Automatic Test Generation from UML Sequence Diagrams for Android Mobiles

Anbunathan R
Test Manager and Research Scholar
Bharathiar University
Coimbatore, India
anbunathan.r@gmail.com

Anirban Basu
Professor, Department of CSE
APS College of Engineering
Bangalore, India
abasu@anirbanbasu.in

***Abstract*— Automatic generation of test cases for functional testing is a challenging task. It involves complex sub tasks such as capturing user scenarios, parsing user scenarios to generate test cases, ensuring test coverage criteria, test script generation from test cases, test execution and report generation. In this paper, a method is proposed to generate test cases after parsing Sequence diagram and thereafter generating XML based test cases, and subsequently APK based test scripts for Android mobiles. A tool called Virtual Test Engineer (VTE) has been developed based on this method, which is used for testing several Android applications. In this paper it is explained how navigation through menu tree of Android mobile is achieved through a menu tree data base. The effectiveness of this method is discussed from experimental results. Comparison has been made of the proposed method with others and the proposed method has been found to be more effective.**

Keywords*— Android mobile testing; Test framework; Test automation; Menu tree navigation; Test case generation; Test script generation; APK generation; XML generation; Model based testing.

I. Introduction

Testing involves activities such as identifying test cases from requirement document, designing test cases to ensure test coverage, writing scripts to automate test execution etc. From requirement document, normal and alternative flow test cases are identified. Test design techniques are used to identify test cases from requirements to ensure test coverage. Test automation involves activities such as choosing an automation framework, writing scripts using a programming language, executing scripts automatically to generate test report etc.

With the increase in use of UML diagrams in capturing requirements, generating test cases from UML diagrams is gaining more attention, as it helps to be more systematic, and focused and facilitates automated testing [1]. Systematic testing ensures every combination of input is tried. Focused testing explores the information about where bugs likely to be found. Automated testing helps to produce and run the greatest number of consistent, repeatable and reusable tests [9].

Test automation involves many challenges in terms of test design, test generation and test execution. Some are listed below:

- Reusability of scripts
- Support for programming constructs (Ex: loop)
- Test design automation
- Test generation automation
- Test execution automation
- Support for verification points (Ex: Text, log)
- PC connectivity independence
- Handle menu tree changes

In this paper, a method is proposed to meet the above challenges. The objective of this method is to generate inexpensive, reusable, portable tests automatically for testing mobile devices, irrespective of changing UI menu items, pixel relocations, changing form factors. In this approach, UML Sequence diagram is used for capturing test scenarios. XML Metadata Interchange (XMI) [14] file obtained from Sequence diagram is parsed to derive Control Flow Graphs (CFG). From CFG, XML based test cases are arrived. Navigation through menu tree of mobile device is achieved by using menu tree database [15]. An APK is generated to handle this XML file and menu tree database and then execute test cases on target device.

This method is illustrated through detailed block diagrams. Algorithms used in this method are explained. Extensive experimentation is done at different stages such as test design, test generation and test execution, to cross check effectiveness of algorithms. A detailed case study is given as a proof of concept to systematically generate tests from Sequence diagram based scenarios. Also this method is compared with other methods available in literature.

II. Related Work

This section discusses various methods that have been proposed for test automation. Several test automation frameworks [2, 16] are available in literature.

In [3], D. Kundu et al. proposed a method to parse Sequence diagram based XMI file and then generate Control Flow Graph (CFG). Different Sequence diagram components such as Messages, Operands, Combined fragments, Guards are considered. A Java based application parses XMI file, recognizes components such as messages, operands and combined fragments, and then extracts Nodes, Edges, and Guards to generate CFG. A defined set of rules are applied, which are based on the program structures such as loop, alt, break etc. and then CFG is arrived.

Ruifeng Chen et al. [4] proposed a Selenium based automation framework. XML based test suites are generated, which comprise of test cases. Each test case consists of one or more test steps. Each test step is composed of Selenese and verification tags. Each Selenese includes command, locator and value. The command can be user actions such as typing text or clicking button. A Java based test script is generated automatically from XML test case. This script is automatically executed to generate HTML based test report.

Tuomas Pajumen et al. [5] proposed a keyword/action driven test automation framework. In this approach, model based testing tool called TEMA is integrated with Robot framework. Labeled State Transition System (LSTS) based models are created to represent test cases. These test cases comprises of high level key words. These high level keywords are mapped with low level keywords provided by Robot library. When test cases are executed Robot generates user events such as typing text, click button etc.

Tommi Takala et al. [6] proposed a keyword/action driven test automation framework for testing Android mobile. TEMA toolset is used to create LSTS models to capture scenarios. To generate low level events, tools such as Monkey and Hierarchy viewer are used. Monkey is used to simulate user input events. Hierarchy viewer is used to capture UI layout information.

In [7], P.Costa et al. proposed Pattern Based GUI Testing (PBGT) approach. Models are created using PARADIGM-DSL language, which supports User Interface Test Patterns (UITP) such as Call UITP, Input UITP etc to test web applications. Call UITP is used to check functionality of corresponding invocation of a link, which leads to a different web page. Input UITP is used to test input fields, when valid and invalid inputs are given. PARADIGM-ME helps to map web elements with UITPs. PARADIGM-TG generates test cases from model. PARADIGM-TE executes these test cases. To test mobile application, a Selendroid based driver is developed to generate low level user action events in mobile device.

In [8], D. Amalfitano et al. proposed a method to test Android application by considering both user events and context events. User events are generated by user such as click button, type text etc. Context events are generated by system such as GPS signal loss and recovery, Network instability, USB plugging etc. Three different techniques are adopted to consider context event patterns. First technique is manually injecting context event and then test the application. Second technique is mutating the existing test case with context event pattern and then test. Third is exploration technique, iteratively inject context event patterns and test behavior of application. These techniques are applied to JUnit test cases.

In [18], Domenico et al. proposed a GUI (Graphical User Interface) crawling-based framework to test Android mobiles. This framework works along with Robotium framework [20] for analyzing the components of a running Android application. Using this info, it generates firable events to crawl through various widgets and detects crashes.

In [19], Lulu et al. proposed a framework which involves a model based activity page. It is implemented through open source frameworks such as Robotium and Monkey runner [21]. A crawling algorithm navigates through components of activity page. It also generates scripts to run with Robotium for emulator and Monkey runner for device.

III. Architecture of proposed Framework

In this section, architecture and design constraints of proposed test automation framework are discussed.

A. *Overview of the proposed framework*

The proposed framework is based on UML Sequence diagram based testing. The major steps involved in this approach are illustrated through a block diagram as shown in Figure 1. In this approach, Sequence diagrams are created to capture input scenarios. XMI file obtained from this Sequence diagram is parsed to extract model information such as messages and their precedence relations [3]. Using this information, edges connecting nodes along with labels are found. A Control Flow Graph is derived from edges by applying a set of rules as explained in [3]. A recursive algorithm is developed to obtain basis path test cases from CFG.

A XML file is generated to represent these basis path test cases. An APK is generated, which can parse this XML file and fire commands in Android mobile device. These commands generate events in Android mobile such as click menu item, read UI text etc.

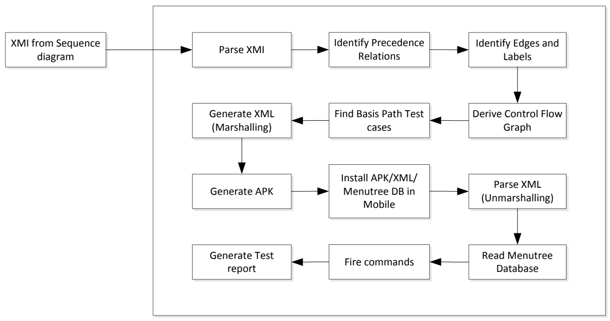

XMI from Sequence diagram
Parse XMI
Identify Precedence Relations
Identify Edges and Labels
Generate XML (Marshalling)
Find Basis Path Test cases
Derive Control Flow Graph
Generate APK
Install APK/XML/ Menutree DB in Mobile
Parse XML (Unmarshalling)
Generate Test report
Fire commands
Read Menutree Database

Fig. 1. Block diagram of proposed test automation framework

B. *Architecture of the proposed framework*

Figure 2 illustrates proposed framework to generate test script for Android mobile. The framework includes two major tools known as Virtual Test Engineer (VTE) and a menu tree generator. Menu tree generator [15] recursively navigates through menus of Android mobile and stores the path of each menu in a database. VTE is a Java based application, consists of a User Interface (UI) having controls and buttons to select input files. It has major modules such as XMI parser, Test case generator and APK generator. XMI parser is exactly same as mentioned in [3], generates CFG from Sequence diagram. Test case generator converts this CFG into basis path test cases in the form of XML file. APK generator takes this XML file and menu tree database file and then creates a new APK file, which can be installed in Android mobile. This APK invokes Android service, which in turn parses XML test cases and then generates events. These events are passed to an UI automator [13] based jar file, which is nothing but library of functions such as Click button, Click menu, Navigate, Wait, VerifyUIText etc. These functions perform Android button/menu clicks to simulate user actions, and then reading UI texts to verify expected results. Test scheduler [17] is an APK, which invokes generated APKs one by one sequentially. Each generated APK generates XL based test report automatically.

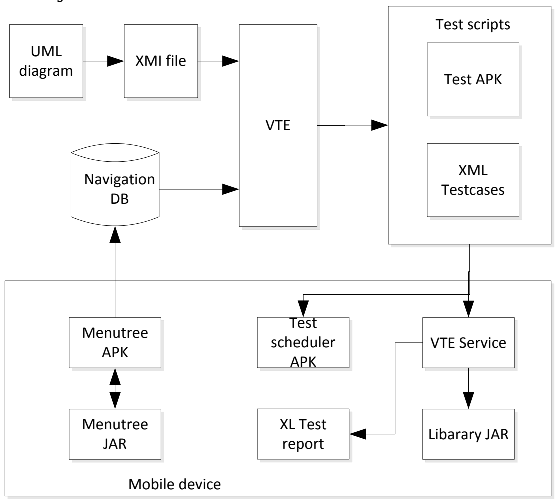


Fig. 2. Architecture of proposed framework

C. *Class diagram of proposed framework*

Class diagram of proposed framework is shown in Figure 3. Message class consists of method names and arguments. Method name is library function name such as Navigate, ClickButton etc. Arguments are either 0 or 1 or 2, depending upon the library function. Messages are inside operand or outside. Operands include guard conditions. Operands are contained in combined fragments. Nested combined fragments contain fragments inside a fragment. Sequence handler class extracts method name, arguments from Message class. Parse XMI class performs operations such as find edges, find CFG, derive basis path test cases etc. UI class provides interface to user to select input file, generate test cases and generate APKs etc.

