Jurnal Teknologi

Full Paper

AUTOMATIC GENERATION OF TEST CASES FROM ACTIVITY DIAGRAMS FOR UML BASED TESTING (UBT)

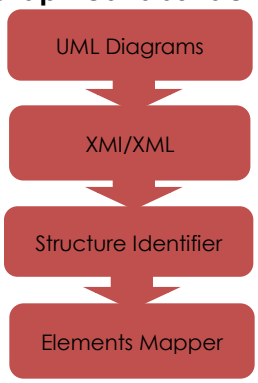
Oluwatolani Oluwagbemi, Hishammuddin Asmuni*

Soft Engineering Research Group (SERG), Faculty of Computing, Universiti Teknologi Malaysia, 81310 UTM Johor Bahru, Johor Malaysia

Article history
Received
15 April 2015
Received in revised form
29 September 2015
Accepted
12 November 2015

*Corresponding author
hishamudin@utm.my

Graphical abstract



Abstract

Activity diagrams are one of UML behavioural models suitable for system testing because it has the capacity to effectively describe the behaviours of systems under development. In this paper, a technique is proposed that generates test cases from activity diagrams by constructing an activity flow tree (AFT) which stores all the information extracted from the model file of the diagram through the help of a parser. Then, we applied an algorithm to generate test cases from the constructed tree. Test cases were generated based on the elements of activity diagrams such as activity sequences, associated descriptions and conditions. The proposed technique generated accurate test cases that completely tallied with the modeled requirements in the diagram. We utilized all-paths, basic pair paths, conditions, branches and transition criteria for generating test cases using ATM withdrawal operation software as a case study.

Keywords: MBT, TCG, UML diagrams, DFT, algorithms

1.0 INTRODUCTION

The complexities associated with system testing have led to the need for automatic generation of test cases. This is because, user's requirements are becoming larger and organizations are demanding for robust systems that can serve the needs of their customers irrespective of their geographical locations. Therefore, testing a fully integrated system with large requirements manually, can prove to be a difficult task. With the constant increase of system sizes, the concept of automatic design of system test cases is attracting serious research attention [1]. Test case generations are the foundation of any testing exercise [2]. Correctly generated test cases may not only detect errors in a software system, but also minimizes the high cost and time associated with software testing [3]. Unified Modeling Language (UML) is a de-facto standard for analyzing and modeling user's requirements otherwise known as design artefacts. With UML, software developers can easily analyze and visualize various views of a system. These views could be structural, behavioural or other related constraints envisaged or associated with the development processes. An activity diagram is used to describe all possible flows (data, control and objects) of executions and also good at describing the logical flow of the system under development [4]. This make it possible to generate test cases that captures all the expected functionalities or requirements of the system under development to aid conformance.

Testing based on design models has a lot of merits which mainly centers on two major facts. Firstly, testing can be initiated as soon as the requirements/design documents become available; thus, saving time, cost, and detecting errors early during the development span [5]. Secondly, test cases remain valid even when codes are slightly modified [4]. With these motivations, we propose a technique for test case generation using ArgoUML activity diagram, considering four major coverage criteria namely; all-paths, basic pair paths, conditions and branches coverage criteria. This is in a bid to engender the generation of accurate test cases

that tallies with the total number of modeled requirements.

The rest of the paper is structured as follows: section 2 provides the definitions of basic terms and concepts used in the proposed technique. Section 3 presents details of the proposed technique. Section 4 discusses the experimental design issues. Section 5 presents and discusses the results obtained from the proposed approach while Section 6 conclude the paper and suggest area for future research.

2.0 BASIC CONCEPTS AND DEFINITIONS

This section briefly describes UML activity diagrams by formally defining different elements of activity diagrams which will be used in the test case generation. Next, we describe the coverage criteria utilized for ensuring completeness of test cases during the generation process.

2.1 Activity Diagrams

Typical activity diagrams consist of about nine major elements namely; initialization (start), swimlanes, activity, branch, conditions (guard expressions), fork, join, merge and end (termination). All these elements can simply be integrated into nodes and edges. The nodes represents processes which include action states, activity states, decisions, swimlanes, forks, joins, objects, signal senders and receivers while the edges have to do with occurring sequence of activities, objects involving the activity, including control flows, message flows and signal flows [2]. Activity states and action states are represented with round cornered boxes. Transitions are shown with arrows. Branches are depicted with diamond shapes with one incoming arrow and multiple exit arrows, each labelled with a Boolean expressions. Forks or joins are shown by multiple arrows entering or leaving a synchronization bar. Activity diagram can be used to model the work flow or complex behaviour of systems or operations. The proposed technique focuses on UML activity diagrams which model both the work flows and operations of a system under development to generate test cases. Figure 1 shows an example of an activity diagram. In order to automatically analyze the activity diagram for artefacts extraction, the associations and concepts of an activity diagram are defined as follows:

Definition 1: An activity diagram is a tuple $D=\{A,T,F,C,a_I a_F\}$ where: $A=\{a_1,a_2,...,a_n\}$, are finite set of activity states; $T=\{t_1,t_2,...,t_n\}$ are finite set of transition states; $C=\{c_1,c_2,...,c_n\}$ are finite set of conditions (guard expressions) and c_i is in the corresponding transition t_1. $F\subseteq(A\times T\times C)\cup(T\times C\times A)$, is the flow relationship between the activities and transitions. $a_I\in A$, is the initial activity state; $a_F\in F$ is the final activity state. There is only one transition t such that $(a_I,t)\in F$ while $(t',a_I)\notin F$ and $(a_F,t')\notin F$ for any t'.

