An Approach for Automatic Generation of Test Cases from UML Diagrams

Oluwatolani Oluwagbemi #1 and Hishammuddin Asmuni #2

# Faculty of Computing, Universiti Teknologi Malaysia, 81310 UTM Skudai, Johor, Malaysia
1 tolapeace@gmail.com, 2 hishamudin@utm.my

Abstract

The Unified Modeling Language (UML) is a standard notation used to model user’s requirements for object oriented software systems. With the growing adoption of UML by software developers and researchers, studies have begun to investigate how it can be used to conduct testing. Several approaches of UML-based software testing have been proposed where test cases are derived from UML diagrams based on efficient algorithms. However, these approaches still suffer from three major limitations namely; inadequate test coverage criteria, insufficient diagrams coverages and incompatibility with various UML tools. Therefore, this paper proposes an improved approach for generating test cases from various UML diagrams based on full coverage criteria. To achieve this aim, a robust method for extracting artefacts from the underlying diagrams of the software under test (SUT) was developed, where the artefacts are represented in an intermediate form using a tree and test cases are generated by traversing the contents of the tree. The proposed approach is fully automated and the generated test cases satisfied the criteria defined for the generation process. The novelty of this approach is its systematic rather than ad hoc test case generation from UML diagrams to achieve high test coverage.

Keywords: UML, SUT, requirements, software, test case generation

1. Introduction

Software testing is used to expose bugs in systems so as to ensure requirements conformance and verify that the system responds in the intended way. Before testing is conducted, test cases have to be generated. Test case generation is the foundation of any testing technique because, the effectiveness of the testing process depends on the quality of generated test cases (Zhu et al., 1997; Bertolino et al., 2007; Anand et al., 2013).

UML-based testing is a subset of model-based testing (MBT) where test cases are derived from the diagrams used to model user’s requirements (Dias Neto et al., 2007; Zheng and Bundell, 2007; Kaur and Kaur, 2013; Hussain and Frey, 2006; Xu and He, 2007). These diagrams include Activity, Class, Sequence, Statechart, and Use case among others. A UML-based testing technique compares the test cases (expected output) generated from the underlying model of the system under test (SUT) with features of implemented system (actual output). These underlying models contain artefacts drawn from the user’s requirements expressed in any of the software development modeling tools like ArgoUML, Rational Rose or Magic Draw to mention a few (Nebut et al., 2006; Brucker et al., 2011; Ogata and Matsuura, 2010; Santiago et al., 2006). According to Dalal et al., 1999, generation of test cases from UML models solely depends on three major elements namely: the UML model used to represent user’s requirements, the test case generation algorithm employed and the tools that generate supporting infrastructure for the test. In UML-based testing; hereafter referred to as UBT, test cases are generated from the artefacts extracted from the UML diagrams used to model requirements. This is achieved with a parser where the extracted artefacts are stored in an intermediate form.
using a tree or graph and its contents traversed to generate test cases. With increase in the complexities and sizes of software applications, more interest is been vested in object oriented design strategies in order to reduce the cost and time of testing as well as enhance software usability (Kaur and Vig, 2012). This has made UBT techniques more viable and widely adopted means of testing systems among developers and practitioners.

Conducting testing from UML diagrams have many advantages (Gutiérrez et al., 2006; Murthy et al., 2006; Sokenou et al., 2006). First, the specified requirements can be used to determine the functional and non-functional tests for the SUT. Secondly, the specified requirements concisely describe the basic features to test for. Thirdly, the testing process can be initiated as soon as the specification and design documents are ready. Lastly, testing and development activities can be executed concurrently. Several UML-based techniques for testing systems have been proposed (Kaur and Vig, 2012). In these techniques, test requirements and coverage criteria are exercised on UML diagrams.

Bangalore et al., 2000 generally divided software testing processes into four major tasks which include: modeling the software requirements, generating test cases, evaluating the test cases and measuring the testing process.