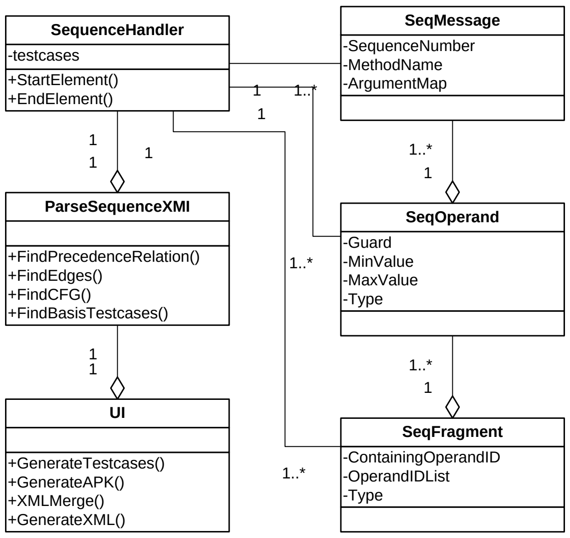


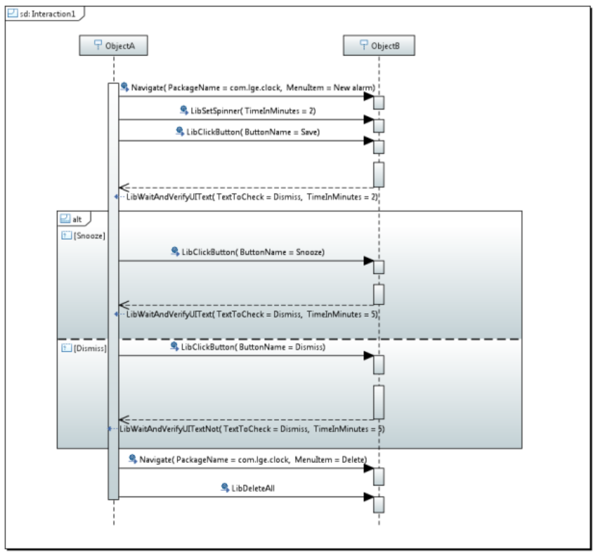
Fig. 3. Class diagram of proposed framework

IV. TEST DESDIGN AUTOMATION

In this section, algorithms involved in test design automation are discussed. UML Sequence diagram is used to capture test scenarios. A parser [3] is developed to parse XMI file obtained from this Sequence diagram. The method proposed in [3] is used for this purpose. From XMI file, Sequence diagram components such as messages, operands, and combined fragments are extracted. An algorithm is developed to find precedence relation from this information. Using precedence relations, Edges are found. A set of rules [3] are used to make CFG from Edges. A recursive algorithm is developed to derive basis path test cases from CFG. UML 2.0 supports different programming structures such as loop, alt, opt and break. A detailed experimentation is done to parse these programming structures and then generate basis path test cases.

1) *XMI parser*

An UML Sequence diagram as shown in Figure 4 is created using Papyrus tool [10]. This Sequence diagram is represented in the form of XMI notation [14] and saves as *.uml file. This XMI file is parsed using SAX parser and then different Sequence diagram components such as synchronous messages, asynchronous messages, reply messages, combined fragments, interaction operands and constraints are extracted. The sequence of messages determines sequence of test execution steps. Combined fragment includes different interaction operators such as alternative (alt), option (opt), break and loop.


sd: Interaction1
ObjectA
ObjectB
Navigate(PackageName = com.lge.clock, MenuItem = New alarm)
LibSetSpinner(TimeInMinutes = 2)
LibClickButton(ButtonName = Save)
LibWaitAndVerifyUIText(TextToCheck = Dismiss, TimeInMinutes = 2)
alt
[Snooze]
LibClickButton(ButtonName = Snooze)
LibWaitAndVerifyUIText(TextToCheck = Dismiss, TimeInMinutes = 5)
[Dismiss]
LibClickButton(ButtonName = Dismiss)
LibWaitAndVerifyUITextNot(TextToCheck = Dismiss, TimeInMinutes = 5)
Navigate(PackageName = com.lge.clock, MenuItem = Delete)
LibDeleteAll

a.

Fig. 4. Sequence diagram and corresponding CFG

While parsing combined fragments, attributes such as *Id*, *Type*, *ContainingOperandId and SeqOperandIDList* are obtained. Fragment *Id* is unique and used for recognizing the fragment. *Type* is having values such as alt, opt, break and loop. *ContainingOperandId* contains id of operand in which this fragment is present. *SeqOperandIDList* contains ids of all operands present in this fragment.

While parsing operands, attributes such as Id, *Guard*, *MinValue*, *MaxValue* and *SeqMessageIDList* are obtained. Operand *Id* is unique and used for recognizing the operands. *Guard* represents condition for corresponding programming constructs, for example if-else or loop. *MinValue*, *MaxValue* contains start and end values of a loop. *SeqMessageIDList* contains ids of all messages present in this operand.

While parsing messages, attributes such as *Id*, *MessageType*, *MethodName*, *ParameterArgumentMap* and *SeqNumber* are obtained. Message *Id* is unique and used for recognizing the messages. *MessageType* is having values such as synchCall, asynchCall and reply. *MethodName* is having name of the message. *ParameterArgumentMap* contains a map of argument names and their corresponding values. *SeqNumber* is having sequence number of the message.

Using these attributes, messages outside fragments, nested fragments, operands inside fragments, fragments inside operands, messages inside operands, message sequence are recognized. From this information, a graph (CFG) is obtained with messages and fragments as nodes, edges connecting these nodes, guards as labels for these edges.

2) *Precedence Relation*

Precedence relation [3] is the relationship between two consecutive entities and represented as '<'. The entity is either message or fragment. If m1<m2 is true, then m1 is preceding m2. The following properties are exhibited by entities:

a. Asymmetric => if m1 < m2 then m2 < m1 is not true
b. Non-transitive => if m1 < m2 < m3 then m1<m3 is not true
c. Non-reflexive => m1<m1 is not true

Null precedence relation is the relationship between Null interaction and message and represented as '^'. For example, if a single message is present in a fragment, then the relationship is defined as ^ < m < ^. Similarly the first message inside fragment and last message inside fragment also follow Null precedence relationship. The algorithm for finding precedence relation between entities is as shown in Algorithm 1.

Algorithm 1. Find Precedence Relation

```
ALGORITHM :: FindPrecedenceRelation(msglist){
Precedence list ← {}
For(i=0; i<Size of msglist; i++)
{
        first ← msglist[i];
        second ← msglist[i+1];
        if(second==end)
        {
          RETURN(Precedence list);
        }
        IF (first is not fragment && second is not fragment) THEN
        {
          Precedence list ← AddPrecendenceInList(First, Second);
        }
        ELSEIF ( first is not fragment && second is fragment) THEN
        {
          Precedence list ← AddPrecendenceInList(First, Second);
          newmsglist ← Get_Message_list(Second);
          Precedence list ← FindPrecedenceRelation(newmsglist);
        }
        ELSE IF( first is fragment && second is not fragment) THEN
        {
          newmsglist ← Get_Message_list(first);
          Precedence list ← FindPrecedenceRelation(newmsglist);
          Precedence list ← AddPrecendenceInList(First, Second);
        }
        ELSE IF( first is fragment && second is fragment) THEN
        {
          newmsglist ← Get_Message_list(first);
          Precedence list ← FindPrecedenceRelation(newmsglist);
          Precedence list ← AddPrecendenceInList(First, Second);
          newmsglist ← Get_Message_list(second);
          Precedence list ← FindPrecedenceRelation(newmsglist);

        }
        ENDIF
}
RETURN(Precedence list);
}

AddPrecendenceInList(First, Second)
{
        Precedence list ← {};
        IF (Both first and second are inside same operand||
                        Any one is not inside operand) THEN
        {
          Precedence list ← First < second;
        }
        ELSE IF (first and second are inside different operands) THEN
        {
          Precedence list ← First < ^;
                Precedence list ← ^ < second;
        }
        ENDIF
          RETURN(Precedence list);
}
```

The precedence relation between two entities (first, second) is found using this algorithm. Each entity is either message or fragment. Hence four different combinations such as (message, message), (message, fragment), (fragment, message) and (fragment, fragment) are dealt. If any one of the entity is fragment, then recursive call is made to find relation for messages inside the fragment. After executing this algorithm, Precedence relation (<) and Null precedence (^) are added to messages appropriately.

3) *Finding Edges*

Using precedence relations, edges are identified to connect 'from' node and 'to' node (first, second). The nodes are either message or fragment. Hence four different combinations such as (message, message), (message, fragment), (fragment, message) and (fragment, fragment) are considered to find edges. The presence of Null precedence (^) represents fragment with start or end node. The algorithm for finding edges is as shown in Algorithm 2.

Algorithm 2. Find Edges

```
ALGORITHM :: FindEdges(Precedence list){
Edge list ← {}
For each Precedence in Precedence list
{
        first ← Get first from Precedence
        second ← Get Second from Precedence
        IF (first is not fragment && second is not fragment) THEN
        {
          IF (first == "^") THEN
                fromNode = node of (containing fragment of second + "S");
                toNode = node of second;
          ELSE IF (second == "^") THEN
                fromNode = node of first;
                    toNode = node of (containing fragment of first + "E");
          ELSE
                fromNode = node of first;
                    toNode = node of second;
          ENDIF
          Edge list ← New edge joining fromNode and toNode;
        }
        ELSEIF ( first is not fragment && second is fragment) THEN
        {
          IF (first == "^") THEN
                fromNode = node of (containing fragment of second + "S");
                toNode = node of (second + "S");
          ELSE
                fromNode = node of first;
                    toNode = node of (second + "S");
          ENDIF
          Edge list ← New edge joining fromNode and toNode;

        }
        ELSE IF( first is fragment && second is not fragment) THEN
        {
          IF (second == "^") THEN
                fromNode = node of (first + "E");
                    toNode = node of (containing fragment of first + "E");
          ELSE
                fromNode = node of (first + "E");
                    toNode = node of second;
          ENDIF
          Edge list ← New edge joining fromNode and toNode;
        }
        ELSE IF( first is fragment && second is fragment) THEN
        {
          fromNode = node of (first + "E");
              toNode = node of (second + "S");
          Edge list ← New edge joining fromNode and toNode;
        }
        ENDIF
}
RETURN(Edge list);
}
```

After executing this algorithm, all edges connecting each node are found.

4) *Control Flow Graph*

A Control Flow Graph (CFG) is derived from Edges, by assigning guard conditions associated with operands of each fragment to Edges. In a relation (first, second), if 'first' is Null precedence, then label the edge same as guard condition associated with the operand of the fragment containing 'first'. A set of control flow rules [3] are applied to edges to construct CFG for programming constructs such as loop, alt, opt, break etc. The algorithm for finding CFG is as shown in Algorithm 3.

Algorithm 3. Find CFG

```
ALGORITHM :: FindCFG(Edge list){
FOR EACH outerEdge in Edge list
{
        firstouter ← Get fromNode from outerEdge;
        secondouter ← Get toNode from outerEdge;
        IF (firstouter contains "loop" and "S") THEN
        {
          FOR EACH innerEdge in Edge list
          {
                firstinner ← Get fromNode from innerEdge;
                    secondinner ← Get toNode from innerEdge;
                    IF (firstinner contains "loop" and "E" &&
                            outer and inner belongs to same loop)
                    {
                        Delete Edge from firstinner to secondinner;
                        Add Edge from firstinner to firstouter;
                        Add Edge from firstouter to secondinner with label = !guard;
                    }
                    ENDIF
          }
        }
        ELSEIF (firstouter contains "break" and "S") THEN
        {
          FOR EACH innerEdge in Edge list
          {
                firstinner ← Get fromNode from innerEdge;
                    secondinner ← Get toNode from innerEdge;
                    IF (firstinner contains "break" and "E" &&
                            outer and inner belongs to same break)
                    {
                        Delete Edge from firstinner to secondinner;
                        Add Edge from firstouter to secondinner with label = !guard;
                    }
                    ENDIF
          }
        }
        ELSE IF(firstouter contains "opt" and "S") THEN
        {
          FOR EACH innerEdge in Edge list
          {
                firstinner ← Get fromNode from innerEdge;
                    secondinner ← Get toNode from innerEdge;
                    IF (firstinner contains "opt" and "E" &&
                            outer and inner belongs to same opt)
                    {
                        Add Edge from firstouter to secondinner with label = !guard;
                    }
                    ENDIF
          }
        }
        ENDIF
}
RETURN(Edge list);
}
```

After executing this algorithm, CFG for all edges is obtained as shown in Figure 5. The Sequence diagram for Alarm scenario consists of an 'Alt' fragment. Two alternatives such as Dismiss and Snooze are represented by this diagram. The corresponding CFG is shown is Figure 5. GraphViz 'Dot' tool [12] is used to generate a visualization image for CFG. All nodes, edges and labels are represented in Dot language format. Using this information, Dot tool generated visual CFG.