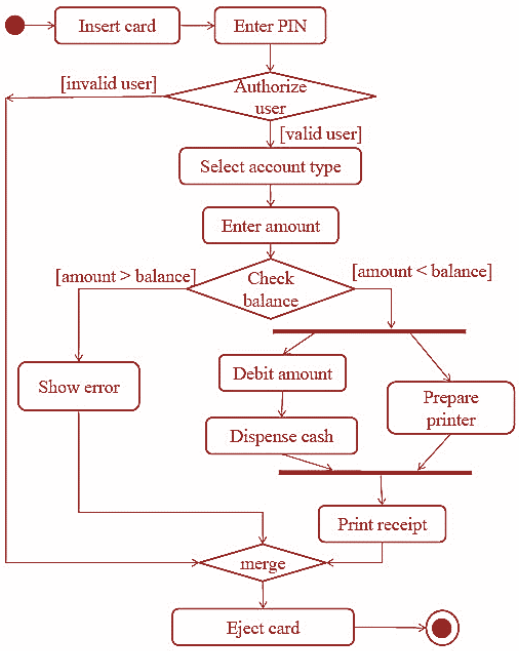


Figure 1 Activity diagram for ATM cash withdrawal operation

Definition 2: A test case *TC* generated from activity diagram *AD*, is the traversals of sequences of activities $\{a_1,a_2,...,a_n\}$, where $a_{i,(i=1,..,n)}$ are collections of activities within the diagram and type a_1 and a_n are start and end nodes respectively while, type $a_i,(i\neq1\wedge i\neq n)$ is either branch, fork, join, action, decision or merge. However, a transition must exist for any a_i and a_{i+1}($i=1..n-1$), otherwise, it is considered to be a parallel activity. Parallel activities and decision points must be taken into cognizance during the test case generation process. *Fork* and *join* actions signify the synchronized behaviour of activities, which requires that all parallel activities are executed during system testing. Contrarily, *branch* and *merge* activities signify the optional behaviour of actions which requires that only one of optional behaviours is executed during system testing.

Definition 3: Let $D=\{A,T,F,C,a_I a_F\}$ be elements of an activity diagram; the current state denoted by *CS* for any transition *t* is defined as **t, t**, representing pre-set and post-set of *t* respectively. For the latter, $\{A,T\in F\}$ will hold while for the former $\{T,A\in F\}$ will exist. Invariably, enabled (*CS*) represents the set of transitions that are linked with all out-going edge flows of *CS*. Therefore, enabled $(CS)=\{t\,|\,{}^*t\subseteq(CS)\}$. Similarly, firable(*CS*)

represents the set of transitions that are fired from *CS*; meaning, firable(*CS*) = $\{t \mid t \in \text{enabled}(CS)\}$.

Definition 4: Let $D = \{A, T, F, C, a_I a_F\}$ be elements of an activity diagram; the concurrent transition τ for current state *CS* is defined as the set of exercised or visited transitions $\{t_1, t_2, ..., t_n\} \in \text{firable(CS)}$ where,

$\forall i, j(1 \le i < j \le n), *t \cap t* = \text{NULL}$ and

$\forall t \in (\text{enabled (CS)} - \{t_1, t_2, ..., t_n\}, \exists i(1 \le i \le n) s.t * t \cap t* \ne \text{NULL}$

Definition 5: Activity flow tree (AFT) is a directed graph containing nodes and edges where each node and edge in the activity graph denoting all the extracted information from the model file of the activity diagram based on its elements. The nodes represent the processes which include action states, activity states, decisions, swimlanes, forks, joins, objects, signal senders and receivers while edges represent occurring sequence of activities as mentioned previously.

2.2 Test Coverage Criteria

Test coverage criteria are rules that enhances the generation of comprehensive test cases based on the number of elements to cover or visit within a diagram. The proposed technique utilized five major criteria namely; all-path coverage, basic-pair path coverage, condition coverage, branch (decision) and transition coverage.

a. All-path in an activity diagram is defined to be sequences of activities where an activity in that path is exercised exactly once. It also ensures that every loop in an activity diagram is exercised either zero or once in order to cover all iterations and transitions.
b. The basic-pair path coverage criterion ensures that test cases are generated from concurrent activities contained in an activity diagram. It is a complementary path emanating from a set of basic path where identical set of activities exist for each basic path. This is executed by visiting the concurrent nodes in forward and reverse successions at least once.
c. Condition coverage criterion ensures that test cases are generated from a true and a false result as well as all possible combinations of condition outcomes in each decision.
d. Branch (decision) coverage criterion generates test cases from each reachable decision made true by some actions and false by others.

3.0 PROPOSED TECHNIQUE

The starting point for UML-based testing is the extraction of artefacts contained in the test model which describes the expected behaviour of the system under test (SUT) and, determines the behaviour of each component or unit of the software. Models developed in formal methods are transformed into some textual representation supported by the modelling tool (usually XMI format). A model parser front-end reads the model text and creates an internal model representation (IMR) of the abstract syntax. A transition relation generator creates the initial state and the transition relation of the model as an expression, referring to pre-and post-states in order to extract artefacts. The proposed parsing process supports the use of UML for creating test models.

The structure of the SUT is expressed by composite or block diagrams, and behaviour is specified by means of state machines, activity or sequence diagrams. The parser front end reads model exports from different tools in XMI format and transforms it into intermediate representations. The intermediate model representation of extracted artefacts is capable of representing abstract syntax trees for a wide variety of formalisms. These artefacts form the basis of test case generation process and a conformance relations method is defined to determine the consistency between the test model and test cases. The root of the XMI file can consist of several child-nodes. However, the area of interest for the extraction process is principally the model sub-tree, which houses the structure that contains all the artefacts to be parsed. The elements of the sub-tree are descriptions of the expected functionalities of the SUT. Furthermore, there are some special tags of XMI such as the documentation and extension or stereotype which aids the insertion of new data or information about the model without distorting the original model contents. The proposed extraction process has the capacity of retrieving additional artefacts included in the model or model file. As shown in Figure 2, structure identifier is responsible for the extraction of artefacts from the extension tags which is mapped with the contents of the entire file. Given that every child-node of the root represent a classifier or an association in the meta-object facility (MOF) meta-model, the proposed extraction method is capable of serializing these descriptive elements of XMI file with the help of a conformance checker which identifies every MOF classifier as a separate child-node of the root and nest classifiers as a child-node representing their composition characteristics.

For large software projects, the complexity level can be high, hence the need to design a scalable extraction algorithm. Features that are not meant to be nested, like the name of a class, are identified by XML properties of the node (that could also be a node without identifier). Every property belongs to its node and it is not related with any other node, since it describes a characteristic of that specific node. This assertion helps in extracting distinct artefacts between elements. For the extraction process, XMI values were considered as the elements which can have identifiers (IDs) or without an identifier (ID).