UBT offers techniques for the automatic generation of test cases from artefacts extracted from UML diagrams. We are interested in proposing an improved approach for generating test cases from UML diagrams based on the limitation of existing techniques. From literature, some of the challenges associated with existing UBT techniques are: (1) Generation of test cases from other UML models or diagrams (Samuel et al., 2006; Swain et al., 2012); (2) Inadequate test coverage criteria (Li et al., 2013; Patel and Patil, 2013); (3) Validation of techniques with various and larger sizes of UML diagrams (Fan and Wang, 2012; Hametner et al., 2011). This paper presents a contribution to the automatic generation of test cases based on the limitations of existing techniques which has generally led to the generation of incomprehensive or erroneous test cases. In addressing the first challenge, a method capable of extracting artefacts from any XMI file of UML diagrams known as element mapper was developed. For the second challenge, criteria for coverage across XMI files of UML diagrams were defined and implemented. For the third challenge, robust XMIs of UML diagrams for different software applications was used for test case generation in order to determine the scalability prowess of the proposed approach.

The inputs of the proposed technique consist of XMI files of UML diagrams. The proposed technique automatically generates test cases in order to aid the accurate analysis of the actual and expected outputs of the SUT. Whilst the expected output is obtained from the generated test cases, the actual output is obtained by running the implemented components of the SUT and mapping it with the sequences of generated test cases to detect system faults.

To validate or determine the performance of the proposed technique; XMI files of Activity, Class, Sequence, Statechart, and Use case diagrams for four different case studies were utilized. These case studies include: library application, student information system, bank ATM system and cellular phone system. Furthermore, the quality of the proposed technique was evaluated based on two basic metrics:

(a) Accuracy: It calculates the percentage of correlations between elements contained in the generated test cases and elements contained in the source diagrams.

(b) Percentage of coverage criteria: This measures the number of elements in the XMIs of the underlying UML diagrams, exercised in the generated test cases.

The remainder of this paper is organised as follows. Section 2 discusses related work. Section 3 presents the proposed technique. Section 4 presents the experimental setup and results. Section 5 discusses the implications and impacts of the results. Section 6 concludes the paper and suggests areas for future work.
2. Related Work

This section provides an analysis of the techniques utilized for generating test cases with UML diagrams. Firstly, the diagrams utilized in modeling user’s requirements are explained; secondly, the analysis of artefacts extraction methods are provided; thirdly, the transformation into intermediate representations are x-rayed; fourthly, the generation algorithms are discussed and lastly, existing test coverage criteria are enumerated.

In UBT techniques, test cases are derived from high-level specifications (Briand and Labiche, 2002; Offutt and Abdurazik, 1999; Zeng et al., 2009). Different UML tools stores or exports its contents in various file formats. These include; XML, XMI and .MDL file formats. A parser is usually required to extract artefacts contained in these formats as can be seen in the works of Samuel et al., 2007; Li et al., 2013; García-Domínguez et al., 2013; Prasanna and Chandran, 2011; Sawant and Shah, 2011; Nayak and Samanta, 2011; Asthana et al., 2010 and Oluwagbemi and Asmuni, 2014. Also, some authors have proposed techniques, capable of generating test cases from more than one UML diagram (Pilskalns et al., 2007; Sarma and Mall, 2007; Lamancha et al., 2013; Swain et al., 2013; Swain et al., 2010; Swain and Mohapatra, 2010; Sawant and Shah, 2011; Asthana et al., 2010). Trees and graphs were used to transform the extracted artefacts into intermediate representations and traversed to generate test cases as seen in the works of Samuel et al., 2007; Rapos and Dingel, 2012; Swain et al., 2013; Fan et al., 2009; Prasanna and Chandran, 2009 while in Swain et al., 2012; Pilskalns et al., 2007; Sarma and Mall, 2007; Samuel et al., 2008; Patel and Patil, 2013; Jena et al., 2014; Heinecke et al., 2010; Sawant and Shah, 2011; Nayak and Samanta, 2011; Khandai et al., 2011; Swain and Mohapatra, 2010; Boghdady et al., 2011; Priya and Malarchelvi, 2013.