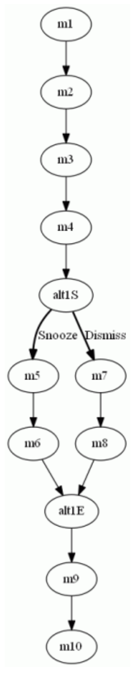


Fig. 5. CFG for Alarm Sequence diagram

5) *Basis Path Test cases*

A Basis Path (sometimes called independent path) through the program is any path from starting node to a terminal node that introduces *at least one new set of processing statements or a new condition* [2]. From CFG, Basis Path test cases are derived by identifying an independent path that includes an edge, which has not been traversed before path is defined. Test cases are designed such that all independent paths are executed at least once. The algorithm for finding Basis Path test cases is as shown in Algorithm 4.

Algorithm 4. Find Basis Path test cases

```
ALGORITHM :: FindBasisPaths (BasisPath list, firstEdge, lastEdge){
//Use global list variable 'Edge list'
fromNode of firstEdge ← Get fromNode from firstEdge;
toNode of firstEdge ← Get toNode from firstEdge;
BasisPath list ← Add firstEdge;
VisitedNodes ← Add fromNode of firstEdge and toNode of firstEdge;
edgePosition ← Position of firstEdge in Edge list;
size ← Size of Edge list;
FOR (i= edgePosition+1; i<size; i++)
{
        edge ← Get Edge list[i];
        fromNode ← Get fromNode from edge;
        toNode ← Get toNode from edge;
        IF (VisitedNodes not contains fromNode) THEN
        {
          VisitedNodes ← Add fromNode and toNode;
          BasisPath list ← Add edge;
          IF (edge == lastEdge) THEN
                  RETURN(BasisPath list);
          ENDIF
        }
        ELSEIF (VisitedNodes contains fromNode) THEN
        {
          //branch required
          branchPosition ← Get position of fromNode in BasisPath list;
          new BasisPath list ← Get BasisPath list till branchPosition;
          FindBasisPaths (new BasisPath list, edge, lastEdge);
        }
        ENDIF
}
RETURN(BasisPath list);
}
```

After executing this algorithm, Basis Path test cases are obtained as shown in Figure 6. The CFG for Alarm scenario consists of two alternative flows for Dismiss and Snooze as shown in Figure 5. The corresponding Basis Path test cases for Snooze and Dismiss scenarios are shown in Figure 6.

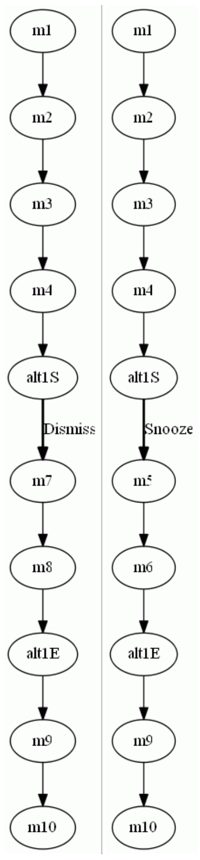


Fig. 6. Basis Path test cases

6) *Experimentation1*

A detailed experimentation is done to ensure Sequence diagram with different programming constructs such as loop, alt, opt and break are parsed properly and then converted to CFG, Basis Path test cases without error. The experimentation results are shown in Figure 7, 8 and 9. Simple If, If-Else, Loop and Break based Sequence diagrams are created and then corresponding CFG and test cases are generated as shown in Figure 7. Then two Ifs, If-Elses, Loops and Breaks are created and their corresponding CFG, test cases are generated as shown in Figure 8. A complex data structure containing all programming structures such as Alt, Opt, Loop and Break is created. Then their corresponding CFG and test cases are generated. The cyclomatic complexity of this complex structure is calculated as 9. Hence 9 basic path test cases are generated as shown in Figure 9.

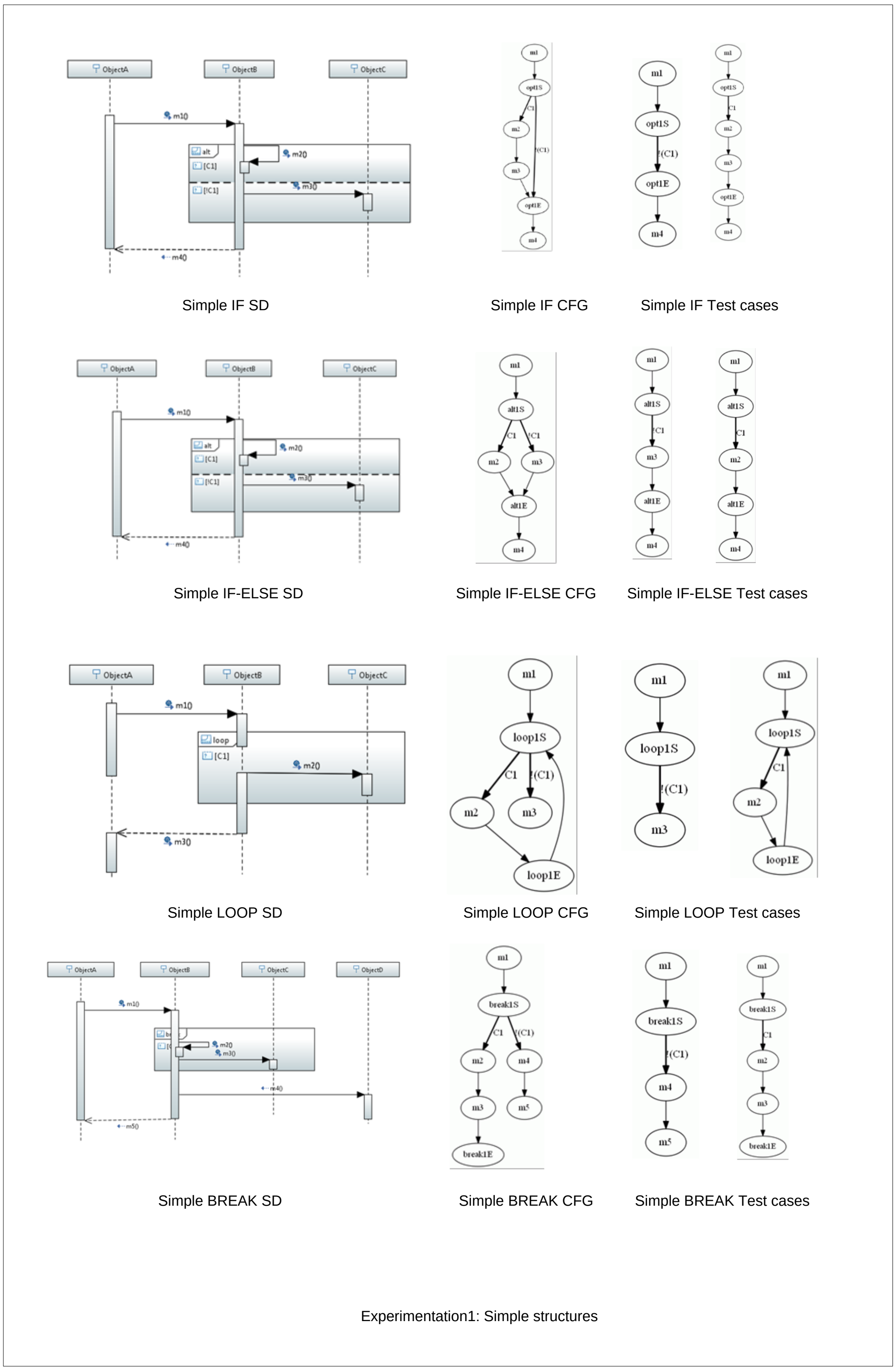


Fig. 7. Experimentation 1: Simple structures

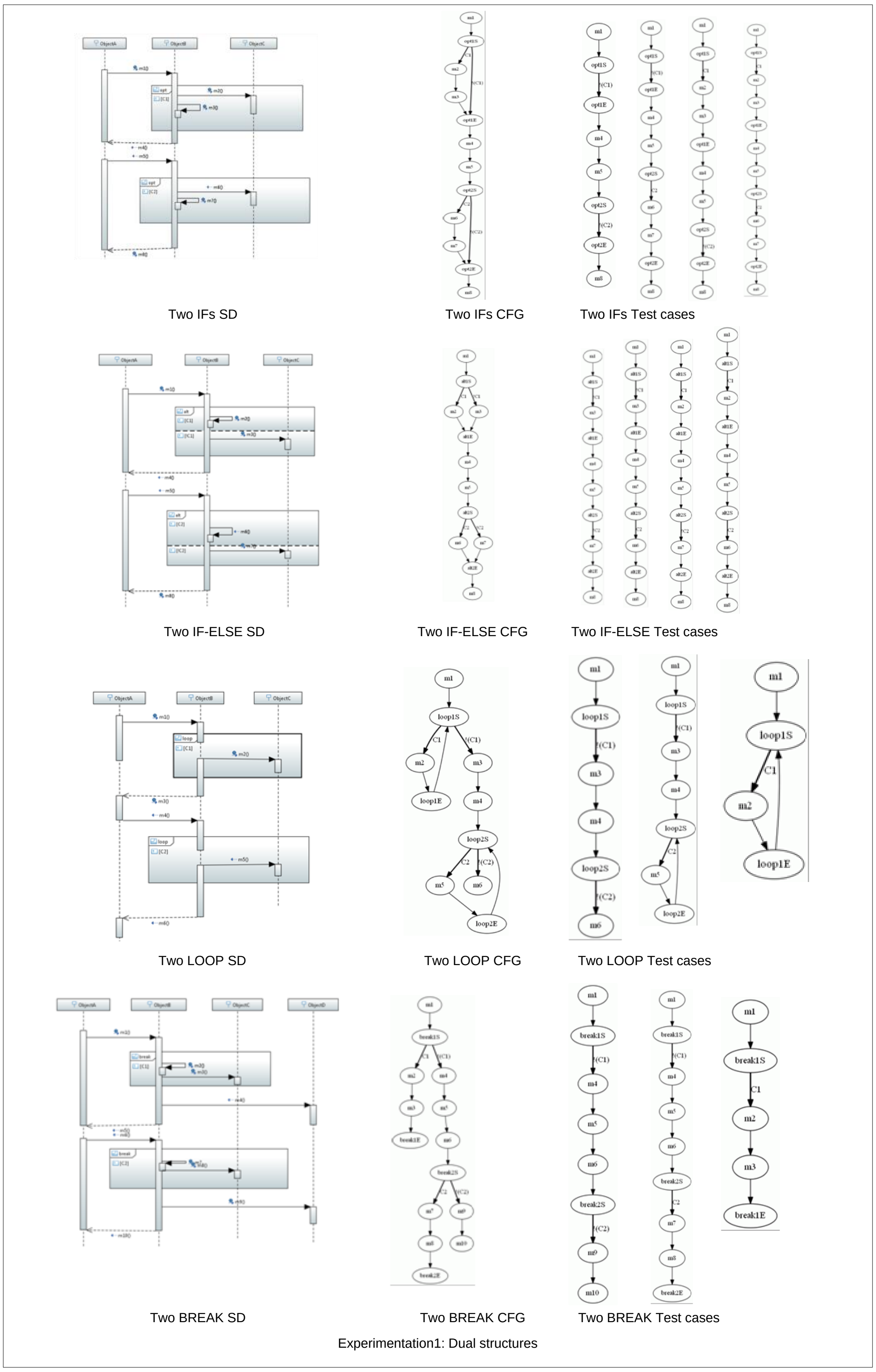
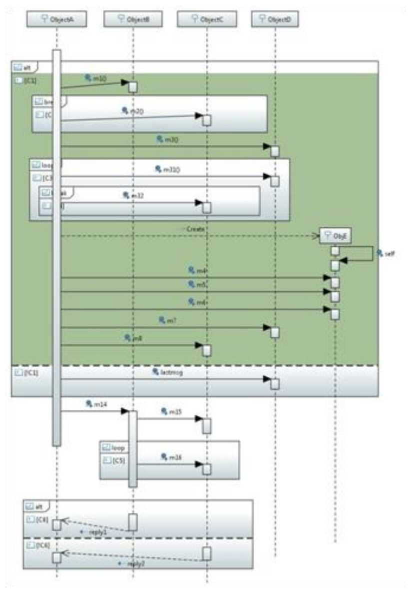
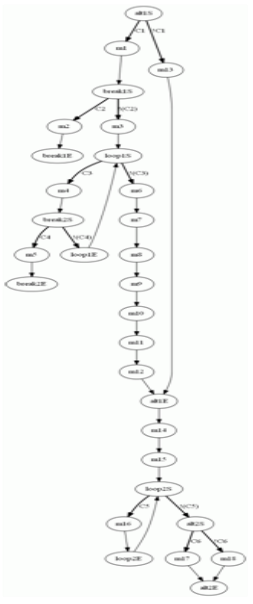