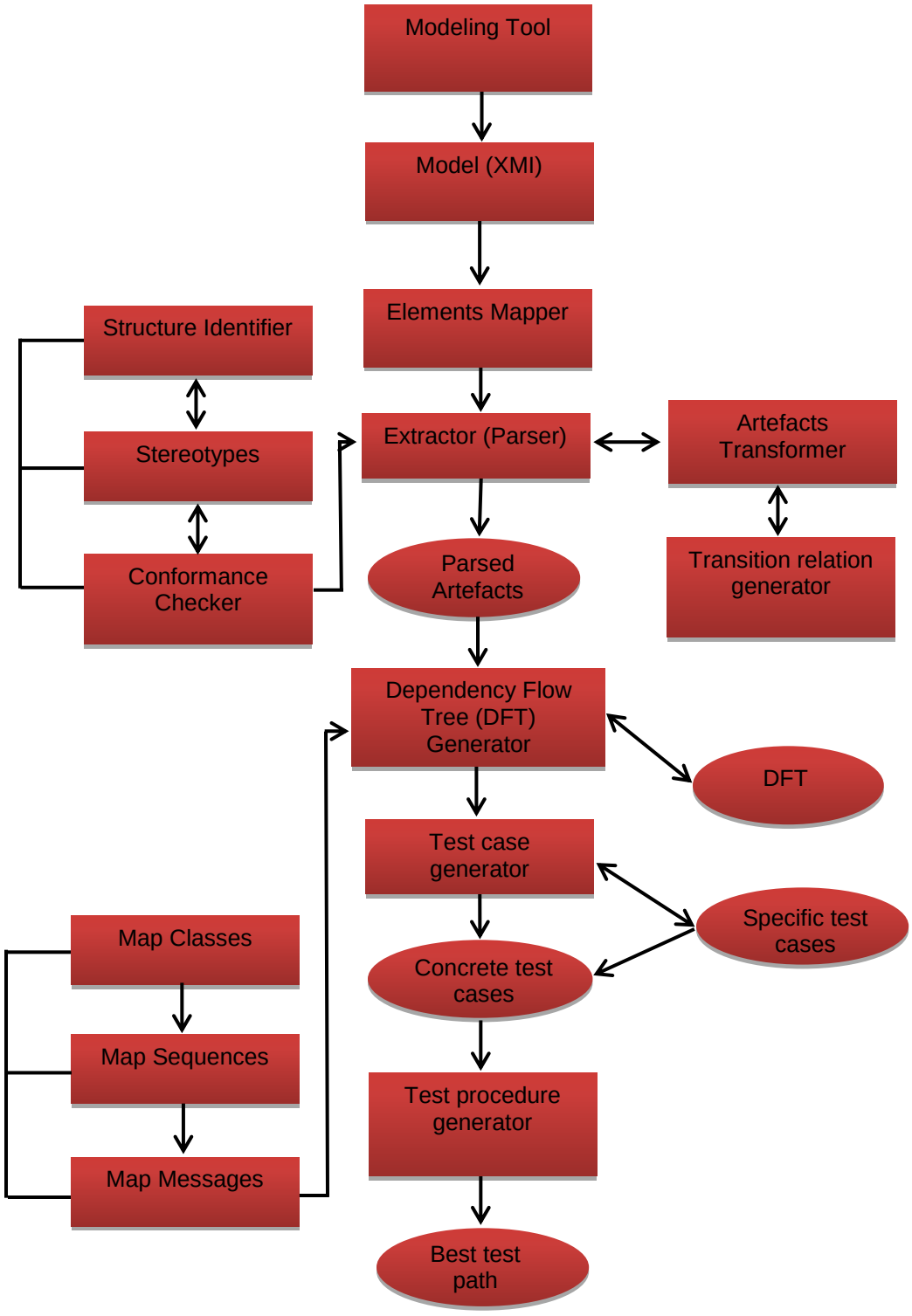


Figure 2 Proposed technique

To extract artefacts, the name and value of an element is identified and parsed. This was achieved by exploring the elements IDs to create global and local identifiers to uniquely identify the nodes and edges of an XMI file. This way, every node could have a property that uniquely identifies it and reachable without relying on its path from the root. These IDs were useful for matching the contents of the dependency flow tree (DFT) for conformance checking. A major merit of the proposed method is that, elements can be

identified and parsed whether such have IDs or not. For elements without IDs, the strings contained in the namespaces of the elements are extracted and an XMI tree generated automatically.

3.1 The Parser

The parser deals with the process of extracting the artefacts into concrete target test cases suitable for test execution. To generate comprehensive test cases, it is important to develop robust artefacts extraction strategies capable of supporting the construction of test artefacts from UML models for the generation of target-based test cases. To solve this crucial concern, a new extraction-based test derivation technique, called the Artefacts extractor is proposed. The essence of this is to ensure that the artefacts extracted from the UML models are refined to reflect a chronological mapping between the source model and XMI file. That is, artefacts in XMI files are mapped to correlate to the requirements in the UML model so as to generate comprehensive test cases. The artefacts extractor refines and provide the details associated with the methods and processes of UML-based test design and generation, and focuses on how to link testable model artefacts into useful test cases, so that they can be used to generate test scenarios, test sequences, test operations and test elements. The artefacts extractor utilizes series of correlation steps during the extraction process in order to ensure completeness of the artefacts with respect to contents of the UML models and model elements at different modeling levels.

3.2 The Structure Identifier

The primary objective of the proposed approach is to aid the concise extraction of artefacts from formal models so as to enhance the generation of test cases that provides an efficient way of defining transformations, mappings, and refinements of the extracted artefacts. This was accomplished by using ArgoUML as the modeling tool to model requirements of users using activity diagram. The structure identifier is therefore, responsible for detecting additional elements added to an existing diagram or XMI file for extraction. It consist of a meta-modeling mapping technique which provides an avenue of using the abstract syntax and semantics of additional model variables for the extraction process. This allows the extraction to happen at the metamodel level. The transformation process is defined at the metamodel level, while the transformation execution is implemented in the model level. The meta-modeling is the key in this structure. A transformation tool is then saddled with the responsibility of reading, writing and transforming the contents of XMI file. Secondly, the transformation rules are based on MOF but for more robust hierarchical meta-modeling architecture.

