GENERATION OF COMBINATORIAL LOGIC ORIENTED TEST CASES FROM UML SEQUENCE DIAGRAM

SUBHASH TATALE1*, Dr. V. CHANDRA PRAKASH2
1 Research Scholar, 2 Professor
Department of CSE, Koneru Lakshmaiah Education Foundation, Vaddeswaram, AP, India
*Corresponding author E-mail: subhashatale@gmail.com

ABSTRACT

In the current practice, the test cases are generated from UML artefacts depending on the experience of testers in testing. Many researchers used different techniques to generate test cases from UML artefacts. There is a need to generate combinatorial logic-oriented test cases for those systems where combinatorial logic is necessary.

Combinatorial testing plays an essential role in generating a minimum number of the test cases to detect defects caused by interactions among system parameters. To generate combinatorial logic-oriented test cases, information about parameters, their values, and constraints is essential.

UML Sequence Diagram represents the dynamic behaviour of a software system. Extracting and identifying information about parameters, values and constraints from UML Sequence Diagram and detecting interactions among those extracted parameters is a challenging task. The authors proposed multi-stage algorithm to extract and identify information about parameters, values and constraints from Sequence Diagram. The authors designed and developed a technique that automatically generates combinatorial logic-oriented test cases from UML Sequence Diagram.

In this paper, a case study of Concession Management SubSystem of Indian Railways is presented. The authors generated automated test cases using the proposed Combinatorial Logic Oriented Test Case Generator for the case study and compared those test cases with manually generated test cases. It is found that generated automated and manual test cases are matching the same with each other.

Keywords: Covering Array, Combinatorial Test Case Generation, Behavioral UML Diagrams, Sequence Diagram, Railway Reservation System, Concession Management System

1. INTRODUCTION

Generating test cases at design phase has several advantages over coding phase of Software Development Life Cycle (SDLC). The test cases generated in the design phase remain valid even when we do a little bit change in the code [1]. The design models which are designed in design phase can be used as an input for generating the test cases. This will help to identify problems early in the stages of SDLC, which significantly reduces the time and cost of testing. However, generating test cases from Unified Modelling Language (UML) model is a difficult task. UML has now become the de facto standard for object-oriented modelling and design. UML models are an essential source of information for generating test cases [2]. An automatic generation of test cases from UML model is a practically essential and receiving more awareness from researchers. Many researchers have presented various methodologies and techniques to generate test cases automatically using UML Sequence Diagrams [3]. There are many systems like Concession Management SubSystem (CMSS) of Indian Railways, College admission system, Tuition fee concession subsystem, etc., in which combinatorial logic is extensively used.

Combinatorial Testing (CT) is gaining high importance to test such type of systems.

The Combinatorial Test Design Model (CTDM) is popularly used to generate combinatorial test cases automatically [4]. There is a need to generate combinatorial test cases for those systems where combinatorial logic is essential. Different UML diagrams are used to model these systems. The same UML diagrams are used to generate combinatorial test cases.

The primary challenge of generating combinatorial logic-oriented test cases from the UML Sequence Diagram is to identify the input parameters, associated values, and constraints.
Many times application fails because of an interaction among the values of the different parameters of that application. It is an error-prone and challenging task to identify the input parameters, values and constraints manually. Hence, there is a need to model these parameters so that it reduces error and increase quality of the software system. A multi-stage algorithm is proposed to extract parameters, values and constraints from the sequence diagrams.

In this paper, the authors presented a case study of CMSS of Indian Railways to generate combinatorial logic-oriented test cases. The requirement specifications of CMSS are firstly modelled in the UML Sequence Diagram, and then all parameters, their values and constraints are extracted from this diagram using proposed multi-stage algorithm. After extraction of this information, combinatorial logic rules are applied and combinatorial test cases are generated. In the next section, concepts related to Sequence Diagram and Combinatorial Test Case Generation is discussed.

1.1 Combinatorial Test Case Generation

Grindal et al. [5] and Nie et al. [6] did a detailed survey on CT. The authors covered all aspects of CT including test modelling and CT applications. If the number of parameters and the values of these parameters are large, then it is practically impossible to generate all the combinatorial test cases. Covering Array (CA) which is a mathematical approach is used to reduce the test cases is [7] [8]. CA has four parameters, namely the number of input parameters (p), number of values (v), interaction strength (t) and number of test cases (N) [9].

For example, input has three parameters (namely X, Y, Z) having two values to two parameters each and three values to one parameter. Total $3 \times 2 \times 2 = 12$ test cases are required to test combinations of all these parameters and values.