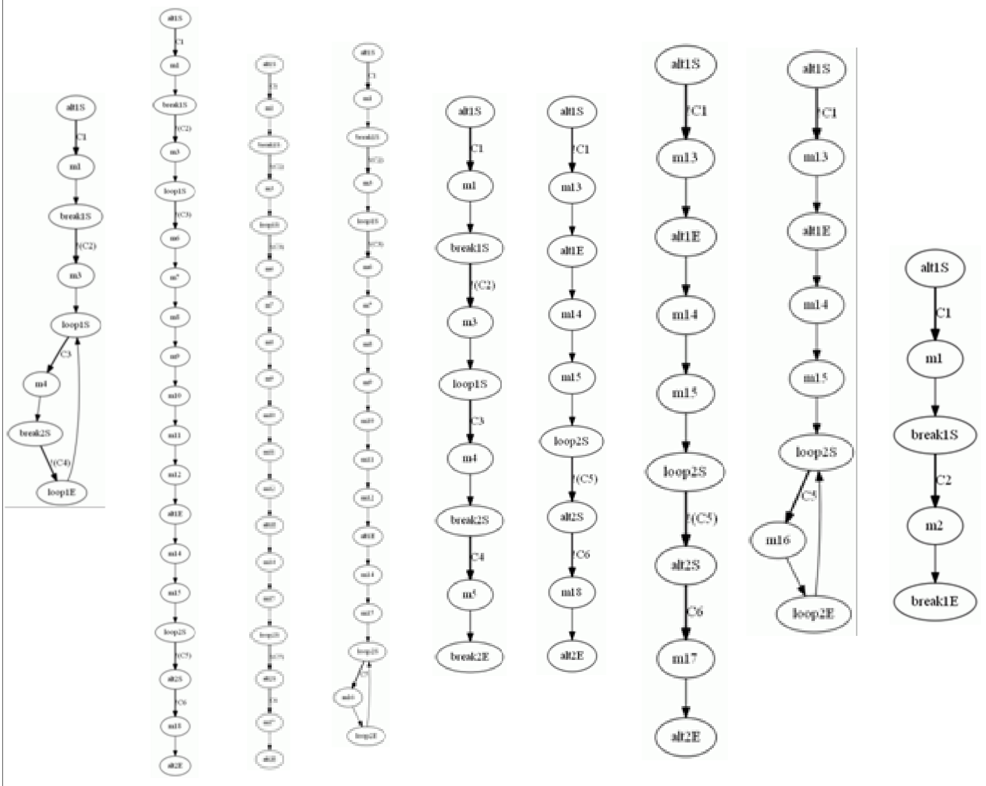
Fig. 8. Experimentation 1: Dual structures



Complex structure SD



Complex structure CFG



Complex structure Test cases

Experimentation1: Complex structures

Fig. 9. Experimentation 1: Complex structures

V. Test Generation Automation

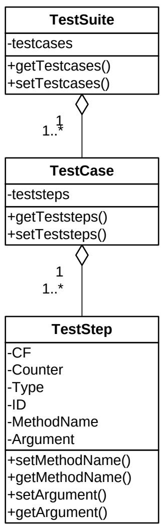
In this section, algorithms involved in test generation automation are discussed. Basis Path test cases are converted to 'executable' test cases by representing these test cases in XML format. Simple tool [11] is used to generate XML file from Basis Path test cases. Algorithm for generating XML file is explained. A menu tree generator tool [15] is developed to obtain menu tree of mobile device. This tool stores menu tree in Menu tree database, which is nothing but a Sqlite based database. Algorithm for generating Menu tree database is explained. An Android APK is generated in order to handle this XML file and Menu tree database to fire commands in Android device. Steps are explained to generate APK. A detailed experimentation is done to generate XML file for different programming constructs such as Alt and Loop, which is explained under Experimentation 2. Similarly, a detailed study is done for generating Menu tree database for different Android applications, which is explained under Experimentation 3.

1) XML Marshalling

Simple tool is used to generate XML file from TestSuite object. Simple tool provides APIs such as Serializer and Persister to manipulate XML file. The Serializer interface is used to represent objects that can serialize and deserialize objects to and from XML. This exposes read and write methods that can read from and write to various sources. The Persister object is used to provide an implementation of a Serializer. This implements the Serializer interface and enables objects to be persisted and loaded from various sources. Serialization is performed by passing TestSuite object and an XML stream into one of the write methods. The serialization process uses the class of the TestSuite object as the schema class. The object is traversed and all fields are marshalled to the result stream. The following line of code converts TestSuite schema class into XML file:

persister.write(testsuite, xmlfile);

The class diagram of TestSuite consists of three classes namely TestStep, TestCase and TestSuite as shown in Figure 10. TestSuite object is a list of one to many TestCase objects. Similarly TestCase object is a list of one to many TestStep objects. TestStep class consists of attributes such as CF, Counter, Type, ID, MethodName and Argument. ID is used to identify combined fragment and it is unique. CF represents combined fragment and is true, if the current message contained in a combined fragment. Counter value represents loop count, if the combined fragment type is loop. Type represents combined fragment type and contains value such as loop, alt, opt and break. MethodName is obtained from Node and represents the command to be fired. Argument is also obtained from Node and it is a map between argument name and its value.



b.

Fig. 10. Class diagram of TestSuite

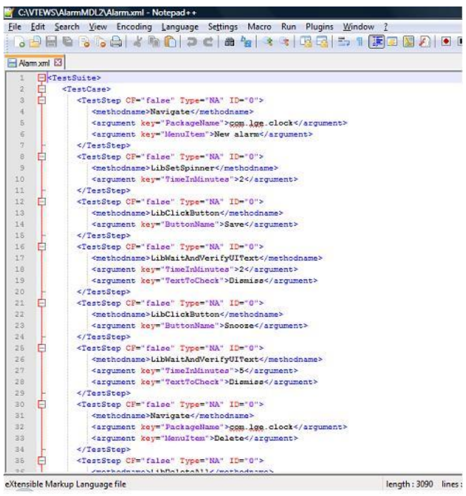
2) XML Generation Logic

Basis Path test case includes edges, nodes and labels. From each edge, from and to nodes are obtained. Each node contains method name and corresponding arguments. A TestStep object is created with these method name and arguments. Each test step is added to a list called test case. Each test case is added to a list called test suite. Persister API is used to convert this test suite to XML file. The algorithm for generating XML is as shown in Algorithm 5.

Algorithm 5. XML Generation

```
ALGORITHM :: XMLGeneration (BasisPath list){
CF ← false;
ID ← 0;
counter ← 0;
type ← NotApplicable;
testsuite ← {};
FOR EACH BasisPath in BasisPath list
{
        testcase ← {};
        VisitedNodes ← {};
        teststeps list ← {};
        FOR EACH BasisEdge in BasisPath
        {
          fromNode ← Get from node from BasisEdge;
                toNode ← Get to node from BasisEdge;
                label ← Get label from BasisEdge;
                IF (fromNode contains "loop" and "S") THEN
                {
                        CF ← true;
                        ID ← loop number;
                        counter ← loop count;
                        type ← "loop";
                }
                ELSE IF (fromNode contains "loop" and "E") THEN
                {
                    CF ← false;
                }
                ENDIF
                IF (VisitedNodes not contains fromNode) THEN
                {
                    VisitedNodes ← Add fromNode and toNode;
                    methodname ← Get method name from 'fromNode';
                    argument ← Get argument from 'fromNode';
                    teststep1 ← Add methodname, argument, CF, ID, type, counter;
                    methodname ← Get method name from 'toNode';
                    argument ← Get argument from 'toNode';
                    teststep2 ← Add methodname, argument, CF, ID, type, counter;
                    teststeps list ← Add teststep1 and teststep2
                }
                ENDIF
        }
        testcase ← Add teststeps list
        testsuite ← Add testcase
}
Convert testsuite into XML file using Persister;
}
```

Fragment start and end are checked to set CF flag. If type is 'loop', then loop count is considered. After executing this algorithm, XML file is generated for Alarm scenarios as shown in Figure 11.



```
<TestSuite>
    <TestCase>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>Navigate</methodname>
            <argument key="PackageName">com.lge.clock</argument>
            <argument key="MenuItem">New alarm</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibSetSpinner</methodname>
            <argument key="TimeInMinutes">2</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibClickButton</methodname>
            <argument key="ButtonName">Save</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibWaitAndVerifyUIText</methodname>
            <argument key="TimeInMinutes">2</argument>
            <argument key="TextToCheck">Dismiss</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibClickButton</methodname>
            <argument key="ButtonName">Snooze</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibWaitAndVerifyUIText</methodname>
            <argument key="TimeInMinutes">5</argument>
            <argument key="TextToCheck">Dismiss</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>Navigate</methodname>
            <argument key="PackageName">com.lge.clock</argument>
            <argument key="MenuItem">Delete</argument>
        </TestStep>
        <TestStep CF="false" Type="NA" ID="0">
            <methodname>LibDeleteAll</methodname>
```

C:\VTEWS\AlarmMDL2\Alarm.xml - Notepad++
File Edit Search View Encoding Language Settings Macro Run Plugins Window ?
Alarm.xml
eXtensible Markup Language file
length : 3090 lines :

Fig. 11. Generated XML file

3) *Menu tree Generator*

The major part of a test script is usually navigation through menu items. To automate test script generation, it is essential to automate navigation. To achieve this, a menu tree generation tool is developed to obtain hierarchical menu items from mobile device. This tool recursively navigates through menu items and stores the menu items encountered during navigation in a database called Menu tree database. Along with menu name, menu type and navigation path are also stored in Menu tree database. Menu type can be text view, button, hot menu etc. Navigation path is the path traced during navigation.