3.3 XMI Model Transformer

The basic advantage of the proposed model transformer is in its ability of specifying the transformation strategy at the meta-model level. Figure 3 shows the basic structure of the proposed model transformer. A transformation engine takes XMI of the source model as input, execute them through set of rules to generate an output model in a textual format. This means that, the output of the transformation engine is a refinement of the XMI. A XMI is regarded as a file containing the textual descriptions of the model elements that are in consonance with a metamodel element via the instantiation relationship. Metamodel based transformations use only the elements of the meta-models, thus the transformation description is expressed in terms of the two meta-models. The XMI file of the underlying model of the SUT was used to extract the textual syntax defined by the metamodel. Since XMI provides a textual syntax in a form that most programming languages conform to, the internal storage of the extracted artefacts was built on tree, where the nodes and edges in the tree are constructs of the XMI file. However, to capture more complicated constructs like loops, mathematical formulations were used to depict the relationships of different constructs. Considering its tree structure and such relationships, the transformation process was executed on the Elements and Attributes which signify Nodes and Edges. This enhances the transformation of the abstract syntax and semantics of the XMI file. Based on the proposed method, a template based model transformation was designed to obtain the information necessary for test cases from the artefacts repository. A set of interchangeable templates can be provided for model transformation between different versions of XMIs. Meta-modeling is a critical part of the transformation approach. It provides a mechanism of unambiguously defining modeling languages, ArgoUML in our case. It is the prerequisite for a model transformation tool to access and make use of the models. The fact that XMI notations have textual syntaxes makes it possible to extract artefacts in an unambiguous manner. The method utilized in this research is based on trees as abstract data types.

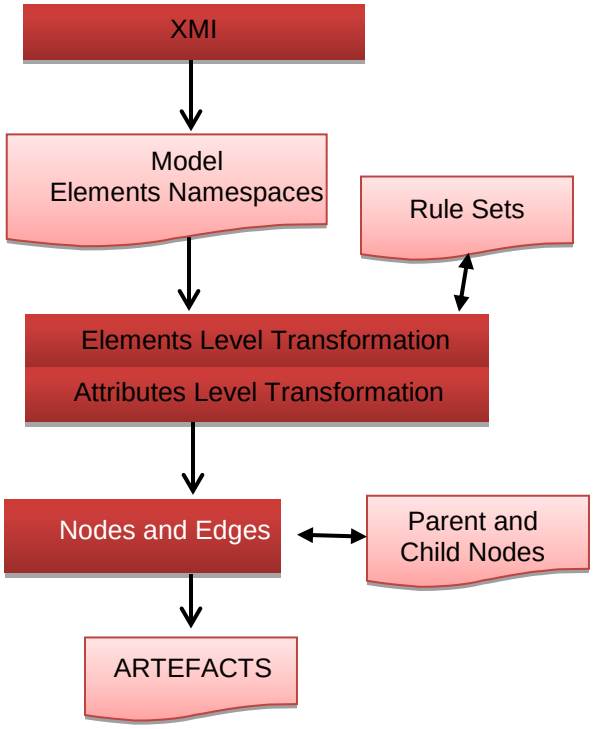


Figure 3 XMI model transformation process

In this context, a requirement is designated as node while the attributes describing the functionalities of each requirement are considered to be edges. The proposed approach accepts input in XMI format. As mentioned previously, this research utilized the ArgoUML tool due to the fact that, it is open source. Therefore, the diagram is first exported to XMI format. After exporting in XMI format, the proposed approach converts the artefacts into a tree by identifying the requirements with specific xmi:id as nodes and the xmi:dref as edges. Each requirement with their respective attributes is identified by the pseudo code described below:

```
startRequirement(String rName,
// "r = requirement" name
            Attributes atts//
endRequirement(rName)
```

The "start and end requirement" shows that, each requirement and its attributes should be visited once to extract all its artefacts. This cycle is completed for all the requirements and attributes contained in the XMI file. Therefore, the first task in test cases generation is to develop a parser that is capable of extracting artefacts from model files of UML diagrams. After parsing artefacts, an output is generated. Algorithm 1 shows the artefacts extraction algorithm while Listing 1 shows the rules.

```
INPUT:
1.  N = Artefacts lists from XMI file.
    s: an ID of the first artefact.
    d: an ID of the last artefacts.
2.  T (V, E) be a tree graph with a set of V nodes
    and set of E edges; where V, E is the number
of
    requirement and attributes respectively.

PARAMETERS:
3.  R(s,d), is a nonnegative number that stands for
    the artefacts where "s" start node and "d" is
    last node.
4.  i, j; loop index, T (2, i) is array of vertexes
source;
    T (3, j) is array of vertexes attributes.
5.  T (3, j) is array of edges; T (3, i) is array of
Nodes.
6.  For each node and edge VE (i, j), Extraction is
    executed from the shortest path of the origin
to
    final node.

OUPUT:
7.  R: Extracted artifacts.
8.  INITIALIZATION:
9.  // All nodes are identified with their
    corresponding edge //Applying Rule 1//
10. For each edge, do the operation in two steps
    as follows:
11. set ArtefactsArray[1... node-1] =
    ArtefactsArray(node) = n; //Applying Rule 2//

BEGIN
12. for all N; j = first to last edge // j is set to
    be the artefacts at edges; i = is set to be
    the artifacts at node
13. if (T(2 , j) = i ) // k is set to be end of node
    of edges.
14. Artifacts = ArtifactsArray(i) + T(3i, j);
15.                 end if
16.             end for
17.         end for
18.       for i = first to last edges
19.    while (the origin (k) is the same in T )
20. if (T(3 , i,j) = ArtefactArray(i)⊆
    ArtefactArray(j)//Applying Rule 3//
21. T(k) = T(2 , i); Extract artifacts//          else
22.  i = i + 1;  k = T(i , j);
23.             end if
24.          end while
25.       end for
26. END
```

Algorithm 1 Artefacts extraction

a. **Rule 1:** If the current element is a node, the algorithm pauses and extracts artefacts.
b. **Rule 2:** If the artefacts of the current node have been extracted, the algorithm checks to ensure that, all the attributes has been visited and extracted.

c. **Rule 3:** The extracted artefacts are then represented in an arraylist.