<table>
<thead>
<tr>
<th>Test case number</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X1</td>
<td>Y1</td>
<td>Z1</td>
</tr>
<tr>
<td>2</td>
<td>X1</td>
<td>Y2</td>
<td>Z2</td>
</tr>
<tr>
<td>3</td>
<td>X2</td>
<td>Y1</td>
<td>Z1</td>
</tr>
<tr>
<td>4</td>
<td>X2</td>
<td>Y2</td>
<td>Z2</td>
</tr>
<tr>
<td>5</td>
<td>X3</td>
<td>Y1</td>
<td>Z1</td>
</tr>
<tr>
<td>6</td>
<td>X3</td>
<td>Y2</td>
<td>Z2</td>
</tr>
</tbody>
</table>

The optimal test suite has only six tests by using pairwise testing as shown in Table 2.

Combinatorial testing does not cover all parameter combinations, but it does show significant results in terms of detecting maximal defects in a small test suite. Table 2 shows a reduction in the size of the test suite from 12 to 6. Although it may not be exciting, we may test the effectiveness by using a more complex test input. Consider a test input with 20 parameters and ten values per parameter. $10^{20}$ test cases were developed by exhaustive testing. We may minimise the test suite size to 213 test cases by using combinatorial testing.

1.2 UML Sequence Diagram

UML Sequence Diagrams are used to capture dynamic behaviour of a system from a different perspective. Sequence Diagram visualizes time-dependent interaction among the objects. It depicts the message sequence, as well as their names, responses, and probable counter arguments. In a Sequence Diagram, the vertical line shows time, whereas the horizontal line shows interaction among different objects [10].

In UML 2.0, different interaction fragments are used to describe many traces compactly and concisely. A fragment is an interaction operator.
which is used to showcase a conditional flow in the Sequence Diagram. It operates on a group of operands. Each operand represents a sequence of messages that occur under a guard condition. Some of the interaction operators are explained in Table 3.

Table 3: Interaction operators with its purpose

<table>
<thead>
<tr>
<th>S. No.</th>
<th>Operator Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>alt (alternatives)</td>
<td>This operator is used using multiple operands to capture alternative flows</td>
</tr>
<tr>
<td>2</td>
<td>opt (optional)</td>
<td>This operator has only one operand that is interpreted optionally</td>
</tr>
<tr>
<td>3</td>
<td>Break</td>
<td>This operator is used to capture an exit pathway of the systems</td>
</tr>
<tr>
<td>4</td>
<td>Loop</td>
<td>This operator is used to model the repetitive interactions in a diagram</td>
</tr>
<tr>
<td>5</td>
<td>neg (negative)</td>
<td>This operator describes a combined fragment of traces that are defined to be negative (invalid)</td>
</tr>
</tbody>
</table>

The rest of the paper is structured as follows: In Section 2, the related work is briefly discussed. Section 3 explains the proposed work. Section 4 describes the results and findings of the proposed work, while Section 5 contains concluding remarks.

2. RELATED WORK

The related work of test case generation and combinatorial test design model from the Sequence Diagram is discussed in this section. Also, a case study of Concession Management SubSystem of Indian Railways is discussed.

2.1 Test Case Generation from Sequence Diagram

Many researchers published research articles on the test case generation from Sequence Diagram using different approaches. Subhash Tatale et al. [13] published a survey paper on Test Case Generation using UML Diagrams and Feasibility Study to Generate Combinatorial Logic Oriented Test Case. The authors covered various test case generation techniques from Sequence Diagram.

Using the Formal specification approach, Panthi Vikas et al. [14], Zhang Chen et al. [15], Rihnman Waskur et al. [16], and Nour El Houdea Dehimi et al. [17] developed test cases. For this, the authors used Model checking, Formal specification, Object Constraint Language (OCL), and an Agent-based approach. Message and Path coverage criteria are met using these methods and approaches.

Using the Graphical representation approach, Samuel Philip et al. [18], Swain et al. [19] [20], and Dhinesh Kumar M et al. [21] developed test cases. The methods of dynamic slicing and iterative deepening Depth First Search are employed. This technique meets the path and full predicate coverage criteria.

Jena Ajay Kumar et al. [22] employed a heuristic approach to generate test cases from a Sequence Diagram automatically. For this, a genetic algorithm is applied, and it meets the message coverage criteria.

For producing test cases, Beyer et al. [23] and Costa Leandro et al. [24] employed the Direct UML specification processing approach. The Markov Chain Usage Model and Parsing method are used to achieve a message coverage condition.

The Concurrent model approach was used by Khandal Monalisha et al. [25] [26] [27] and Mani P. et al. [28] to build test cases. Message, Sequence, and Path coverage criteria are met using Depth First Search, Breadth First Search, and Stack array approaches.