Figure 12 illustrates architecture of Android Menu tree Generator tool. It consists of an UI to select AUT (Application Under Test). This tool is nothing but an APK which is built upon basic Android building blocks such as Activity, Service etc. It also incorporates UI automator jar file. This jar file implements a recursive learning algorithm to extract UI properties such as text, image buttons, radio buttons, checkview etc. XL (Microsoft Excel file) based report is generated for menu tree representation. Sqlite database interface is used to store the path traced by this tool.

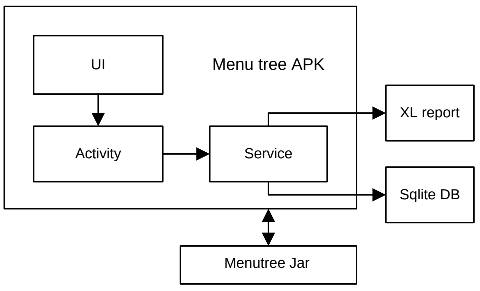


Fig. 12. Architecture of Menu tree Generator

4) *Menu tree Generator Logic*

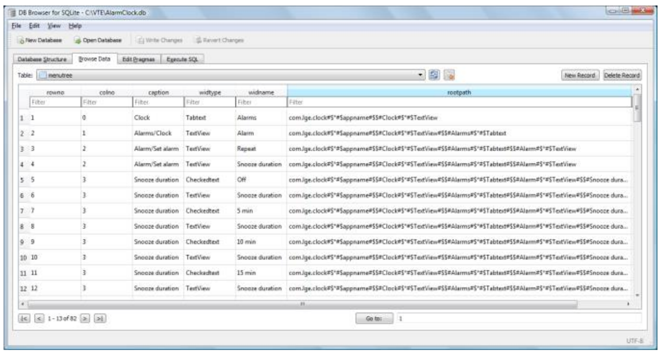
Menu tree APK identifies layouts such as linear layout, relative layout, frame layout extra. From these layouts it extracts UI objects such as text, buttons. When it clicks one text item, it checks whether new page or popup is opened. If it is new page, it recursively calls itself to do the learning. If the UI object is a popup or a button, it is handled in different manner. In this way all UI objects are learned and then each item is clicked based on the type of widget. The type of widget can be text, radio button, button with text, button with description, system event such as back key press, home key press. The algorithm for generating menu tree is as shown in Algorithm 6.

Algorithm 6. Menu tree Generation

```
ALGORITHM :: Menu TreeGeneration (){
I.      Extract widgets in current screen and add in stack
II.     Take one widget from stack
III.    Check stack is empty
IV.     IF empty, RETURN
V.      IF widget type = text, click and wait for new window
VI.     Store widget name, widget type, path from root into Sqlite DB
VII.    IF new window = new page, CALL 'MenuTreeGeneration'
VIII.   ELSE IF new window = popup, handle popup
IX.     GOTO II
}
```

5) *Menu tree Database*

Menu tree APK stores all UI menu items along with its path from root and type of widget in a Sqlite database, as shown in Figure 13.



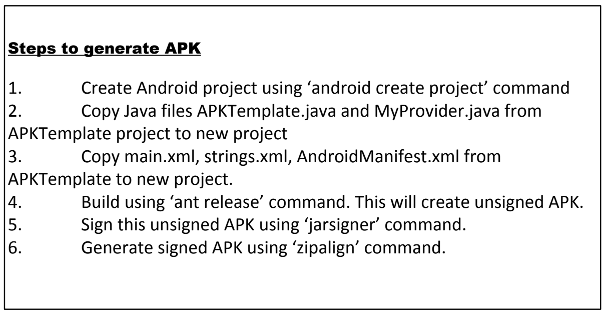
	rowno	colno	caption	widtype	widname	rootpath
	Filter	Filter	Filter	Filter	Filter	Filter
1	1	0	Clock	Tabtext	Alarms	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView
2	2	1	Alarms/Clock	TextView	Alarm	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext
3	3	2	Alarm/Set alarm	TextView	Repeat	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView
4	4	2	Alarm/Set alarm	TextView	Snooze duration	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView
5	5	3	Snooze duration	Checkedtext	Off	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
6	6	3	Snooze duration	TextView	Snooze duration	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
7	7	3	Snooze duration	Checkedtext	5 min	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
8	8	3	Snooze duration	TextView	Snooze duration	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
9	9	3	Snooze duration	Checkedtext	10 min	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
10	10	3	Snooze duration	TextView	Snooze duration	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
11	11	3	Snooze duration	Checkedtext	15 min	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...
12	12	3	Snooze duration	TextView	Snooze duration	com.lge.clock#$*#$appname#$$#Clock#$*#$TextView#$$#Alarms#$*#$Tabtext#$$#Alarm#$*#$TextView#$$#Snooze dura...

Fig. 13. Menu tree database

Row number and column number are useful to represent menu items in a tree format in XL. Caption is page title extracted from particular screen. The type of menu item is stored under column 'widtype'. This can be TextView, CheckedText, Button, CheckBox etc. The menu item is stored under 'widname' column. The path traced by the Menu tree generator for each menu item is stored under 'rootpath' column. To navigate to a particular menu item, this root path is obtained from database by searching menu item in the database. Using this path, navigation to menu item is achieved.

6) APK Generation

Android APK is generated in order to handle test case XML and Sqlite database. A template APK is used as input for generating APK. From this template APK, files such as activity.java, manifest.xml, string.xml are copied to new APK. These files are renamed and necessary modifications are done programmatically. After that, code is compiled and then build procedure is followed to make unsigned APK. Finally signing is done, so that APK can be installed in release binary. The steps to generate APK are shown in Figure 14.



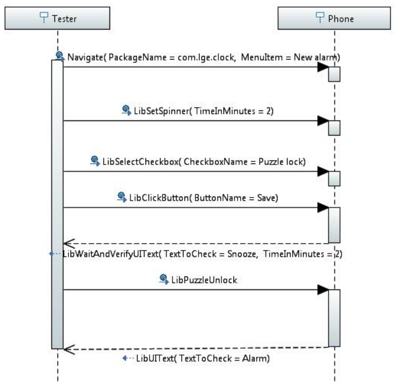
Steps to generate APK

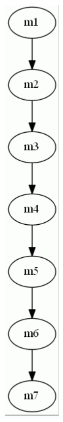
1. Create Android project using 'android create project' command
2. Copy Java files APKTemplate.java and MyProvider.java from APKTemplate project to new project
3. Copy main.xml, strings.xml, AndroidManifest.xml from APKTemplate to new project.
4. Build using 'ant release' command. This will create unsigned APK.
5. Sign this unsigned APK using 'jarsigner' command.
6. Generate signed APK using 'zipalign' command.

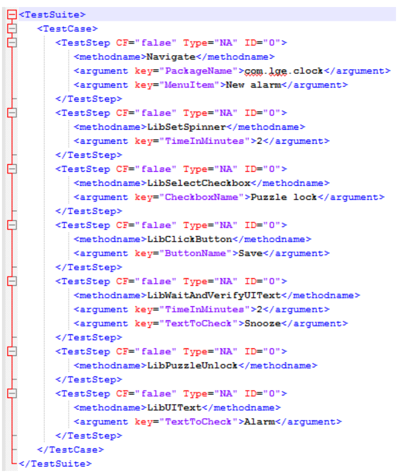
Fig. 14. Steps to generate APK

7) Experimentation2

A detailed experimentation is done to ensure XML file is generated from Sequence diagram with programming constructs such as linear, If-Else and Loop. The experimentation results are shown in Figure 15. Linear construct is simple to handle and generate XML. If-Else construct is slightly complex while generating Basis Path test cases, but generating XML is straight forward. Loop is most complex as both Basis Path test case generation and XML generation involve more complex logic. XML is not programming language. So implementing loop construct in XML is challenging. Getting loop count from Sequence diagram to XML is achieved through Simple tool. For every TestStep node, parameters such as CF, ID, Type and Count are added in order to distinguish loop and other constructs. If Type is found loop, then Count provides the loop count for which the TestSteps need to be repeated. If TestSteps are having attributes as CF is true and Type is loop, then they are considered us TestSteps within loop and repeated for loop count.

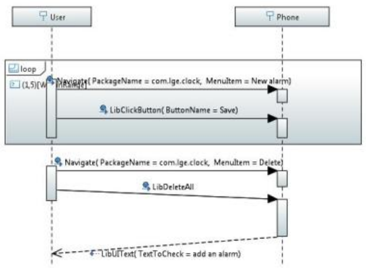

Tester
Phone
Navigate(PackageName = com.lge.clock, MenuItem = New alarm)
LibSetSpinner(TimeInMinutes = 2)
LibSelectCheckbox(CheckboxName = Puzzle lock)
LibClickButton(ButtonName = Save)
LibWaitAndVerifyUIText(TextToCheck = Snooze, TimeInMinutes = 2)
LibPuzzleUnlock
LibUIText(TextToCheck = Alarm)

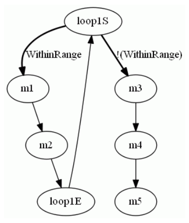


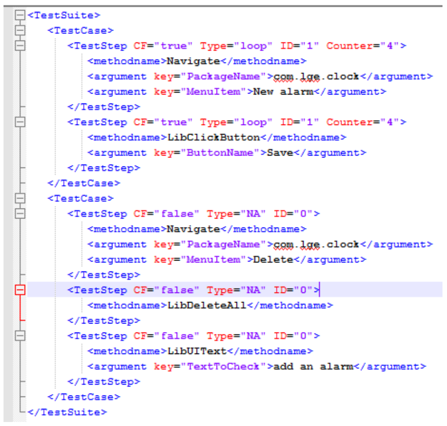


```
<TestSuite>
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.lge.clock</argument>
      <argument key="MenuItem">New alarm</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetSpinner</methodname>
      <argument key="TimeInMinutes">2</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSelectCheckbox</methodname>
      <argument key="CheckboxName">Puzzle lock</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Save</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWaitAndVerifyUIText</methodname>
      <argument key="TimeInMinutes">2</argument>
      <argument key="TextToCheck">Snooze</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibPuzzleUnlock</methodname>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibUIText</methodname>
      <argument key="TextToCheck">Alarm</argument>
    </TestStep>
  </TestCase>
</TestSuite>
```

Linear SD Linear CFG Linear XML

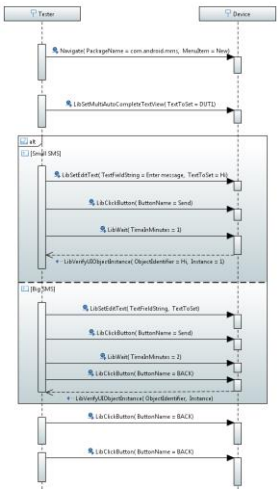

User
Phone
loop
Navigate(PackageName = com.lge.clock, MenuItem = New alarm)
LibClickButton(ButtonName = Save)
Navigate(PackageName = com.lge.clock, MenuItem = Delete)
LibDeleteAll
LibUIText(TextToCheck = add an alarm)

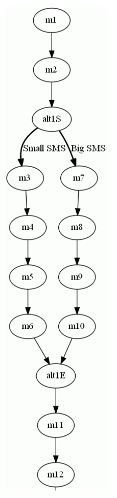


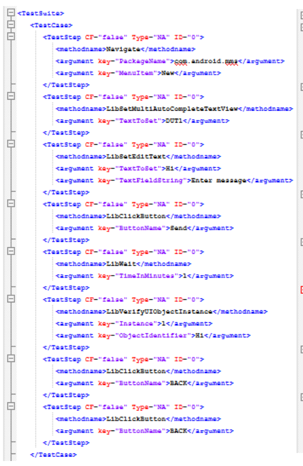


```
<TestSuite>
  <TestCase>
    <TestStep CF="true" Type="loop" ID="1" Counter="4">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.lge.clock</argument>
      <argument key="MenuItem">New alarm</argument>
    </TestStep>
    <TestStep CF="true" Type="loop" ID="1" Counter="4">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Save</argument>
    </TestStep>
  </TestCase>
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.lge.clock</argument>
      <argument key="MenuItem">Delete</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibDeleteAll</methodname>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibUIText</methodname>
      <argument key="TextToCheck">add an alarm</argument>
    </TestStep>
  </TestCase>
</TestSuite>
```

