test on cost

In order to statistically demonstrate the cost differences between the MUTCASGenerator and MATCASGenerator, z-test for two independent samples was applied to the results to test the formulated hypothesis (H_{0C}). The level of significance for the hypothesis test was also set to $\alpha = 0.05$. As can be observed from table-X, the results suggest rejecting $_{0C}$ in favor of H_{1C} at the 0.05 significance level (since p-value < α, that is 0.0001 < 0.05). The analysis led to a conclusion that there is a statistically significant difference in cost between MUTCASGenerator and MATCASGenerator.
This means that MUTCASGenerator is less costly as compared to MATCASGenerator

Table-X: Z-test for test case generation cost

Difference	-235.439
z (Observed value)	-19.145
|z| (Critical value)	1.96
p-value (Two-tailed)	< 0.0001
Alpha	0.05

N. Linear regression test on cost against number of test cases

The result from the table-XI shows the relationship between variables in MUTCASGenerator has values of R Squire of 0.751 while in MATCASGenerator has value of 0.418. This means that in MUTCASGenerator, the number of test cases can explain about 75% of the variation in the dependent variable, cost. In MATCASGenerator, the predictor offer less explanation of about 42% of the variation in the dependent variable, cost.

Table-XI: Showing correlation between cost and number of test cases

Model Summary				
Technique	***R***	***R Square***	***Adjusted R Square***	***Std. Error of the Estimate***
MUTCASGenerator	.867[a]	0.751	0.746	0.1945814
MUTCASGenerator	.646[a]	0.418	0.406	4.415

Predictors: (Constant), NoOfTest cases

The F-ratio in the ANOVA (table-XII) test whether the regression model is a good fit of the data. The table shows that Number of test cases statistically significant predict the cost, F (2, 48) = 144.980, p (.000<.05) i.e. the regression model is a good fit of the data in both approaches.

Table-XII: Showing ANOVA between cost and number of test cases

ANOVA[a]						
Technique		***Sum of Squares***	***Df***	***Mean Square***	***F***	***Sig.***
MUTCASGenerator	Regression	5.489	1	5.489	144.980	0.000[b]
	Residual	1.817	48	0.038		
	Total	7.307	49			
MUTCASGenerator	Regression	671.044	1	671.044	34.433	0.000[b]
	Residual	935.456	48	19.489		
	Total	1606.500	49			

a. Dependent Variable: NoOfTest cases

b. Predictors: (Constant), Cost

In the coefficient table-XIII, the t-values and corresponding p-values are in the "t" and "sig" columns shows that number of test cases p(.000)<0.05, is significant. In other words, number of test cases variable adds a substantial contribution in explaining dependent variable, cost in each approach.

Table-XIII: Showing correlation coefficient between cost and number of test cases

Coefficients[a]						
Technique	*Statistic*	*Unstandardized Coefficients*		*Standardized Coefficients*	*t*	*Sig.*
		B	*Std. Error*	*Beta*		
MUTCAS Generator	(Constant)	1.231	0.082		15.023	0.000
	NoOfTest cases	0.058	.005	0.867	12.041	0.000
MUTCAS Generator	(Constant)	81.539	28.235		2.888	0.006
	NoOfTest cases	9.815	1.673	0.646	5.868	0.000

a. Dependent Variable: Cost

VII. CONCLUSION

The paper firstly involved determination of MUTCASGenerator requirements by analyzing existing test case generation techniques. Secondly, the flowchart and design of MUTCASGenerator are presented. On the same, an algorithm to help in generation of test case is described. Thirdly, the algorithm is implemented by developing a parser (MUTCASExtractor) that analyzes XML file and extract the required information (test cases). Finally, we conducted an experiment by comparing our technique with a manual process. The results show that the proposed technique can produce same test cases as manually writing test cases of the same system model but this technique saves a lot of time, effort and cost as well. The technique show the result that, the approach is useful to generate test case after completion of design phase and these test cases can help to discover software faults early in the software development cycle.

VIII. FUTURE WORK

The combination of class and activity diagram was considered in designing the technique and models of various systems. In future, there is need to add more diagrams like sequence diagram that capture time dependent sequence of interactions between objects, state machine diagram that capture dynamic behavior of class instances. The parser is unable to read other extension like (XMI). In future, the researcher has intention to upgrade the parser so as to read other file formats. Also the technique works with integer variable hence there is need to further generalize the approach by considering float, string data types..