2.2 Combinatorial Test Design Model from Sequence Diagram

The related work on the combinatorial test design model is discussed in this section. Sasi Bhanu et al. [29] [30], Mudarakola et al. [31] and M. Laxmi Prasad et al. [32] [33] published research articles for testing distributed embedded systems using combinatorial testing methods. V. Chandra Prakash et al. [34] [35] performed a review on automated generation of combinatorial test cases using particle swarm optimization and generated test cases for pairwise + testing. For safety-critical embedded systems, Vudatha et al. [36] [37] [38] used a genetic algorithm to derive combinatorial test cases from the output domain. This method ensures that the largest numbers of output combinations are thoroughly generated and tested. In a constraints management context, Ramgouda Patil et al. [39] [40] [41] proposed a Neural Network strategy to improve
combinatorial coverage, as well as multi-objective crow search and fruit-fly optimization strategies to optimize combinatorial test cases.

There is a need to generate combinatorial logic-oriented test cases from UML artefacts. Satish Preeti et al. [42] presented a rule-based semi-automated approach for obtaining the information of combinatorial test design model. Sajad Esfandyari et al. [43] employed model checking approaches to extract parameters and values from state space. Subhash Tatale et al. [44] proposed an approach of enhancement in acceptance test-driven development using combinatorial logic.

2.3 Concession Management SubSystem – A Case Study

In this section, the authors presented a case study of CMSS of Indian Railways. Indian Railways offers concessions on ticket fares to different categories of concessions like Disabled Passenger, Patient, Senior Citizen, Child, War Widow, etc. These concessions are offered based on various types of journey classes like Sleeper Class, First Class, etc. There are several types of concessions in each concession category.

Subhash Tatale et al. [44] published a research article on applying Combinatorial Logic to improve the Acceptance Test Driven Development Model. The authors presented Software Requirement Specification (SRS) of CMSS of Indian Railways in the view of combinatorial logic. While generating combinatorial test cases for the mentioned journey classes (7 journey classes), concession categories (11 concession categories) and their types (174 concession types) in the research article [44], the size of test cases may be enormous because of too many combinations of parameters and values in the input. If we apply All Combinations testing technique to those concession categories and types, then the total number of generated test cases will be $2 \times 3 \times 15 \times 127 \times 5 \times 6 \times 3 \times 3 \times 2 = 9258300$. It is challenging and unrealistic to generate and to test such a large number of test cases. It is called a Combinatorial Explosion problem of test cases.

Therefore, the authors of this paper considered limited journey classes and concession categories to avoid the combinatorial explosion problem of test cases. The authors condensed some of the concession categories and types in the revised SRS of the CMSS. Only limited Journey class, concession categories and types are considered in the revised SRS of the CMSS. The list of different concession categories, types as per revised SRS of CMSS is shown in Table 4. These requirements are considered to generate combinatorial logic oriented test cases from the Sequence Diagram.

<table>
<thead>
<tr>
<th>Category of Passenger</th>
<th>Reservation Class</th>
<th>Percentage of Concession</th>
</tr>
</thead>
<tbody>
<tr>
<td>Disabled Passenger</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Handicapped</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Mentally retarded</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Patient</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cancer</td>
<td>100</td>
<td>75</td>
</tr>
<tr>
<td>Heart</td>
<td>75</td>
<td>75</td>
</tr>
<tr>
<td>Senior Citizen (&gt;60 years)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Child (&lt;12 years)</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>Widow</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 1 depicts Sequence Diagram of CMSS considering revised SRS as per Table 4. Figure 1 is shown at the end of the paper.

3. THE PROPOSED COMBINATORIAL LOGIC-ORIENTED TEST CASE GENERATOR (CLOTCG)

It is a very tedious and challenging task to identify manually the exact number of parameters, their values, and constraints from UML diagrams. It is very difficult to extract the information in manual way when the values of the parameters are dynamic. In this section, the authors presented a technique that helps to extract the required preliminary information automatically from the UML Sequence Diagram in the form of parameters, values and constraints. The combinatorial logic is applied to that extracted information to generate combinatorial logic-oriented test cases.

The authors show how to generate combinatorial logic-oriented test cases using the suggested Combinatorial Logic-Oriented Test Case Generator (CLOTCG) technique from UML Sequence Diagram. A model is designed based on Software Under Test (SUT) and then the elements of CTDM are generated. The multi-stage algorithm is used to extract the information
from those CTDM. Combinatorial logic is used to construct the combinatorial logic-oriented test cases.