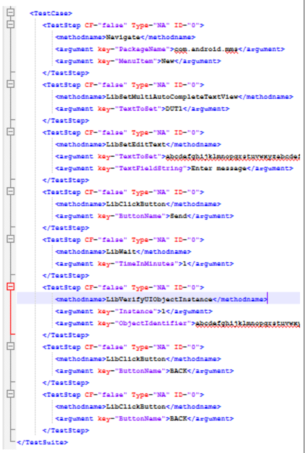
LOOP SD LOOP CFG LOOP XML


Tester
Device
Navigate(PackageName = com.android.mms, MenuItem = New)
LibSetMultiAutoCompleteTextView(TextToSet = DUT1)
alt
[Small SMS]
LibSetEditText(TextFieldString = Enter message, TextToSet = Hi)
LibClickButton(ButtonName = Send)
LibWait(TimeInMinutes = 1)
LibVerifyUIObjectInstance(ObjectIdentifier = Hi, Instance = 1)
[Big SMS]
LibSetEditText(TextFieldString, TextToSet)
LibClickButton(ButtonName = Send)
LibWait(TimeInMinutes = 2)
LibClickButton(ButtonName = BACK)
LibVerifyUIObjectInstance(ObjectIdentifier, Instance)
LibClickButton(ButtonName = BACK)
LibClickButton(ButtonName = BACK)





```
<TestSuite>
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.android.mms</argument>
      <argument key="MenuItem">New</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetMultiAutoCompleteTextView</methodname>
      <argument key="TextToSet">DUT1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetEditText</methodname>
      <argument key="TextToSet">Hi</argument>
      <argument key="TextFieldString">Enter message</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">1</argument>
      <argument key="ObjectIdentifier">Hi</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">BACK</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">BACK</argument>
    </TestStep>
  </TestCase>
```



```
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.android.mms</argument>
      <argument key="MenuItem">New</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetMultiAutoCompleteTextView</methodname>
      <argument key="TextToSet">DUT1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetEditText</methodname>
      <argument key="TextToSet">abcdefghijklmnopqrstuvwxyzabcdef
      <argument key="TextFieldString">Enter message</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">1</argument>
      <argument key="ObjectIdentifier">abcdefghijklmnopqrstuvwxy
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">BACK</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">BACK</argument>
    </TestStep>
</TestSuite>
```

If-Else SD If-Else CFG If-Else XML

Experimentation2: XML generated for different programming constructs

Fig. 15. Experimentation 2: XML generated for different programming constructs

8) Experimentation3

A detailed experimentation is done to ensure the algorithm is working for different applications such as Alarm, Messaging, Settings, Contacts, Gallery and Calculator. In the case of Messaging 791 menu items are extracted. In the case of settings 1133 menu items are extracted. This has to be elaborated to cover all applications. Table I shows execution time taken by menu tree generator for extracting menu items from different Android applications.

TABLE I. EXECUTION RESULT OF MENU TREE GENERATOR

Application name	**Execution parameters**	
	Learning time in minutes	***Number of menu items extracted***
Alarm	21	265
Messaging	76	791
Calculator	12	55
Settings	93	1133
Contacts	16	90
Gallery	18	126

VI. TEST EXECUTION AUTOMATION

In this section, algorithms involved in test execution automation are discussed. The generated APK invokes an Android service called 'VTE service'. Generated APK passes XML file name to VTE service. After parsing XML file, VTE service generates commands and sends to UI automator based 'Library.jar' file. Based on this command, a method is invoked by the jar file. This method is called as library function. A test scheduler is developed to handle multiple generated APKs. Test scheduler is nothing but an APK, invokes generated APKs one by one for executing test cases. A XL based test report is generated to store test results of each APK. A detailed experimentation is done to ensure generated test scripts are reusable across Android OS, form factors, UI versions, which is explained under Experimentation 4.

1) Mobile side architecture

Generated APK is installed in Android mobile along with VTE service, XML file, menu tree database file and Library jar file. After installation, the UI layout of generated APK appears in Android mobile as shown in Figure 16.

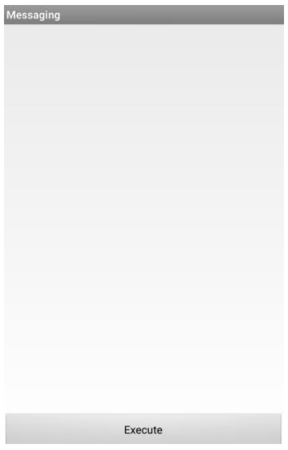


Fig. 16. UI layout of generated APK

Mobile side architecture is as shown in Figure 17. The generated APK passes file names of generated XML file and menu tree Sqlite db file to VTE service. The name and path information of XML test case and menu tree database are passed to service through Android bundle. VTE service parses XML file and extracts method name, arguments and other parameters using 'Unmarshalling' process. VTE Service generates commands such as Navigate, ClickButton, VerifyUIText etc. These commands are sent to UI automator based jar. This jar includes library functions such as Navigate(), ClickButton(), VerifyUIText() etc. For example, ClickButton library function contains UI automator based commands to simulate button click as given below:

```
        UiObject Button = new UiObject(new
UiSelector().text(ButtonName));
        if(Button.exists() && Button.isEnabled())
        {

Button.clickAndWaitForNewWindow(10000);
                sleep(1000);

                clickwidgetstatus=true;
        }
```

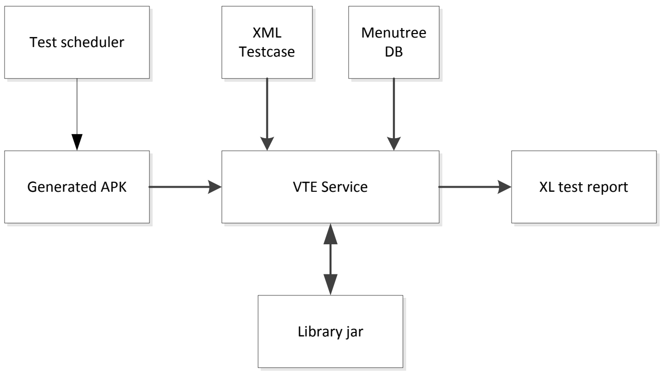


Fig. 17. Mobile side architecture

Result of test execution is saved in XL format automatically.

2) *XML Unmarshalling*

Simple tool is used to convert XML file to TestSuite object. Serializer is the API exposed by Simple tool to get Testsuite object from XML file. The XML file is read and it is unmarshalled to get TestSuite object. The following line of code converts XML file into TestSuite schema class:

TestSuite testsuiteinput = serializer.read(TestSuite.class, xmlinput);

From TestSuite object, TestCase object is obtained. From TestCase object, TestStep object is obtained. TestStep consists of method name and arguments. Using this information, a command is framed and then passed to library jar file. The algorithm for generating commands from XML file is as shown in Algorithm 7.

Algorithm 7. XML to Command generation

```
ALGORITHM :: XMLtoCommands (){
Convert XML into testsuite using Serializer;
testcases ← Get testcases from testsuite;
FOR EACH testcase in testcases
{
        command list ← {};
        teststeps ← get teststeps from testcase;
        FOR EACH teststep in teststeps
        {
          CF ← Get CF from teststep;
                IF (CF == true) THEN
                {
                            type ← Get type from teststep;
                        ID ← Get ID from teststep;
                        counter ←  Get counter from teststep;
                        IF (type == “loop”) THEN
                        {
                            command ←  Extract method name, arguments from teststep;
                            command list ← Add command;
                        }
                        ENDIF
                }
                ELSE IF (command list is not empty) THEN
                {
                    Execute command list;
                    Clear command list;
                }
                ELSE
                {
                    command ←  Extract method name, arguments from teststep;
                    Execute command;
                }
                ENDIF
        }
}
}
```

After executing this algorithm, a command is sent to library jar file for each TestStep. The following line of code shows the command:

proc = java.lang.Runtime.getRuntime().exec("uiautomator runtest Library.jar -c com.uia.example.my.Library" +

" -e command " + commandstr +

" -e argument1name " + argument1name +

" -e argument2name " + argument2name +

" -e argument1value " + argument1value +

" -e argument2value " + argument2value +

" -e database " + databasestr +

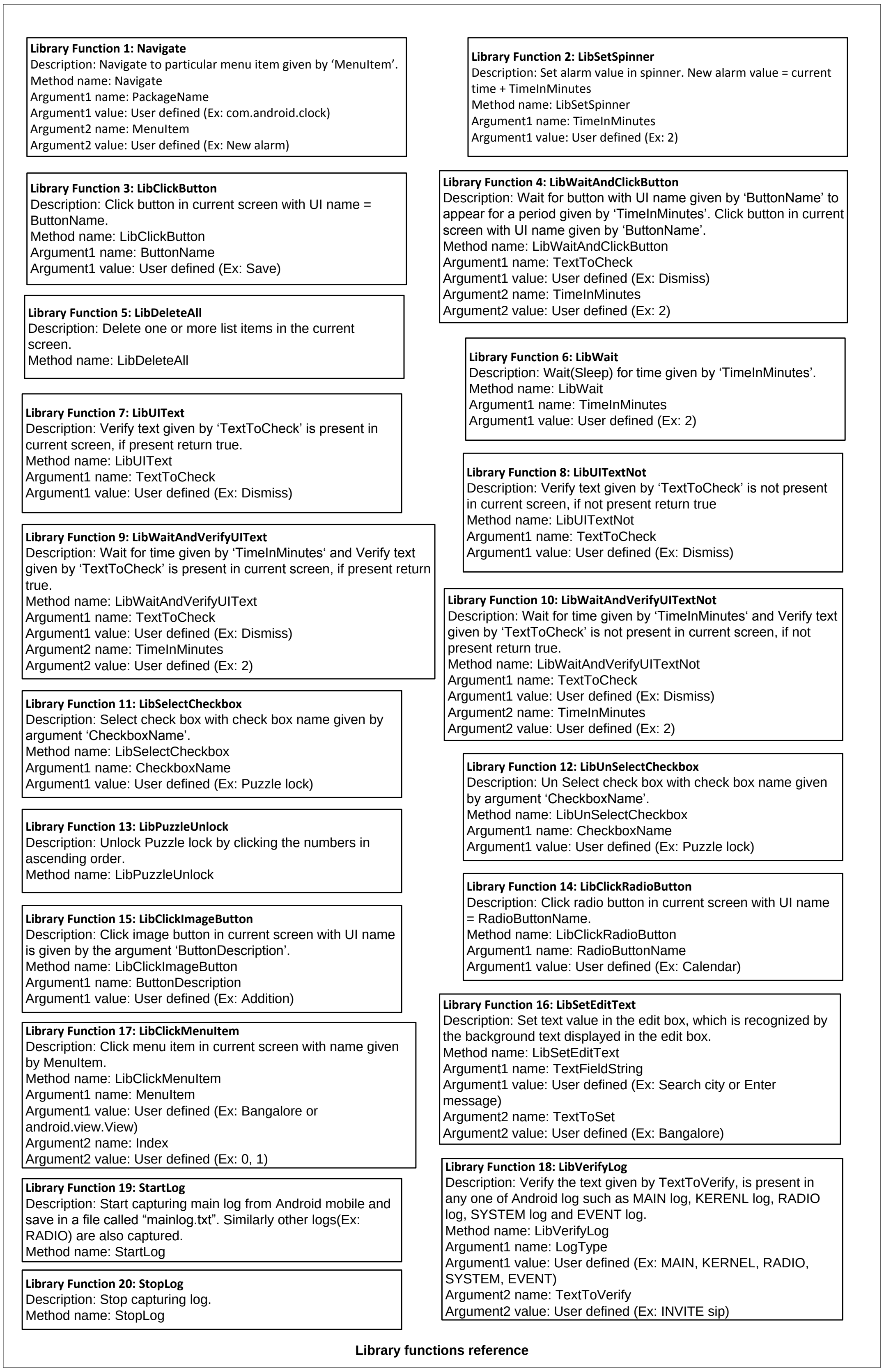
" -e logstatus " + logstatusstr

);

In case of loop, a list of commands are added together for TestSteps inside loop and then fired.