3. Proposed Technique

As mentioned earlier, the proposed technique was developed based on the limitations of existing ones such as inadequate coverage criteria, lack of integration with other modeling tools and diagrams. The proposed technique is depicted in Figure 1 and its components described in subsequent sub-sections as follows:

3.1. Elements Mapper

Every XMI file has an Element node named RootNode. The RootNode has one Attribute node denoted as a1 for its name and v1 as its value. The RootNode has two child nodes (Element and Text). Element nodes consist of a child node while Text node consists of Root texts. The Element node has a child node of its own; while a Text node has its value as Child text. A step-by-step overview of the mapping process is presented and implemented with Java programming language. It creates and maps XMI files across UML diagrams.
Figure 1. Proposed Test Case Generation Technique

For each object in an XMI file, the Element node with XMI name for the object’s class is identified; this is known as the root node. Additional attributes describing any other objects within the namespaces in the XMI file are also identified; this is considered as the child node. Then, the node attributes and its descendant, element nodes based on the object’s attribute values and references IDs are identified. Every XMIObjec has an XMI name, which is the tag name for the XML element that represents the object in an XMI file. Each object has an id, uuid, and label, which identifies the object in XMI files. The xmi:id attribute for each object clearly indicate the XML elements in the XMI file that are objects.

3.2. Feature Selector

Figure 2 shows the profile metamodel of the mapped features which is a profile definition corresponding to the definition of a metamodel element. A ProfileElement contains one or more UMLConcept which specify the UML elements associated to the ProfileElement, a stereotype, and the NestedReferences that is required in defining the mapping links. The Nodes enhances the possibility of navigating through the UML elements. The definition of the mapping links between XMI files of UML diagrams was implemented with Java. The
mapper takes XMIs as input where the profiled metamodel of the XMI is used for the identification of XMI elements.

Figure 2. The Mapped Features

Five major features were used in tracing the elements of XMIs across UML diagrams. They are enumerated below:

(a) LinkIdef: This identifies the links of the ProfileDefinition and its associated metamodel elements. It has a root element, the topmost element and children can be LinkElements.

(b) LoopElem: This identifies the link between the ProfileElement and the metamodel elements. It has a left element, a right element and children which can be of type: Loop, IfLooped or New type.

(c) MainElem: This identifies the direct mapping of UML element attributes and metamodel elements. It has a left and right element.

(d) SerialTab: This is a mapping table which serializes the metamodel elements and attributes that do not exactly tally to UML elements. It has a left element, a right element, looped and un-looped links. This is required for defining mappings.

(e) ExtType: This is used to identify the type of extensions that defines a metamodel element with UML attributes which relate to different UML elements. It has a right element and Looped links which are needed to define mappings.
The selection of features from an XMI file is achieved from Metamodel Definition Profile. This profile defines the UML elements of the metamodel. Each element has its own attributes, relationships, types, and metaclasses. Excerpt 1 shows a metamodel definition profile which contains the UML elements. Features of UML diagrams are selected based on these elements which contain the Parent and Child nodes.

```xml
<modelelement name="RootNode">
  <attribute name="context" type="ref"/>
  <attribute name="id" type="data"/>
  <attribute name="name" type="data"/>
  <attribute name="ChildNode" type="ref" multiplicity="many"/>
</modelelement>
```