Figure 2 depicts the Data Flow Diagram of the proposed CLOT CG. The Sequence Diagram is drawn using the StarUML tool as per the requirement specifications. A test manager will give Sequence Diagram as an input to the proposed system. The information like parameters, values and constraints are extracted from Sequence Diagram using multi-stage algorithm. Sequence diagram is converted into XML Metadata Interchange (XMI). The XMI document is extracted using JavaScript Object Notation (JSON) object. JavaScript is used to parse JSON object for validation of the extracted information of the Sequence Diagram. Once it is validated, then the information required for the CLOT CG is extracted. The combinations of parameters and values are generated from extracted parameters and values. The constraints are processed on a combinatorial generated list. After that, combinatorial logic-oriented rules are applied. Finally, combinatorial logic-oriented test cases are generated. The proposed technique for extracting useful information from a Sequence Diagram is explained in detail in the following section.

3.1 Extraction of information from Sequence Diagram

The information like parameters and values is essential to generate combinatorial logic-oriented test cases. In this section, the authors presented a technique that extracts the information which is essential to generate combinatorial logic-oriented test cases. The UML sequence diagram is drawn using the StarUML tool.

3.1.1 Generate XML Metadata Interchange (XMI) from UML Sequence Diagram

XMI is a perceptive way of converting UML models into XML documents. It is used to exchange metadata information using Extensible Markup Language (XML). The main rationale for extracting XMI is to allow for a smooth interchange of metadata in distributed heterogeneous contexts between modelling tools and metadata repositories. In XMI format, the message, start and end of a fragment are first found out. The XMI Code is shown in Figure 3.

```
<uml:Model xmi:id="AAAAAAAFjohXshGwltM=" xmi:type="uml:Model" name="RootModel">
  <packagedElement xmi:id="AAAAAAAFF+qBWK6M3Z8Y=" name="Model" visibility="public" xmi:type="uml:Model">
    <package xmi:id="AAAAAAAFzTuURy4VdAI=" name="Collaboration1" visibility="public" isAbstract="false" isFinalSpecialization="false" isLeaf="false" xmi:type="uml:Package">
      <ownedMember xmi:id="AAAAAAAFzTuURy4WSOI=" name="Interaction1" visibility="public" isReentrant="true" xmi:type="uml:Interaction">
        <lifeline xmi:id="AAAAAAAFzTuXvBY4k9h4="/ name="Passenger" visibility="public" xmi:type="uml:Lifeline" represents="AAAAAAAFzTuXvBY4k9h4="/>
        <lifeline xmi:id="AAAAAAAFzTuYYgo5EvS=="/ name="Railway\%20Authority" visibility="public" xmi:type="uml:Lifeline" represents="AAAAAAAFzTuYYgo5EvS="/>
        <message xmi:id="AAAAAAAF4WWSr/e8zO4="/ name="Select\%20Journey\%20Class" visibility="public" xmi:type="uml:Message">
          <messageSort="synchronCall" messageKind="complete" receiveEvent="AAAAAAAF5johXIRGy+8="/ sendEvent="AAAAAAAF5johXIRGy+8="/>
        </message>
      </ownedMember>
    </package>
  </packagedElement>
</uml:Model>
```

Figure 3. Code snippet of XMI
3.1.2 Generating JSON code from XMI

JSON is a lightweight data-transfer format for sending data between client and server that is simple to understand and generate. JSON, like XML, is a text-based format that is simple for humans and machines to write and interpret.

A JavaScript library called xml2json converts the given XML code to JSON code to generate a mapping of parameters and values.

This library acts as an XML to JSON converter. In JavaScript, JSON.parse() method is used to parse JSON object. This method is primarily used to return a JavaScript object from the string that is parsed. The string that will parsed with JSON.parse() is in a JSON format. The mapping of the parameter and values are stored in .csv file that is extracted through the JSON code. Figure 4 depicts the JSON code snippet.

```json
4
{"uml:Model":{"$":{"xmi:id":"AAAAAAF5johXshGwLtM=","xmi:type":"uml:Model","name":"RootModel"},
"packagedElement":{"$":{"xmi:id":"AAAAAAAFF+nBK6M32Z8Y=","name":"Model","visibility":"public","xmi:type":"uml:Model"},
"ownedMember":{"$":{"xmi:id":"AAAAAAAFAfTuRy4VdA1=","name":"Collaboration1","visibility":"public","isAbstract":false,"isFinalSpecialization":false,"isLeaf":false,"xmi:type":"uml:Collaboration"},
"ownedMember":{"$":{"xmi:id":"AAAAAAAFAfTuRy4W5SOI=","name":"Interaction1","visibility":"public","isReentrant":true,"xmi:type":
"uml:Interaction"},
"lifeline":{"$":{"xmi:id":"AAAAAAAFAfTuXbY4DzQ=","name":"Passenger","visibility":"public","xmi:type":
"uml:Lifeline","represents":AAAAAAAFAfTuXbY44k94="},
"lifeline":{"$":{"xmi:id":"AAAAAAAFAfTuYYgo5F4Bo=","name":"Railway Authority","visibility":"public","xmi:type":
"uml:Lifeline","represents":AAAAAAAFAfTuYYgY5EVoQ="}},
"message":{"$":{"xmi:id":"AAAAAAAFAfW4W5W5STxRy4VdA1=","name":"Select Journey Class","visibility":"public","xmi:type":
"uml:Message","messageSort":"synchCall","messageKind":"complete","receiveEvent":AAAAAAAFAfjohX8Ry4VdA1+","sendEvent":AAAAAAAFAfjohX8Ry4VdA1+}}
}]
```