3) *Library functions*

Library functions are UI automator based wrapper functions to generate Android device events. When a command reaches from VTE service to library jar file, based on the command, any one of the library function is invoked. Navigate is one of the library function consists of two arguments such as package name and menu item. When this library function is invoked, the menu item is searched in the database corresponding to the package, and then the navigation path is retrieved. Using this path, navigation to this menu item is achieved, by generating repeated Android events such as click menu, click button etc. A set of library functions used in our approach are shown in Figure 18.



Library Function 1: Navigate
Description: Navigate to particular menu item given by ‘MenuItem’.
Method name: Navigate
Argument1 name: PackageName
Argument1 value: User defined (Ex: com.android.clock)
Argument2 name: MenuItem
Argument2 value: User defined (Ex: New alarm)

Library Function 2: LibSetSpinner
Description: Set alarm value in spinner. New alarm value = current time + TimeInMinutes
Method name: LibSetSpinner
Argument1 name: TimeInMinutes
Argument1 value: User defined (Ex: 2)

Library Function 3: LibClickButton
Description: Click button in current screen with UI name = ButtonName.
Method name: LibClickButton
Argument1 name: ButtonName
Argument1 value: User defined (Ex: Save)

Library Function 4: LibWaitAndClickButton
Description: Wait for button with UI name given by ‘ButtonName’ to appear for a period given by ‘TimeInMinutes’. Click button in current screen with UI name given by ‘ButtonName’.
Method name: LibWaitAndClickButton
Argument1 name: TextToCheck
Argument1 value: User defined (Ex: Dismiss)
Argument2 name: TimeInMinutes
Argument2 value: User defined (Ex: 2)

Library Function 5: LibDeleteAll
Description: Delete one or more list items in the current screen.
Method name: LibDeleteAll

Library Function 6: LibWait
Description: Wait(Sleep) for time given by ‘TimeInMinutes’.
Method name: LibWait
Argument1 name: TimeInMinutes
Argument1 value: User defined (Ex: 2)

Library Function 7: LibUIText
Description: Verify text given by ‘TextToCheck’ is present in current screen, if present return true.
Method name: LibUIText
Argument1 name: TextToCheck
Argument1 value: User defined (Ex: Dismiss)

Library Function 8: LibUITextNot
Description: Verify text given by ‘TextToCheck’ is not present in current screen, if not present return true
Method name: LibUITextNot
Argument1 name: TextToCheck
Argument1 value: User defined (Ex: Dismiss)

Library Function 9: LibWaitAndVerifyUIText
Description: Wait for time given by ‘TimeInMinutes‘ and Verify text given by ‘TextToCheck’ is present in current screen, if present return true.
Method name: LibWaitAndVerifyUIText
Argument1 name: TextToCheck
Argument1 value: User defined (Ex: Dismiss)
Argument2 name: TimeInMinutes
Argument2 value: User defined (Ex: 2)

Library Function 10: LibWaitAndVerifyUITextNot
Description: Wait for time given by ‘TimeInMinutes‘ and Verify text given by ‘TextToCheck’ is not present in current screen, if not present return true.
Method name: LibWaitAndVerifyUITextNot
Argument1 name: TextToCheck
Argument1 value: User defined (Ex: Dismiss)
Argument2 name: TimeInMinutes
Argument2 value: User defined (Ex: 2)

Library Function 11: LibSelectCheckbox
Description: Select check box with check box name given by argument ‘CheckboxName’.
Method name: LibSelectCheckbox
Argument1 name: CheckboxName
Argument1 value: User defined (Ex: Puzzle lock)

Library Function 12: LibUnSelectCheckbox
Description: Un Select check box with check box name given by argument ‘CheckboxName’.
Method name: LibUnSelectCheckbox
Argument1 name: CheckboxName
Argument1 value: User defined (Ex: Puzzle lock)

Library Function 13: LibPuzzleUnlock
Description: Unlock Puzzle lock by clicking the numbers in ascending order.
Method name: LibPuzzleUnlock

Library Function 14: LibClickRadioButton
Description: Click radio button in current screen with UI name = RadioButtonName.
Method name: LibClickRadioButton
Argument1 name: RadioButtonName
Argument1 value: User defined (Ex: Calendar)

Library Function 15: LibClickImageButton
Description: Click image button in current screen with UI name is given by the argument ‘ButtonDescription’.
Method name: LibClickImageButton
Argument1 name: ButtonDescription
Argument1 value: User defined (Ex: Addition)

Library Function 16: LibSetEditText
Description: Set text value in the edit box, which is recognized by the background text displayed in the edit box.
Method name: LibSetEditText
Argument1 name: TextFieldString
Argument1 value: User defined (Ex: Search city or Enter message)
Argument2 name: TextToSet
Argument2 value: User defined (Ex: Bangalore)

Library Function 17: LibClickMenuItem
Description: Click menu item in current screen with name given by MenuItem.
Method name: LibClickMenuItem
Argument1 name: MenuItem
Argument1 value: User defined (Ex: Bangalore or android.view.View)
Argument2 name: Index
Argument2 value: User defined (Ex: 0, 1)

Library Function 18: LibVerifyLog
Description: Verify the text given by TextToVerify, is present in any one of Android log such as MAIN log, KERENL log, RADIO log, SYSTEM log and EVENT log.
Method name: LibVerifyLog
Argument1 name: LogType
Argument1 value: User defined (Ex: MAIN, KERNEL, RADIO, SYSTEM, EVENT)
Argument2 name: TextToVerify
Argument2 value: User defined (Ex: INVITE sip)

Library Function 19: StartLog
Description: Start capturing main log from Android mobile and save in a file called “mainlog.txt”. Similarly other logs(Ex: RADIO) are also captured.
Method name: StartLog

Library Function 20: StopLog
Description: Stop capturing log.
Method name: StopLog

Library functions reference

Fig. 18. Library function reference

User actions such as clicking button, clicking menu item, select checkbox are simulated using library functions *LibClickButton*, *LibClickMenuItem*, *LibSelectCheckbox* respectively. Different verification methods like UI text verification, UI object verification, log verification are achieved using library functions *LibUIText*, *LibVerifyUIObjectClass*, *LibVerifyLog* respectively. The log capturing is enabled using *StartLog* and disabled using *StopLog* functions. Android supports five different logs such as *MAIN* log, *KERNEL* log, *RADIO* log, *SYSTEM* log and *EVENT* log.

4) *Test scheduler*

Test scheduler is required to invoke generated APKs one by one. It allows user to select all APKs or one or more APKs to execute by selecting corresponding check boxes.

Figure 19 illustrates architecture of test scheduler [17]. The test scheduler is nothing but an APK which is built upon basic Android building blocks such as Activity, Service, Content provider and Broadcast receiver. The States of a Testcase is monitored continuously by a Service, started from Activity of Scheduler. As some test cases are running for long time (more than 30minutes), the Service has to be declared as foreground Service. Also this Service has to be invoked periodically using an alarm manager timer, so that it listens to notifications coming from Content provider updates.

As per the Android life cycle definition, when Activity goes to back ground, its state will be changed to Pause and then Stop State. When the Activity goes to Resume State, the data related to Activity has to be retrieved, so that Activity can continue its functionality. State persistence within Activity is maintained using Shared preference. This is achieved through Shared preference. For example, user selects 2 complex test cases and then rotates screen from portrait to landscape, even after the same 2 test cases are selected.

Content observers are used by Scheduler to get notifications, whenever Content provider is updated by test cases. Content observer sets a flag, when one update in Content provider is notified. Scheduler after reading the update, resets this flag.

Broadcast receiver is the mechanism used to establish communication between Activity and Service of Scheduler. When user selects or de-selects one Testcase (APK) in UI, message is sent from Scheduler to Service using registered receivers. Critical information such as package names, checkbox status is sent as 'extras'. In Service, these extras are extracted from Intent to update Shared preference and Content provider.

Scheduler launches each Testcase using Intent. It updates Content provider, so that Service gets message through receiver.

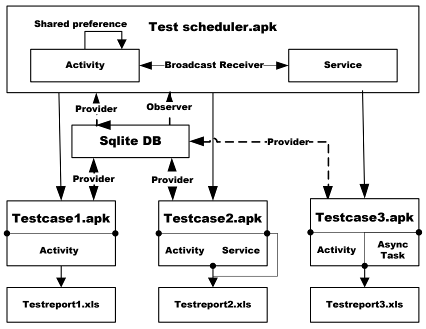


Fig. 19. Architecture of test scheduler

The test cases also APKs which can be invoked from scheduler in a predefined sequence. The scheduler monitors the execution of current test case. Once execution of current test case is completed, next test case is selected for running automatically. It provides checkbox options to choose test cases and then execute test cases using a button labeled as 'Execute'. Reset button resets the State of test cases to facilitate re-running the test cases. The UI layout of test scheduler is shown in Figure 20.

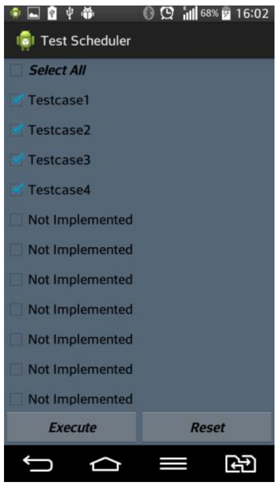


Fig. 20. UI layout of test scheduler

5) *Test Scheduler Logic*

There are 4 different States defined for each test case namely *NotExecuted*, *Started*, *Running* and *Executed* as shown in Figure 21. These States need to be shared between Scheduler and all test cases to ensure proper start of each test case one after other. This is achieved through content provider of Android framework, which is nothing but Sqlite database. Different operations such as *Insert*, *Update*, and *Query* are performed with this database. The initial State of Test case is 'Notexecuted'. When user clicks 'Execute' button, the State is updated to 'Started' and then Test case is invoked by Scheduler. The State is changed to 'Running' by Test case at the beginning of execution and then to 'Executed' at the end of the execution by Test case. The State is changed to 'NotExecuted' again, when user clicks 'Reset' button.

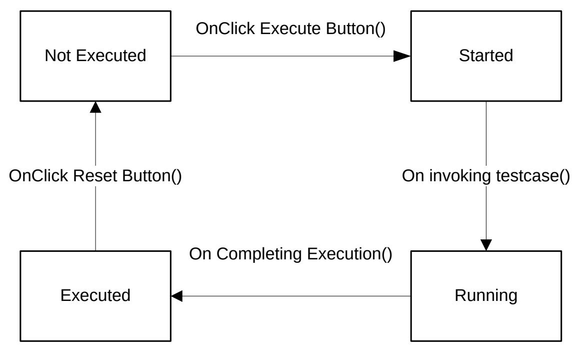


Fig. 21. States of a TestCase

The algorithm for test scheduler is as shown in Algorithm 8.