REFERENCES

1. R.Abdul, N. A. J. Dayang, N. A .Ahmad, & , R..Abdullah , "Selecting UML models for generation of test cases*:* An experiments of technique to generate test cases," 2017.
2. S. M. Mohi-Aldeen., S. Deris., & R. Mohamad, "Systematic mapping study in automatic test case generation," in *New Trends* in Software Methodologies, Tools and Techniques. Fujita et al., eds. IOS Press, 703-720, 2014.
3. Meiliana, I. Septian, R.S. Alianto, Daniel & F.L. Gaol, *Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm*, 2nd International Conference on Computer Science and Computational Intelligence, ICCSCI 2017, 13, Bali, Indonesia..
4. Kaur & G. Gupta, *Automated Model-Based Test Path Generation from UML Diagrams via Graph Coverage Techniques*, Research article, ISSN 2320-088X, IJCSMC, *Vol. 2*, Issue. 7, 2013, pg. 302 – 311.
5. K. Swain, S. K. Pani, and D. P. Mohapatra, "Model based object-oriented software testing.," Journal of Theoretical & Applied Information Technology, vol. 14, 2010.
6. G. Booch, J. Rumbaugh, & I. Jacobson, *The Unified Modeling Language User* W.-K. Chen, Linear Networks and Systems (Book style). Belmont, CA: Wadsworth, 1993, pp. 123–135.
7. M. Prasanna,, K.R. Chandran, D.B. Suberi, *Automatic test case generation for UML class diagram using object diagram*. Academia Education., 2011, pg 1-7.
8. A.V.K. Shanthi, *Automated Test Cases Generation for Object Oriented Software* Indian Journal of Computer Science and Engineering (IJCSE). ISSN: 0976-5166 Vol. 2 No. 4, 2011.
9. M. Prasanna, K.R. Chandran, K.R., & D.B. Suberi, *Automatic Test Case Generation for UML Class Diagram using Data Flow Approach* .PSG College of Technology, Coimbatore, India, 2010.
10. A. Monim, & R.N.H. Nor, *An Automated Test Case Generating tool using UML Activity Diagram*. International Journal of Engineering & Technology, 7(4.31) 56 63, 2018.
11. R, Singh, & Preeti, *Automating the test case generation for object oriented system.* International Journal of Engineering and Computer Science, ISSN: 2319-7242, 2015.
12. A.Fernando, T. Diniz, B.S. Glaucia, "EasyTest: An approach for automatic test cases generation from UML Activity Diagrams," in Information Technology: New Generations (ITNG 2017), *14th International Conference on ITNG. Guide, 2017.*
13. A. Verma, *Automated Test Case generation using UML diagrams based on behaviour.* International Journal of Innovations in Engineering and Technology (IJIET), Vol.4, ISSN: 2319-1058, 2014
14. S. Jagtap, V. Gawade, R. Pawar, S. Shendge, & P. Avhad, *Generate Test Cases From UML Use Case and State Chart Diagrams* .International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395 -0056, 2016.
15. N. Khurana, & R.S, *Test Case Generation and Optimization Using UML Models and Genetic Algorithm*.3rd International Conference on Recent Trends in Computing 2015 (ICRTC-2015).
16. R. Mall, *Fundamentals of software engineering*, 2nd Ed. New Delhi: Prentice-Hall of India Ltd, 2009.
17. M. Utting, A. Pretschner, & B. Legeard, "A taxonomy of model-based testing approaches*," Software Testing***,** *Verification & Reliability*, vol. 22, no. 5, 2012, pp. 297-312.
18. K.J. Ajay, K.S. Santosh, P.M. Durga, *A novel approach for test case generation from uml activity diagram.* Issues and Challenges in Intelligent Computing Techniques (ICICT), International conference on IEEE, 2014.
19. M. Babu, & R. Kumar, & R. Bhatia, Rajesh, *Interaction Diagram Based Test Case, 2012.*
20. Wohlin, *Experimentation in software engineering: an introduction*. Springer, 2000.
21. R.R. Polamreddy, & S.A. Irtaza, Software Testing: *A Comparative Study Model Based Testing VS Test Case Based Testing*. School of Computing, Blekinge, Institute of Technology, SE-371 79 Karlskrona, Sweden, 2012. unpublished
22. N. Juristo, A.M. Moreno, *Basics of software engineering experimentation.* Springer Netherlands, 2001.

AUTHORS PROFILE

James Maina Mburu is an ICT Trainer at the Department of Information Communication Technology at Baraka Vocational Training Centre, Kenya. He is also a part time IT Lecturer at Murang'a University of Technology. He earned his Bachelor of Science in Information Technology from the Murang'a University of Technology, Kenya in 2016.. He is currently pursuing his MSc. in Information Technology at Murang'a University of Technology, Kenya. His research interests include software Engineering, software Testing and database management system

Geoffrey Muchiri Muketha is an Associate Professor of Computer Science at Murang' a University of Technology, Kenya. He received his BSc. in Information Science from Moi University, his MSc. in Computer Science from Periyar University, and his Ph.D. in Software Engineering from Universiti Putra Malaysia. His research interests include software and business process metrics, software quality, verification and validation, empirical methods in software engineering, and component-based software engineering. He is a Professional Member of the ACM and a member of the International Association of Engineers (IAENG).

Dr. Aaron Mogeni Oirere is a Lecturer and CoD of Computer Science Department at Murang'a University. He obtained his BSc. Degree in Computer Science from Periyar University in 2007, and his MSc. Degree in Computer Science from Bharathiar University in 2010. He holds a Ph.D. in Computer Science from Dr. Babasaheb Ambedkar Marathwada University. His interests include: Database Management Systems, Hardware & Networking, Human Computer Interface, Information Systems, Data Analytics and Automatic Speech Recognition. He has presented papers in scientific conferences and has many publications in refereed journals.