**Excerpt 1. Metamodel Definition Profile for Feature Selection**

### 3.3. Structure Identifier

Since the proposed approach aims to generate test cases from any UML diagram, it becomes imperative to develop a unified structure identifier capable of identifying the nodes and edges in XMI files of UML diagrams. Therefore, if an XMI file of any UML diagram is imported, the Elements Mapper is responsible for identifying the correct diagram source of the XMI file based on the descriptive attributes of the various UML diagrams and then, correlates these attributes to the corresponding diagram based on the running procedures. The Feature Selector refines the mapped elements to aid accurate identification of nodes and edges in an XMI file. In the proposed approach, mapping is executed by considering the nomenclature of the various UML diagrams. The contents of an XMI file consists of the metametamodel, comprising of the XML viewer, the element metamodel which provides the name and version of the XMI file, XMI contents which consist of the UML model and this model consists of XMI.id, UML diagram name, and Namespace.ownedElement. The requirement name and its attributes reside in the Namespace.ownedElement. Consequently, in the proposed approach, the metametamodel, XMI metamodel, model and namespace elements were used to identify the structure of XMI document across UML diagrams. In this research, the nodes connote the requirements while the attributes describing the expected functionalities of a requirement is known as an edge. Therefore, to identify the structure of an XMI file, labels of elements was used. The elements associated with the UML IDs are Nodes while the attributes of the elements are the Edges. Algorithm 1 was used to determine distinct nodes and edges of XMI files which accept XMIs as input.
Algorithm 1. Structure Identification

3.4. Dependency Flow Tree (DFT) Generator

This component is responsible for building a dependency flow tree based on the identified structure. The dependency tree is built based on the number of Nodes and Edges contained in an XMI file using Algorithm 2. It verifies that the Nodes and Edges corresponding to elements and attributes respectively.
3.5. Traversals

The design model was constructed using ArgoUML tool which support XMI file format. It includes class diagrams, sequence diagrams, state charts and so on. The shared model approach is used for test case generation. The same model is used for extracting artefacts as well as for test case generation. A transformation tool or some adaptor transformers that is embedded in the proposed approach can be used to translate abstract test case into an executable or concrete test cases which uses certain templates or mappings to ensure completeness between the extracted artefacts and generated test cases. Test cases are then generated by traversing the contents of the tree using Algorithm 3 which is based on four integrated coverage criteria described below:

3.5.1 Model-Flow Coverage Criteria: Model-flow criteria visit all transitions, transition-pairs, paths and parallel-transitions during test case generation.

3.5.2 Conditional-Flow Criteria: This criterion visits all statements, functions, decision/branch, condition, modified-condition/decision and atomic-condition during test case generation.

3.5.3 Element-Flow Criteria: This aid the generation of test cases from states, actions, entities, classes, use cases and other traceability links of artefacts contained in an XMI file.

3.5.4 Data-Flow Criteria: A data-flow criterion visits all elements and attributes with one-value, multiple-values, boundary-values, and pair-wise values to generate test cases.
Algorithm 3. Test Case Generation

4. Experimental Setup and Results

4.1 Experimental Setup

In order to evaluate the capabilities of the proposed approach, some sets of experiments were conducted on small to medium-sized software projects. In summary, the case studies employed had a total number of nine use case diagrams, seven class diagrams, nine state charts diagrams, ten sequence diagrams and seven activity diagrams across four different small to medium scaled software projects as indicated in Table 1. The approach was developed to extract artefacts and generate test cases from the XMI files of UML diagrams using the proposed algorithms. It is capable of generating reports containing the parsed artefacts with classifications indicating the number of nodes, edges and generated test cases. The experiments was ran on Windows machine, Intel(R) Core(TM) i7, 4 GB
In the first experiment, the structure identification algorithm was ran to determine if the proposed algorithm was able to correctly classify and provide the total numbers of Nodes and Edges in an XMI file. Then, the total number of correctly classified Nodes and Edges are compared to the total numbers of Nodes and Edges contained in the XMI source file in order to ascertain the level of correlations. Meaning, the number of covered and uncovered Nodes/Edges were computed.