Figure 4. Code snippet of JSON

3.1.3 Extraction of parameters, values and constraints

1. Sequence Diagram with synchronous messages

The SUT input variables are considered as test input parameters. There are two object lifelines, namely Passenger called Object 1 and Railway Authority called Object 2. These two objects communicate using synchronous messaging. A solid arrowhead pointing from left to right denotes synchronous messaging. Every message that is sent receives a response message. This reply message is shown by a dotted arrowhead pointing from right to left. Object 1 will not be able to continue processing until it receives a response. As a result, this classification aids in the immediate identification of parameters and their values. The parameters and values shown in Figure 5 (a) are the primary information. This information is treated as an input of the system.

![Sequence Diagram with Synchronous Messages](image)

Figure 5. (a) Sequence Diagram with Synchronous Messages

The loop fragment is used to consider combinations of values for those parameters. The information shown in Figure 5 (b) and (c) are optional information. NS (Not Selected) indicates that the passenger has not selected any concession. ‘and’ keyword indicates that the...
passenger is considering multiple combinations of parameter values.

**Figure 5. (b) Sequence Diagram with loop operator**

![Sequence Diagram with loop operator](image)

**Figure 5. (c) Sequence Diagram with opt operator**

![Sequence Diagram with opt operator](image)

**a) Steps for parameter extraction**
The outgoing messages from the lifelines of Object 1 to the Object 2 are called as parameters. The below steps are used to extract parameters from Sequence Diagram.

1: Read the JSON code
2: Get the object name of the Object 1 from the code
3: Find a message passed from the lifeline of Object 1 to the lifeline of Object 2.
4: Display the name of the message as a parameter.

**b) Steps for value extraction**
The incoming messages towards the lifelines of Object 1 from the Object 2 is values. The below steps are used to extract values from Sequence Diagram.

1: Read the JSON code
2: Get the object name of the Object 1 from the code.
3: Find a message passed to the lifeline of Object 1 from the lifeline of object 2.
4: Display the name of the message as a value.

**c) Steps for identifying constraints**
The neg operator describes a combined fragment of traces that are defined to be negative or invalid.

The outgoing messages from the Object 1 to the Object 2 are combinations of parameters and values, and its reply message from the Object 2 back to the Object 1 is the infeasible combinations.

**Figure 6. Sequence Diagram with neg operator**

![Sequence Diagram with neg operator](image)

The below algorithm is used to identify the constraints from Sequence Diagram.

```java
open the JSON code in read mode
if (combined fragment= neg) {
    get the object name of the Object 1 from the code
    find a message passed from the Object 1 lifeline to the Object 2 lifeline
    if message is found
        search next sibling message node of this message which is passed back to the Object 1 lifeline
}
```
if message is Infeasible Input
{
    display the message name
    map the Infeasible combinations of
    parameters and their values respectively
}
}

Table 5 shows the extraction of constraints or
invalid combinations of parameters and values
after applying above mentioned steps on
components of Sequence Diagram shown in
figure 6.

Table 5: Extracted constraints from Figure 6

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Concession Categories</th>
<th>Concession types</th>
<th>Infeasible concession categories</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Gender</td>
<td>Male</td>
<td>Widow</td>
</tr>
<tr>
<td>2</td>
<td>Passenger type</td>
<td>Child</td>
<td>Widow</td>
</tr>
</tbody>
</table>

2. Identifying combinatorial logic oriented rules

In a sequence diagram, an alt operator is used
to express a "if-then" condition. The outgoing
messages from Object 1 to Object 2 are the
combinatorial logic oriented concession rules,
and the reply message from Object 2 to Object 1
is the percentage of concession offered to various
parameter combinations.