Listing 1 Rules Set

3.4 DFT Generator

An XMI file contains many references to other elements inside the document, but it can also have references to elements that appear in other documents. This necessitated the proposal of the dependency flow tree (DFT) generator algorithm aimed at storing all the extracted artefacts before test case generation. XMI elements contain attribute names and variables that are useful for test case generation. For the extraction process, the elements like <UML:Namespace.ownedElement> and <UML:TaggedValue. type> was used. Algorithm 2 depicts the DFT generator concept.

1: Input: Extracted artefacts;
2: Ouput: DFT
3: initTgt.DFT XMI file [L]
4: for i = 1 to L do
5: XMI file (createNodes.elements; edges.attributes)
6: addElement (Nodes [edge.attributeSize])
7: end for
8: ModelTgt(visitList (element.descriptions, DFT)
9: for i = 1 to L do
10: XMI file = DFT
11: end for

Algorithm 2 DFT Generation

The DFT generator algorithm is responsible for building a dependency flow tree based on the extracted artefacts. The dependency tree is built based on the number of Nodes and Edges contained in the XMI file and using the algorithm verifies that the extracted artefacts of a given XMI file are well structured with their respective Nodes and Edges.

3.5 Test Case Generator

The test cases are generated based on coverage criteria using Algorithm 3. It generates test cases with either pairwise or triple coverage. Pairwise coverage is sufficient for good test case generation. The seeming ineffectiveness of test case generation techniques has to do with low coverage criteria. In the proposed approach, more coverage criteria were used and mapped to avoid redundancy during test case generation.

1: Input: Dependency flow tree (DFT).
2: Output: Set of test cases.
3: elementStack=Ø
4: userObjectStack=Ø
5: decisionStack=Ø
6: resultStack=DFT.rootNode
7: for all nodes of DFT do
8: while DFT.node==expextedOutput.node do
9: userObjecteStack.push(child node of DFT.node)
10: end while
11: end for
12: for all elements of userObjectStack do
13: repeat
14: if elementStack[top] ≠ alt.node ||loop.node|| par.node|| break.node ||DFT.EndNode then
15: resultStack.push(elementStack.pop) == {push elementStack top element in resultStack and pop the top element from elementStack}
16: resultStack.push(child node of resultStack[top] in DFT) {push child node of resultStack top element into resultStack.}
17: else if elementStack.[top] == DFT.EndNode then
18: Mark the last decision node in resultStack as visited
19: while resultStack[top] ≠ decisionStack[top] do
20: resultStack.pop {pop the top element from resultStack.}
21: end while
22: decisionStack.pop {Pop the top element from decisionStack.}
23: else if elementStack[top] == ||alt.node|| break.node || loop.node then
24: decisionStack.push(elementStack.pop) {pop top element of elementStack and push it into decisionStack and resultStack.}
25: for all Child nodes of resultStack[top] in DFT do
26: if Child node is not Marked Visited then
27: elementStack.push(Child Node) {push all child nodes of resultstack[top] in DFT, if marked as visited and insert into elementStack}
28: end if
29: end for
30: else if elementStack[top] == par.node then
31: resultStack.push(resultStack[top] and all its child nodes in DFT
32: runningStack.push(child nodes of resultStack[top] in DFT)
33: else if elementStack.top == UserObjectStack.CurrentNode then
34: resultStack.push(elementStack.pop) {Pop the top element from elementStack and push it into resultStack.}
35: Print resultStack
36: if decisionStack ≠ Ø then
37: while resultStack[top] ≠ decisionStack[top] do
38: resultStack.pop {pop the top element from resultStack.}
39: end while
40: end if
41: if decisionStack ≠ Ø then

Algorithm 3 Test Case Generation

4.0 EXPERIMENTAL DESIGN

A prototype tool named as UBTCG (UML based test case generator) has been developed. The implementation was executed with Java language (Java 2) using NetBeans IDE 6.1. Input of UBTCG is the XMI files of UML diagrams. Activity diagrams for various software specifications were drawn and subsequently exported in XMI file format. UBTCG visualizes the dependency flow tree and display generated test cases as output. UBTCG consists of two main components: DFTGenerationUnit and TestCaseGenerationUnit. DFTGenerationUnit first parses the XMI of UML diagrams and then converts it into a dependency flow tree. Taking DFT as the input, TestCaseGenerationUnit traverse the DFT and generate test cases. The two components: DFTGenerationUnit and TestCaseGenerationUnit are described below:

4.1 DFT Generation Unit

This component parses the XMI representation of UML diagrams using an improved parser. This parser is capable of reading and extracting information from the XMI file or document of any activity diagram to conduct model-based testing. This component comprises of two main classes: ImprovedParser, and DFTGenerator. The ImprovedParser class implements the event-handlers startElement(), endElement(), characters(), and endDocument() to interface with the parser. In the event-handler startElement(), the tagged elements starting with the names are: "ownedAttribute", "lifeline", "fragment", "operand", "guard", "specification", "argument", "body", "ownedParameter", "message" and "packagedElement", "ownedOperation", "ownedBehavior", "guard", and "ownedParameter". Depending on type of tagged elements, they were categorized as "MessageEvent", "Fragment", "CallEvent", "Object", "Class", "Lifeline", "Operand", "Message", "Operation", "Parameter", Transition", and "Guard". When multiple tagged elements start with the same name, then the value of the attribute "XMI type" is used. For example, tagged elements specifying the element name and call event start with the same name "packagedElement". In order to distinguish them, the attribute "XMI type" has the value as "uml:CallEvent" or "uml:Element". For each processed tagged element, the associated meta-information of the UML diagram is retrieved from the parser and stored by means of instance variables: Id, Name, ClassId, ObjectId, FragmentType, Guard, SendEventId, ReceiveEventId, CallEventId, MessageType, OperationId, MessageId, and LifelineId of the class named Parser. After processing the tagged element, only relevant variables would have meaningful values, and the rest would have the null value.