In the second experiment, the DFT generation algorithm was executed to determine the syntactic and semantic correctness of the DFT contents. In this context, the syntactic correctness deals with the chronological arrangement of modelled requirements as contained in the source UML diagram of the SUT while the semantic correctness dealt with correct mappings of the element (requirement) to its corresponding user object (attribute). Also, to determine the reliability of the DFT algorithm, the consistency ratio of the contents of the DFT across all the XMI files of the various UML diagram was calculated. That is, whether the total number of requirements reflected in the DFT tallied with the total number of extracted artefacts as well as the total number of modelled requirements in the underlying UML diagram of the SUT.

In the third experiment, the test case generation algorithm was executed in order to ascertain the accuracy of the generation process. More precisely, the generated test cases were analyzed in terms of whether all the elements or output of the first three experiments were executed and correctly reflected in the generated test cases. The generated test cases were also compared to the modelled requirements in the source UML diagram to discover if there were missing requirements not represented in the generated test cases or whether the generated test cases have been distorted with inclusion of additional or erroneous requirements that were not modelled or contained in the source UML diagram.

In the last experiment, percentage coverage of the criteria implemented for the test case generation process were determined. In other words, the completeness of the generated

---

**Table 1. Description of Requirements for the Adopted Case Studies**

<table>
<thead>
<tr>
<th>Case Study</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Library Software</td>
<td>A librarian should easily create, update, and delete information about book titles, borrowers, loans, and reservations in the system. The system should be able to run on all popular Web browser platforms (Internet Explorer 5.1+, Netscape 4.0+, and so on). The system should be easy to extend with new functionalities</td>
</tr>
<tr>
<td>Automated Teller Machine (ATM) Software</td>
<td>The software should be developed to support computerized banking activities such as accepting relevant debit card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and print receipts</td>
</tr>
<tr>
<td>Cellular Phone System</td>
<td>This system allows users make calls and send short messages using the keypads from any geographical region</td>
</tr>
<tr>
<td>Ordering System</td>
<td>The ordering system must take care of sales information of the company and analyze the potential of the trade and inventory</td>
</tr>
</tbody>
</table>
test cases based on the test coverage criteria was determined and mapped with the modeled requirements in the UML source diagram to find out whether or not; the test cases were generated based on all the elements and other descriptive links or attributes of the UML diagrams.

4.2. Results

Table 2 shows the results obtained by the structure identifier across XMI files of the various UML diagrams. From the results, it can be observed that the proposed approach was accurately able to identify the fragments of structure contained in an XMI file for activity, class, sequence, state chart and use case diagrams respectively. The first fragment identified by the proposed technique is the XML viewer which generally contains the XMI, its header and content. The XMI contains the version of the XMI file, the developers of the UML tool utilized in requirements modeling and the time/date which the XMI file was built. The header contains the XMI documentation and metamodel. The documentation provides the XMI exporter and its version, responsible for the conversion of the modeled requirements into its XMI equivalent while the metamodel provides the name and version of the exported XMI file of the UML tool. On the other hand, the XMI content contains the UML model which houses the generated XMI IDs, UML diagram name (Activity, Class, State chart, Sequence or Use case), Specification, Root, Leaf and Abstract Nodes of the generated XMI files. Finally, the fragment identified by the proposed approach is known as UML: namespace.ownedElement which actually contains the artefacts otherwise known as requirements. The most important fragments of the identified structure that was utilized in the proposed approach are XMI content, UML model and UML: namespace.ownedElement. This is because, the XMI content contains the model that provides the precise name and nomenclature of the UML diagram utilized in the modeling process while namespace.ownedElement provides the artefacts that are to be parsed. In the namespace.ownedElement, the very first requirement at the top of the XMI file is considered to be the root node while other sub-nodes branching from the root node are known as node. The descriptive links or attributes that concisely explains the expected functionality of a node is known as edge denoted by name in the XMI file. Also, multiple nodes which are child nodes to a particular node will have the same value. This could be a decision node, conditional node, loop node, control node, merge/fork nodes, branch node etc., depending on the diagram undergoing the extraction process. With the correct identification of all the structure contained in an XMI file, it is easy to conclude that the structure identifier is robust enough to retrieve artefacts based on all the descriptive attributes of an XMI file from any UML diagram used in modeling user’s requirements.