Figure 7. Sequence Diagram with alt operator

<table>
<thead>
<tr>
<th>alt Concession Rules</th>
<th>17: Maximum Allowed Concession</th>
</tr>
</thead>
<tbody>
<tr>
<td>18: 100% Concession</td>
<td></td>
</tr>
<tr>
<td>19: If No. of concession types selected = 1</td>
<td></td>
</tr>
<tr>
<td>20: % of total concession is applicable as per SRS</td>
<td></td>
</tr>
<tr>
<td>21: If No. of concession types selected = 2</td>
<td></td>
</tr>
<tr>
<td>22: Sum of % of highest concession type and 5% of remaining higher concession type</td>
<td></td>
</tr>
<tr>
<td>23: If No. of concession types selected = 3</td>
<td></td>
</tr>
<tr>
<td>24: Sum of % of highest concession type and 7% of remaining higher concession type</td>
<td></td>
</tr>
<tr>
<td>25: If No. of concession types selected more than 3</td>
<td></td>
</tr>
<tr>
<td>26: Sum of % of highest concession type and 10% of remaining higher concession type</td>
<td></td>
</tr>
</tbody>
</table>

The below algorithm is used to identify the
combinatorial logic rules from the JSON code.

1. open the JSON code in read mode
2. if (combined fragment== alt)
3. {
4.     find a message passed from the
Object 1 lifeline to the Object 2
lifeline
5.     if message is found
6.     {
7.         search next sibling message
node of this message which is passed
back to the Object 1 lifeline
8.     }
9.     }
10. display the value of the
attributes of the messages
11. return percentage of
concession
12. }
13. }

Table 6 shows the combinatorial logic oriented
concession rules from Figure 7.
Table 6: Combinatorial logic oriented concession rules from Figure 7

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Concession rules as parameter</th>
<th>Total concession (in %) as value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>If no. of selected concession types = 1</td>
<td>% of total concession is applicable as per Table 2.</td>
</tr>
<tr>
<td>2</td>
<td>If no. of selected concession types = 2</td>
<td>% of highest concession type + 5% of remaining concession type as per Table 2.</td>
</tr>
<tr>
<td>3</td>
<td>If no. of selected concession types = 3</td>
<td>% of highest concession type + 7% of remaining higher concession type as per Table 2.</td>
</tr>
<tr>
<td>4</td>
<td>If no. of selected concession types &gt; 3</td>
<td>% of highest concession type + 10% of highest of the remaining concession type as per Table 2.</td>
</tr>
<tr>
<td>5</td>
<td>Maximum allowed concession</td>
<td>100%</td>
</tr>
</tbody>
</table>

3.2 Combination of parameters and values

In the previous section, the parameters and values are extracted from the Sequence Diagram using various steps. Combinations of parameters and values are necessary to generate combinatorial logic oriented test cases. In this section, the algorithm for combination of extracted parameters and values is discussed. For the combination, mapping of appropriate parameters and values are necessary. The below algorithm is used for making combination of extracted parameters and values.

Algorithm:
read JSON code
for all parameter name from JSON code
{
    match message name with parameter
        check sendEvent of that message
            match combined fragment with xmi:id
                check coverage of that fragment
                    if coverage is found then
                        select xmi:id of the respective coverage
                        match this xmi:id to the message of receiveEvent
                        extract the value of corresponding parameter
                }   }

Table 7 shows the extracted parameters and associated values after applying above mentioned steps on components of Sequence Diagram shown in figure 5 (a) (b) (c).

Table 7: Extracted parameters and values from Figure 5

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Journey class</th>
<th>Gender</th>
<th>Passenger type</th>
<th>Disabled Passenger</th>
<th>Patient</th>
<th>Widow</th>
</tr>
</thead>
</table>

3.3 Apply combinatorial logic

The parameters, their associated values, constraints (if any) and combinatorial logic-oriented rules play an important role for generation of combinatorial logic-oriented test cases. We have covered the multi-stage algorithm techniques to extract this information in the previous section. The All Combinations (AC) testing technique generates every possible combination of parameters and values.

In the CMSS, we extracted parameters and values. We extracted six parameters like Journey Class, Gender, Passenger type, Disabled passenger, Patient, Widow and its associated values as per Table 4.

As per the Sequence Diagram shown in Figure 5 (b), the Disabled passenger and Patient category have multiple selection options. The multiple selection options are shown using AND conditional operator in figure 5 (b). It indicates that for Disabled Passenger and Patient category, multiple value combinations are selected.
Therefore, All Combinations test suite satisfies N-wise coverage.

All combination values = \( \sum \) vi value combinations, where N is the coverage number (1-way, 2-way,...N-way) and vi is the number of values of parameter.