4.2 Test Case Generation Unit

The main task of this component is to generate test cases by traversing the dependency flow tree in test cases. It consists of two main classes: ElementListCreator and TestCaseDisplayer. The code of TestCaseDisplayer was developed by taking two arraylists into cognizance DFTNodeList and DFTEdgeList as the input; the ElementListCreator object creates a list of elements by using node and edge specifications. To display the test cases in a chronological form, different methods such as start tree(), addIn(), and end tree() were implemented in Java to create test cases by calling the methods getElementSource(), getTree(), and writeTreeToFile(). UBTCG supports different menu options for selecting an XMI file, displaying XMI file, parsing selected XMI file, starting conversion, and display dependency flow tree in textual forms. In this usage scenario, the XMI file is selected and converted into a DFT and the DFT is traversed to generate test cases. The option "generate tree" is chosen to view the DFT and the option "generate test cases" is selected to view the generated test cases.

5.0 RESULTS AND DISCUSSION

The percentage of criteria coverage is used to evaluate the accuracy or quality of test case generation approaches [6, 7]. The dataset used for the experiment consisted of the XMI file generated from the activity diagram shown in Figure 1. The formula for calculating the percentage of coverage criteria is depicted in Equation 1 [8]. The accuracy of the proposed approach is shown in Table 1. It indicates the number of elements contained in the UML diagram which were exercised in the generated test cases. From the results, it is clear that the proposed approach was comprehensively able to generate test cases based on all the criteria defined for coverage. Further analysis shows that the test cases are consistent and conformed to all the artefacts contained in the XMI file (Figure 4).

$$E_C = \left(\frac{E_{tcs}}{E_{tsUML}} \times 100 \right) \quad (1)$$

E_C : Elements coverage

E_{tcs} : Number of elements exercised in the test cases

E_{tsUML} : Number of elements in the UML diagram

Table 1 Accuracy of the proposed approach

Elements	E_{tcs}	E_{tsUML}	Accuracy
Activities	10	10	100%
Branches	3	3	100%
Conditions	4	4	100%
Fork	1	1	100%
Join	1	1	100%
Basic path	19	19	100%
Total	38	38	100%

The approach was also tested with other activity diagrams for three different software applications, namely library, ordering and cellular phone systems. The proposed approach displayed exciting results by generating comprehensive test cases as well. The overall results including that of ATM software are displayed in Figure 4.

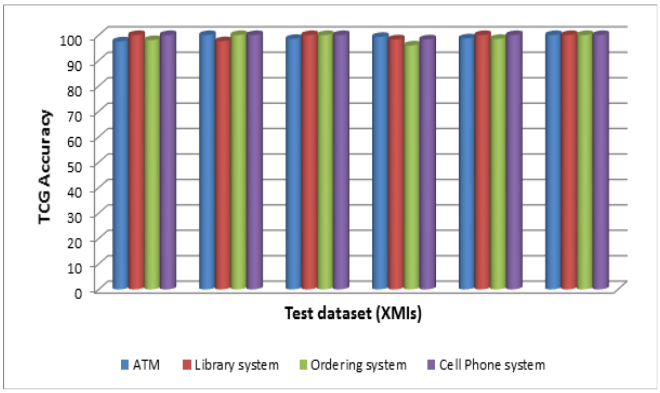


Figure 4 Test case generation from activity diagrams for other software applications

The research contributions borders on adequate test coverage criteria with high accuracy. A modification of the concept of depth-first-search traversals was used to generate test cases. In trees, it is possible for a node to have more than one parent especially when different edges, starting at different nodes exist. Because the parent node is not unique, a node's set of siblings will also not be unique. Worse still, it is possible for a node to simultaneously be the sibling and the child of the current node. In these cases, a depth-first-search algorithm will be unable to carry out traversals effectively because it will not be able to visit the entire node since it is meant to dwell within the connected component from the root node. Therefore, to address this problem, a modified algorithm was developed to visit all the nodes of a tree and their corresponding edges exactly once. To avoid redundancy, visited nodes are marked. Table 2 shows comparisons of the proposed approach with existing ones.

Test case generation is an important part of testing. It determines to a very large extent the success of the overall testing exercise. This paper proposed an approach for generating unambiguous test cases from XMI file of UML diagrams. The proposed approach overcame the difficulties of attributing all information in XMI tagged elements to extract artefacts. The transformational process associated with the DFT into test case generation process is automated. The proposed approach provided solution to limitations of existing systems such as low accuracy and erroneous or redundant generation of artefacts or test cases. The prototype tool UBTCG was implemented based on the descriptions of the proposed approach and the tool is capable of generating DFT and test cases automatically. Screenshots of the proposed tool for DFT and test case generations for the activity diagram in Figure 1 are shown in Figures 5 and 6 respectively.

5.1 Comparison with Previous Approaches

In Table 2, descriptive analysis of some of the activity diagram-based test case generation approaches are enumerated with their limitations. However, in the proposed technique, an approach for comprehensive generation of test cases from XMIs of activity diagrams is presented based on a conformance checker that identifies unwanted or distorted strings of the name attributes of an XMI file and structure identifier which distinctively identify the various numbers of nodes and edges contained in an XMI file. Secondly, algorithms were designed to extract the key information contained in the model's XMI file. Thirdly, we visualized the refined and mapped contents of the XMI file to a directed tree and test cases were automatically generated by traversing the tree. Activity diagrams of typical software applications were converted to their XMI equivalent and used to determine the performance of the proposed approach. A prototype tool was implemented and results show that, the proposed approach can generate test cases automatically that completely conforms to the total number of modelled requirements.