The proposed approach was also able to efficiently extract artefacts contained in an XMI file of all UML diagrams. The main idea behind the efficient extraction and generation processes lies in the optimization of the proposed algorithms which aimed to reduce false classifications that culminates into erroneous extractions and generations respectively. Therefore, to determine if the criteria defined for the proposed approach was fully exercised during the generation process or not, the total number of extracted artefacts was compared to the total number of generated test cases.
Table 2. Results of the Structure Identification Algorithm

<table>
<thead>
<tr>
<th>Structure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML viewer</td>
<td>Contains the entire compressed fragments of the XMI file</td>
</tr>
<tr>
<td>XMI</td>
<td>XMI.version = 1.2</td>
</tr>
<tr>
<td></td>
<td>xmlns: UML = org.omg.xmi.namespace.UML</td>
</tr>
<tr>
<td></td>
<td>timestamp = Thu Jun 05 16:47:10 SGT 2014</td>
</tr>
<tr>
<td>XMI header</td>
<td>XMI documentation</td>
</tr>
<tr>
<td></td>
<td>XMI metamodel</td>
</tr>
<tr>
<td>XMI documentation</td>
<td>XML.exporter</td>
</tr>
<tr>
<td></td>
<td>XML.exporter version</td>
</tr>
<tr>
<td>XMI.exporter</td>
<td>Description: ArgoUML</td>
</tr>
<tr>
<td>XMI.exporterVersion</td>
<td>Description ID and date such description was revised</td>
</tr>
<tr>
<td>XMI metamodel</td>
<td>XMI.name</td>
</tr>
<tr>
<td></td>
<td>XMI.version</td>
</tr>
<tr>
<td>XMI content</td>
<td>UML model</td>
</tr>
<tr>
<td>UML model</td>
<td>XMI.ID</td>
</tr>
<tr>
<td></td>
<td>Name of the precise UML diagram</td>
</tr>
<tr>
<td></td>
<td>Specifications</td>
</tr>
<tr>
<td></td>
<td>Root</td>
</tr>
<tr>
<td></td>
<td>Leaf and</td>
</tr>
<tr>
<td></td>
<td>Abstract nodes</td>
</tr>
<tr>
<td>UML: namespace.ownedElement</td>
<td>Root node</td>
</tr>
<tr>
<td></td>
<td>Nodes</td>
</tr>
<tr>
<td></td>
<td>Edges</td>
</tr>
</tbody>
</table>

In Figure 3, the results of the proposed approach in terms of being able to generate test cases from various UML diagram types are shown. It shows the number of test cases generated from 6 different XMI files of UML diagrams. From the results, the number of generated test cases exactly tallied to the number of modelled requirements in the underlying diagram of the SUT. This indicates that, the proposed approach is capable of generating accurate test cases without repeating or erroneous elements. Similarly, Figure 4 shows the results for the extraction accuracy. The comprehensiveness of generated test cases solely depends of the efficiency of the parser which is responsible for extracting artefacts from the XMI files. From the results, it is clear that the proposed approach was developed based on an efficient parser that completely extracts all the artefacts from an XMI file. This component recorded 100% accuracy. In Figure 5, the proposed approach was again able to reflect comprehensive generation of test cases with adequate coverage criteria as shown in Figure 6. In Summary, the extraction of artefacts, test case generation and coverage criteria accuracy for the proposed approach was exciting with minimum of 98.00% accuracy in each category calculated by dividing the exercised elements by the total number of elements contained in the modelled requirements. Figures 7(a-d) shows the screenshot of sample test cases generated for the various case studies.
Figure 3. Generation of Test Cases from different XMI files of UML Diagrams