Hence, we can calculate all combination values for Disabled passenger and Patient parameter as follows:

All combination value (Disabled Passenger) = \{Handicapped, Mentally retarded, Handicapped and Mentally retarded\}
All Combinations value (Patient) = \{Cancer, Heart, Cancer and Heart\}

Therefore, values of Disabled passenger and Patient concession category are increased due to All Combinations testing techniques. All the remaining values of concession categories viz. Journey class, Gender, Passenger type and Widow are mutually exclusive. Table 8 shows number of values for each parameter after applying All Combinations testing technique.

Table 8:- Number of values for parameters using All Combinations testing technique

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Journey class</th>
<th>Gender</th>
<th>Passenger type</th>
<th>Disabled Passenger</th>
<th>Patient</th>
<th>Widow</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. of Values</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Hence, total number of test case combinations are generated = 2x2x3x4x4x2=384  ---------(1)

There are some Infeasible Combinations are derived as shown in Table 5.

1. Male – Widow infeasible combinations
2. Child- Widow infeasible combinations

These combinations are considered as Not Applicable (NA). The NA value is assigned to respective parameter because of infeasible combinations between parameters and values. For such kind of test cases, NA and NS value conflicts to each other. These test cases are considered as duplicate test cases. The test cases containing NS value are removed as per below calculations.

1. Male – Widow infeasible combinations

Total number of duplicate test combinations because of Male- Widow infeasible combinations

\[ = 2 \text{ (Journey Class) } \times 2 \text{ (Gender) } \times 1 \text{ (Passenger type except Child) } \times 4 \text{ (Disabled passenger) } \times 4 \text{ (Patient) } \times 1 \text{ (Widow) } = 64 \]  -------(2)

2. Child- Widow infeasible combinations

Total number of duplicate test combinations because of Child- Widow infeasible combinations

\[ = 2 \text{ (Journey Class) } \times 2 \text{ (Gender) } \times 1 \text{ (Passenger type - Child) } \times 4 \text{ (Disabled passenger) } \times 4 \text{ (Patient) } \times 1 \text{ (Widow) } = 64 \]  -------(3)

Total number of duplicate test combinations generated because of Infeasible Combinations

\[ = 64+64 \text{ (From Eq.2 and 3) } = 128 \]  -------(4)

Total number of distinct test cases generated = 384-128 (from Eq. 1 and 4) = 256

Hence, total 256 test cases are generated from Sequence Diagram using CLOT CG technique automatically.

The percentage of concession is applied based on combinatorial logic oriented rules mentioned in Table 4 and 7. The test cases generated using the proposed CLOT CG is shown in Table 9.

Table 9:- Test cases generated using CLOT CG

<table>
<thead>
<tr>
<th>TC No.</th>
<th>Journey class</th>
<th>Gender</th>
<th>Passenger type</th>
<th>Disabled passenger</th>
<th>Patient</th>
<th>Widow</th>
<th>Expected concession (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>First</td>
<td>Male</td>
<td>Child</td>
<td>-</td>
<td>-</td>
<td>NA</td>
<td>50</td>
</tr>
<tr>
<td>2</td>
<td>First</td>
<td>Male</td>
<td>Child</td>
<td>-</td>
<td>Cancer</td>
<td>NA</td>
<td>77.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Senior Citizen</td>
<td>Handicapped and Mentally retarded</td>
<td>Cancer and Heart</td>
<td>War</td>
<td>100</td>
</tr>
</tbody>
</table>

5210
These test cases also cover constraints of parameter-value combinations as per Table 6. The ‘NA’ value in the test case indicates all infeasible options. All combinations testing technique is good for the smaller size of parameter and values. If the number of parameters and values is large, it is a very tedious task to generate and test those exhaustive test cases.

4. RESULTS AND DISCUSSIONS

The authors presented All Combinations testing technique to generate combinatorial logic-oriented test cases from UML Sequence Diagram. These test cases will aid in improving the testing efficacy. All Combinations testing technique gives a better result for UML Sequence Diagram. It is found that generated automated and manual test cases are matching the same with each other. The Input Size of Table 8 indicates the number of parameters and values extracted from the UML Sequence Diagram. CA (2\(^1\)2\(^3\)3\(^4\)4\(^2\)) input size indicates 6 concession categories (parameters) having some number of concession types (values). Out of 6 parameters, 3 parameters are having 2 values each, 1 parameter is having 3 values and 2 parameters are having 4 values each. CA (2\(^3\)3\(^4\)4\(^1\)) input size indicates 7 concession categories (parameters) having some number of concession types (values). Out of 7 parameters, 2 parameters are having 2 values each, 2 parameters are having 3 values, 2 parameters are having 4 values and 1 parameter is having 15 values.