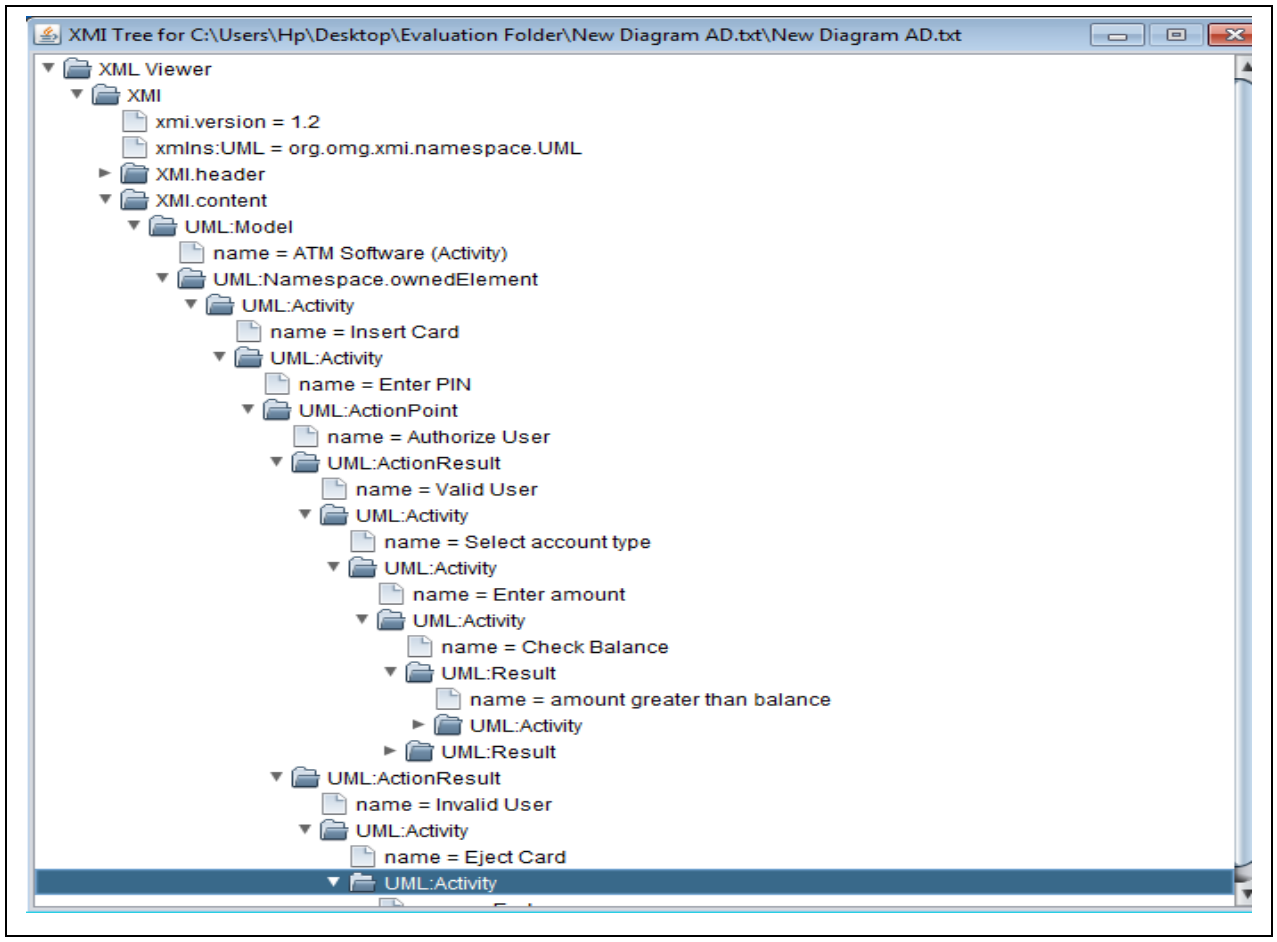


Figure 5 Dependency flow tree generation screenshot

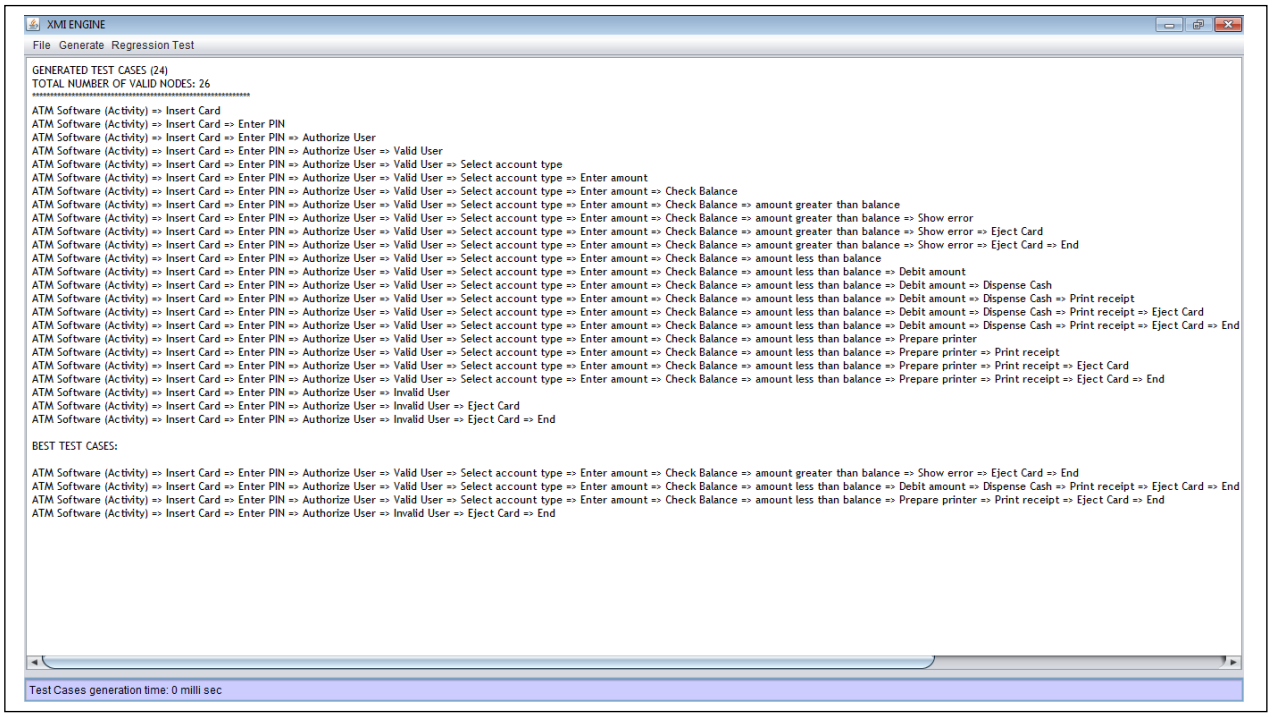


Figure 6 Generated test cases screenshot

Table 2 Descriptive summary of test case generation approaches based on activity diagrams