Figure 4. Overall Extraction Accuracy for the Proposed Approach

Figure 5. Overall Test Case Generation Accuracy for the Proposed Approach
Figure 6. Overall Criteria Coverage Accuracy for the Proposed Approach

Figure 7a. Sample Test Cases for Library Information System
Figure 7b. Sample Test Cases for ATM

Figure 7c. Sample Test Cases for Cellular Phone System
5. Discussion

The proposed approach was robust enough to generate complete test cases based on the defined coverage criteria. The overall performances of components for the proposed approach are shown in Table 3. In simulating the token flow during model execution, the parser attempts to scan through all the artefacts contained in the file to ensure that no artefact remained un-parsed. Unlike conventional parsers, the proposed parser has the capacity to pre-process XMI files based on all the descriptive attributes of all UML diagrams in order to aid generation of comprehensive test cases. It is also extended to other models that are not necessarily UML diagrams provided such model support XMI. Requirements were designated as nodes while the attributes describing the functionalities of each requirement were designated as edges. This research utilized the Argo UML tool due to the fact that, it is open source. Therefore, the model file is first exported to XMI format within the modelling environment of the tool (from File → Export XMI). It is worthy to note that, different modelling tools store their artefacts in various formats. After exporting in XMI format, the proposed approach converts the artefacts into a tree by identifying the requirements with specific elements as nodes and the user objects as edges, which stands for the attributes associated with each node. This research provided solutions to problems of existing UML-based testing case generation technique. These challenges are summarized as follows: Lack of parser that is capable of extracting artefacts from many UML diagram; inadequate test coverage criteria; Partial automation of techniques; Erroneous artefacts extractions and Incomprehensive test case generations. These challenges were important to address because, a diagram is not often enough to describe the requirements of end users. Secondly, requirements could be modelled with a diagram which is not supported by a particular technique. Hence, the need to develop an approach that is robust enough to generate test cases from any type of UML diagram. The results of the proposed approach demonstrated high capabilities and accuracy. The results also showed the automation prowess of the proposed approach. It was possible to automate the testing process; from the extraction of artefacts from an XMI file to the
generation of DFT and traversals of the DFT to generate test cases based on full coverage criteria. To support generation of concise test cases, the proposed approach was developed with unified syntaxes and constructs capable of visiting each descriptive attributes of an XMI file only once to generate test cases. Results of experiments for the proposed technique showed comprehensive and non-redundant generation of test cases on accuracy level of 98.71% with full coverage criteria.

### Table 3. Overall Performances of the Proposed Approach

<table>
<thead>
<tr>
<th>S/No</th>
<th>Algorithms</th>
<th>Accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Structure Identifier</td>
<td>100%</td>
</tr>
<tr>
<td>2.</td>
<td>Artefacts Extractor (Parser)</td>
<td>100%</td>
</tr>
<tr>
<td>4.</td>
<td>DFT Generator</td>
<td>100%</td>
</tr>
<tr>
<td>5.</td>
<td>Test Case Generator</td>
<td>98.71%</td>
</tr>
</tbody>
</table>

### 6. Conclusion and Future Work

A robust approach for UML-based testing was the focus of this research work. It was implemented with Java and fully automated. This has led to the efficient parsing of artefacts from any XMI files of UML diagrams. The proposed approach was validated using project requirements where five different UML diagrams were drawn to model requirements of four different software applications. The diagrams were then used to evaluate the performance of the proposed approach. It performed better that the existing ones in terms of its ability to parse artefacts from XMI of any UML diagram and model and generate test cases. This is an improvement over existing techniques. With the proposed technique, test cases generation becomes comprehensible since artefacts are parsed based on adequate coverage criteria and not randomly selected criteria. The average accuracy of the generated test cases is 98.00%. As for the future work, we hope to integrate our approach with other model file formats such as MDL. Also, a technique capable of generating test cases for non-functional requirements such behavior, software usability, security and reliability is worth investigation. Finally, the validation of this technique in industrial setting is considered as a limitation as well.

### References