The result of the proposed technique is shown in Table 10.

<table>
<thead>
<tr>
<th>Sr. No.</th>
<th>Input Size</th>
<th>No. of Test cases generated manually</th>
<th>No. of Test cases generated by CLOTCG</th>
<th>Accuracy (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CA(2(^1)2(^3)3(^4)4(^2))</td>
<td>256</td>
<td>256</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>CA(2(^3)3(^4)4(^1))</td>
<td>2208</td>
<td>2208</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 8 shows the result comparison of number of test cases generated by Manual method and proposed CLOTCG method. It shows that proposed method gives the 100% accuracy.

4.1 Findings

Generating a large number of test cases is a very laborious, time-consuming and costly task by using manual method. In addition, during manual test case generation, some erroneous test cases may be generated. The test cases are currently designed by the test designers from UML artefacts based on their experience with the manual testing approach.

The authors of this paper found a research gap for generating combinatorial test cases automatically from UML artefacts. An automatic test case generation is essential to avoid erroneous test cases generated using the manual testing approaches. Moreover, automatic generation of test cases can reduce testing cost by eliminating costly manual test case generation. Thus, it reduces the time and cost of testing significantly.

The automatic test case generation from UML diagrams will help to identify problems in the early stages of SDLC. The test cases generated from the UML Sequence Diagram using the proposed CLOTCG are 100% accurate. The accuracy percentage shows a comparison between manually generated test cases and automated generated test cases using the CLOTCG. The authors claimed that the proposed testing technique gives reliability and efficiency ultimately.
4.2. Extension to Existing Techniques

Many researchers proposed different approaches like Formal specification-based approach, Graphical representation approach, Heuristic approach, Direct UML specification processing approach, Hybrid behaviour model approach and Concurrent model approach to generate test cases from UML Sequence Diagram. These approaches are used to generate the test cases for functional testing of the software systems.

There is a need to provide combinatorial logic-oriented test cases for the systems that use combinatorial logic, such as reservation systems, college entrance systems, concession management systems, and so on. The authors of this paper suggested a novel method for generating combinatorial logic-oriented test cases from UML Sequence Diagrams.

The proposed technique generates combinatorial test cases based on parameters, values, and constraints extracted from Sequence Diagram, whereas the existing techniques generate test cases based on paths, messages, sequences, etc.

4.3 Limitations of the Proposed Technique

UML represents software requirements specification in a graphical or diagrammatic way. Sometimes, it is challenging to represent all these requirements using UML diagrams. Some diagram shows behavioural, creational, structural views of the requirements. Any single diagram cannot capture all the requirements of software systems. Hence, it is a limitation of the generation of test cases from one of the UML diagrams. It is infeasible and challenging to generate combinatorial test cases for complex Sequence Diagrams.

Only Sequence Diagram cannot capture all the requirements of the software system. Therefore, some requirements might get missed out and subsequently, those test cases will not get generated. As a result, combinatorial test cases must be built from other UML diagrams like Activity Diagram, Use Case Diagram, and State Machine Diagram in order to capture all requirements and generate test cases.

5. CONCLUSION

Using the manual testing technique for generating a large number of test cases is an extremely time-consuming, labor-intensive, and expensive task. Furthermore, certain incorrect test cases may be generated during manual test case generation. The test cases are currently being created by the test designers using UML artefacts and their manual testing experience. The authors found that there is need to generate combinatorial test cases automatically from UML artefacts to avoid erroneous test cases which are generated using the manual testing approaches.

In this paper, the authors proposed CLOTCG technique to generate combinatorial logic-oriented test cases from Sequence Diagram automatically. The authors presented a multi-stage algorithm to extract parameters, values and constraints from UML Sequence Diagram. The guard conditions of various combination fragments and the messages in synchronous message calls of the Sequence Diagram are used to identify these parameters, values, and constraints. The appropriate rules are applied for various semantic constructs, guard condition, synchronous message calls, and constraints of Sequence Diagram.

CMSS of Railway Reservation System of Indian Railways is presented as a case study to demonstrate the proposed technique. The authors generated automated test cases using the proposed Combinatorial Logic Oriented Test Case Generator for the case study and compared those test cases with manually generated test cases. It is found that generated automated and manual test cases are matching the same with each other. The authors claimed that the proposed testing technique gives reliability and efficiency completely.

In the future, combinatorial logic-oriented test cases can be generated to capture all the requirement specifications from other UML diagrams like Activity Diagram, State Chart Diagram, etc.

REFERENCES


Figure 1. Sequence Diagram of the revised CMSS