S/No	Source	Tool Used	Criteria	Validation	Prototype	Limitation
1	Li *et al*. [9]	Rational Rose	Basic-path coverage	ATM	√	Satisfaction of more coverage criteria
2	Patel and Patil, [10]	UML 2.0	Basic, simple and activity path criteria	ATM	√	The generalization of test case generation algorithm to cater for various the various test coverage criteria within the same test derivation framework
3	Farooq *et al*. [11]	UML 2.0	Sequential and concurrent coverage	Enterprise customer commerce system	×	Expansion to other aspects of activity diagram such as data flow and high level design artifacts with adequate coverage criteria
4	Jena *et al*. [12]	UML 2.0	Activity coverage criteria	ATM	×	Development of an implemented tool
5	Heinecke *et al*., [13]	UML 2.0	All-path	Account report system	√	Integration with other UML tools
6	Pechtanun and Kansomkeat, [14]	UML 2.0	Path	ATM	×	Implementation of a support tool that can generate test cases from other UML diagrams
7	Chen *et al*.[15]	UML 2.0	Activity, transition, key path and interaction coverage	ATM	√	Validation with large- scale requirements
8	Nayak and Samanta [16]	UML 2.0	Control flow	Cell Phone System (CPS)	×	Extension to enable generation of test cases corresponding the test vectors and with other UML models with dependencies
9	This work	ArgoUML	All-Path, All basic-pair path, decision and condition coverage criteria	ATM, Library, Ordering & Cell Phone systems	√	Extension to other diagrams and validation in real-life scenario

6.0 CONCLUSION AND FUTURE WORK

In this paper, the proposed approach utilized four different coverage criteria for the generation of test cases. A parser algorithm was implemented to parse artefacts from an activity diagram model file to generate a DFT which contains the extracted information. The DFT consists of nodes and edges. Consequently, a modified search algorithm was developed to generate test cases from the DFT by visiting all the nodes and edges exactly once. To avoid redundant generation of test cases, visited nodes and edges are marked to signify that, they have been visited. To further test for robustness of the proposed approach, other activity diagrams were drawn for different software applications and the proposed approach was able to efficiently generate test cases from all the activity diagrams. In the future, it will be expedient to validate this approach with experts in real setting and extend the approach to cater for other diagrams.

References

[1] Sharma, M. and Mall, R. 2009. Automatic Generation of Test Specifications for Coverage of System State Transitions. *Information and Software Technology*. 4(51): 418-432.

[2] Linzhang, W., Y. Jiesong, Y. Xiaofeng, H. Ju, L. Xuandong, and Z. Guoliang. 2004. Generating Test Cases from UML Activity Diagram based on Gray-box Method. *11th Asia-Pacific Software Engineering Conference (APSEC04)*. Busan, Korea. Nov. 30-Dec. 3 2004. 284-291.

[3] Li, H. and L. C. Peng. 2007. Software Test Data Generation using Ant Colony Optimization. *International Journal of Computer, Electrical, Automation, Control and Information Engineering*. 1(1): 137-140.

[4] Swain, R. K. Panthi, V. and Beher, P. K. 2013. Generation of Test Cases using Activity Diagram. *International Journal of Computer Science and Informatics*. 2(2): 2231-5292.

[5] Swain, S. K. and Mohapatra, D. P. 2010. Test Case Generation from Behavioral UML Models. *International Journal of Computer Applications*. 6(8): 6-11.

[6] Lam, S. S. B. Raju, M. L. Ch, S. Srivastav, P. R. 2012. Automated Generation of Independent Paths and Test

Suite Optimization using Artificial Bee Colony. *Procedia Engineering*. 1(30): 191-200.

[7] Sun, C. A., B. Zhang, J. Li. 2009. TSGen: A UML Activity Diagram-Based Test Scenario Generation Tool. *International Conference on Computational Science and Engineering (CSE'09)*. Vancouver, BC. 29-31 Aug. 2009. 853-858.

[8] Far, B. H. 2010. SENG 421: Software Test Metrics. Chapter 10, Lecture Notes, Department of Electrical & Computer Engineering, University of Calgary, Canada. [Online]. From: http://www.enel.ucalgary.ca/People/far/Lectures/SENG421/10/. [Accessed on Jan. 15, 2015].

[9] Li, L. Li, X. He, T. Xiong, J. 2013. Extenics-based Test Case Generation for UML Activity Diagram. *Procedia Computer Science*. 17(1): 1186-1193.

[10] Patel, P. E., N. N. Patil. 2013. Testcases Formation Using UML Activity Diagram. *International Conference on Communication Systems and Network Technologies (CSNT2013)*. Gwalior. 6-8 April 2013. 884-889.

[11] Farooq, U., C. P. Lam, H. Li. 2008. Towards Automated Test Sequence Generation. *Australian Software Engineering Conference (ASWEC2008)*. Perth. WA. 26-28 March 2008. 441-450.

[12] Jena, A. K., S.K. Swain, D. P. Mohapatra. 2014. A Novel Approach for Test Case Generation from UML Activity Diagram. *International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*. Ghaziabad. 7-8 Feb. 2014. 621-629.

[13] Heinecke, A., T. Bruckmann, T. Griebe, V. Gruhn. 2010. Generating Test Plans for Acceptance Tests from UML Activity Diagrams. *17th International Conference on Engineering of Computer Based Systems (ECBS2010)*. Oxford. 22-26 March 2010. 57-66.

[14] Pechtanun, K., Kansomkeat, S. 2012. Generation Test Case from UML activity Diagram based on AC Grammar. *International Conference on Computer & Information Science (ICCIS)*. Kuala Lumpur. 12-14 June 2012. 895-899.

[15] Chen, X., N. Ye, P. Jiang, L. Bu, X. Li. 2011. Feedback-directed Test Case Generation based on UML Activity Diagrams. *5th International Journal of Secure Software Integration & Reliability Improvement Companion (SSIRI-C2011)*. Jeju Island. 27-29 June 2011. 9-10.

[16] Nayak, A. and Samanta, D. 2011. Synthesis of Test Scenarios using UML Activity Diagrams. *Software & Systems Modeling*. 10(1): 63-89.