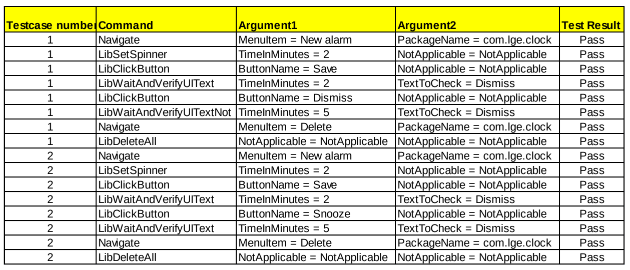
Algorithm 8. Test Scheduler

```
Algorithm for Test Scheduler

1. Check selected test case
2. Check State of the testcase
3. Wait till Execution button is pressed
4. If (Execution button = pressed &&
      Testcase=selected &&
      State=NotExecuted) Then
            Invoke test case
            State=Started
5. Otherwise goto next test case and repeat from step4
6. If(State=Started || State=Running)Then
            Wait for testcase execution to complete
7. If (State=Completed)Then
            Goto next test case and repeat from step4
8. If all test cases are executed, wait for user action.
9. If Reset button is pressed,
   set State of each test case to  'NotExecuted'
```

6) XL based test report

After test execution, for each test case APK, XL based test report is automatically generated, as shown in Figure 22.



Testcase number	Command	Argument1	Argument2	Test Result
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibSetSpinner	TimeInMinutes = 2	NotApplicable = NotApplicable	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
1	LibWaitAndVerifyUIText	TimeInMinutes = 2	TextToCheck = Dismiss	Pass
1	LibClickButton	ButtonName = Dismiss	NotApplicable = NotApplicable	Pass
1	LibWaitAndVerifyUITextNot	TimeInMinutes = 5	TextToCheck = Dismiss	Pass
1	Navigate	MenuItem = Delete	PackageName = com.lge.clock	Pass
1	LibDeleteAll	NotApplicable = NotApplicable	NotApplicable = NotApplicable	Pass
2	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
2	LibSetSpinner	TimeInMinutes = 2	NotApplicable = NotApplicable	Pass
2	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
2	LibWaitAndVerifyUIText	TimeInMinutes = 2	TextToCheck = Dismiss	Pass
2	LibClickButton	ButtonName = Snooze	NotApplicable = NotApplicable	Pass
2	LibWaitAndVerifyUIText	TimeInMinutes = 5	TextToCheck = Dismiss	Pass
2	Navigate	MenuItem = Delete	PackageName = com.lge.clock	Pass
2	LibDeleteAll	NotApplicable = NotApplicable	NotApplicable = NotApplicable	Pass

Fig. 22. XL based test report

The report contains *Testcase number*, *Command*, *Argument1*, *2* and *Test Result* columns. Library function name is added under *Command* column. Argument name and the corresponding values are added under *Argument 1 and 2* columns. The Pass/Fail result of each test step is stored under *Test Result* column.

7) Experimentation4

A detailed experimentation is done to ensure XML the generated APKs are seamlessly running under following different environmental conditions.

a. Different Android applications
b. Different Android OS versions (for example, Kitkat and Lollypop)
c. Devices with different form factors (for example, QVGA(240x320) and WVGA(480x800))
d. Different UI versions
e. Program structures coverage (for example, loop, alternative, option and break)
f. Structural coverage (for example, path coverage, node coverage, edge coverage, guard coverage)
g. Different verification methods (for example, UIText verification, log verification)
h. Test case type (for example, normal flow and alternative flow)

In this experiment, different applications such as Alarm, Messaging, Settings and Calculator are considered. Different coverage criteria as listed above are considered. Total 34 test cases are automated using this method. This has to be elaborated to cover all applications in future. Table II shows deployment data captured for different Android applications. The coverage achieved is showed using a √ symbol. The experimentation results are shown in Table II.

TABLE II. EXECUTION RESULT OF GENERATED TESTCASES

		Android OS		Form factor		UI version		Program structure				Test coverage			
Application name	No of testcases	Kitkat	Lollipop	QVGA	WVGA	4	4.1	Loop	Alt	Opt	Break	Path	Node	Edge	guard
Alarm	5	√	√	√	√	√	√	√	√	√	X	√	√	√	√
Stop watch	5	√	√	√	√	√	√	X	X	X	X	X	√	√	X
Timer	4	√	√	√	√	√	√	X	X	X	X	X	√	√	X
world clock	3	√	√	√	√	√	√	X	X	X	X	X	√	√	X
Settings	6	√	√	√	√	√	√	X	X	X	X	X	√	√	X
Calculator	4	√	√	√	√	√	√	X	X	X	X	X	√	√	X
Message	7	√	√	√	√	√	√	√	√	√	X	√	√	√	√

VII. Case Study

A detailed case study is done to apply all concepts practically to design, generate and execute test automatically. Initial study was done for generating menu tree from Android applications. Applications such as Alarm, Calculator, Messaging and Settings are taken for generating menu tree databases. Once databases are ready, UML based test design is done using Papyrus tool. Tests are generated in the form of XML and APKs. Library functions are incorporated in Library.jar file and then integrated with these APKs in order to simulate device level user events. Finally these APKs are installed in Android device and then tests are executed using a test scheduler. Test reports in the form XL are generated automatically. In this section, the application of this approach is explained with a couple of case studies over alarm and messaging test cases.

1) Loop test case

The first case study involves creating 5 alarms in a loop and then deletes all. A sequence diagram is drawn with following messages, as shown in Figure 23:

Start loop
1. Navigate to new alarm widget
2. Save alarm

End loop

3. Navigate to delete alarm widget
4. Delete All alarms
5. Verify ‘Add alarm’ widget is appearing

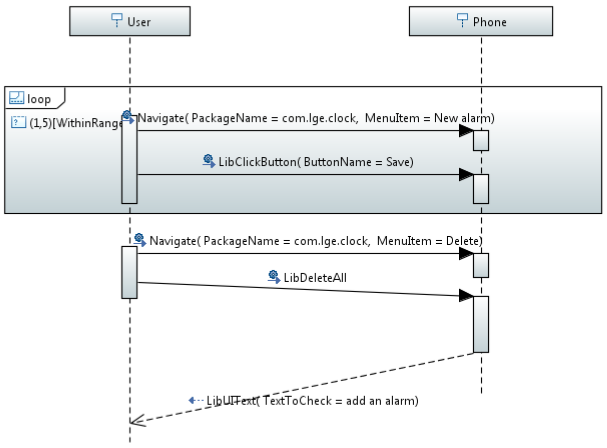

User
Phone
loop
(1,5)[WithinRange]
Navigate(PackageName = com.lge.clock, MenuItem = New alarm)
LibClickButton(ButtonName = Save)
Navigate(PackageName = com.lge.clock, MenuItem = Delete)
LibDeleteAll
LibUIText(TextToCheck = add an alarm)

Fig. 23. Five alarms Sequence diagram

A combined fragment construct ‘loop’ is added to include messages 1 and 2 to create a new alarm. The loop range with minimum and maximum values is set as 1 and 5 respectively, so that 5 new alarms can be created. VTE parses this diagram and generates two basis path test cases, one to cover messages inside loop and other one to cover messages outside the loop. VTE displays these test cases in graphical representation as shown in Figure 24.

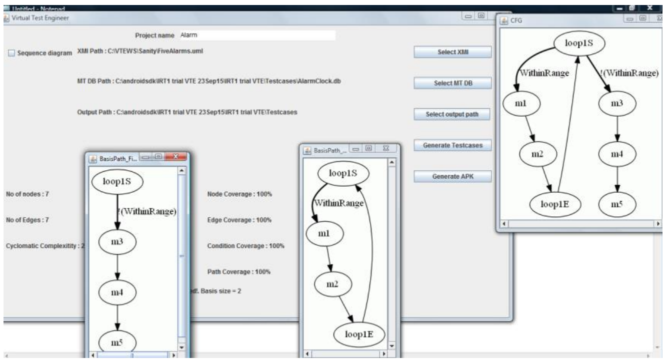
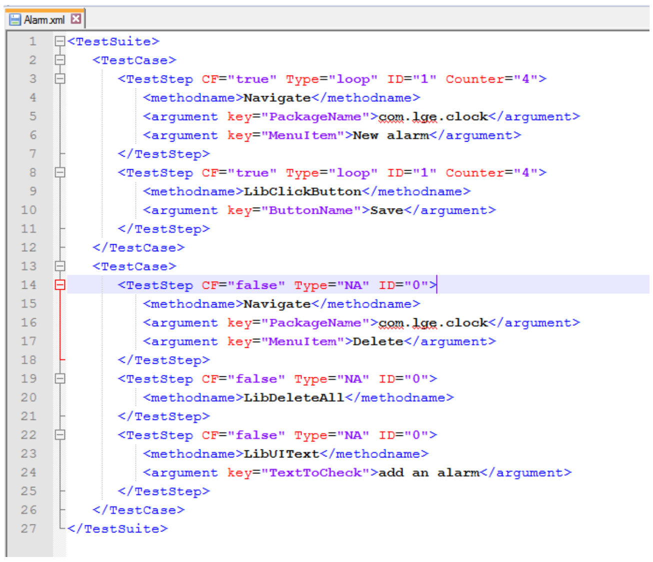

Project name Alarm
Sequence diagram XMI Path : C:\VTEWS\Sanity\FiveAlarms.uml
MT DB Path : C:\androidsdk\IRT1 trial VTE 23Sep15\IRT1 trial VTE\Testcases\AlarmClock.db
Output Path : C:\androidsdk\IRT1 trial VTE 23Sep15\IRT1 trial VTE\Testcases
No of nodes : 7
No of Edges : 7
Cyclomatic Complexitity : 2
Node Coverage : 100%
Edge Coverage : 100%
Condition Coverage : 100%
Path Coverage : 100%
d!. Basis size = 2

Fig. 24. Five alarms Basis Path test cases

On pressing ‘Generate Testcases’ button, a XML file is generated from Basis Path test cases as shown in Figure 25.



```
<TestSuite>
  <TestCase>
    <TestStep CF="true" Type="loop" ID="1" Counter="4">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.lge.clock</argument>
      <argument key="MenuItem">New alarm</argument>
    </TestStep>
    <TestStep CF="true" Type="loop" ID="1" Counter="4">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Save</argument>
    </TestStep>
  </TestCase>
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.lge.clock</argument>
      <argument key="MenuItem">Delete</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibDeleteAll</methodname>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibUIText</methodname>
      <argument key="TextToCheck">add an alarm</argument>
    </TestStep>
  </TestCase>
</TestSuite>
```

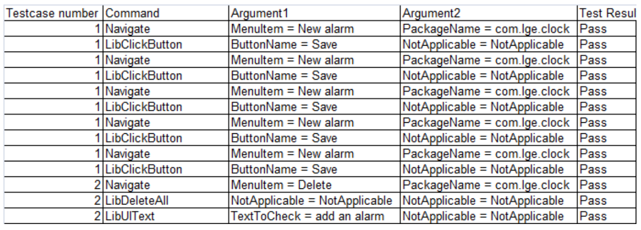
Fig. 25. XML generated for Five alarms scenario

XML file has a top level tag called Testsuite, which includes two test cases. The first test case includes two test steps with method names such as ‘Navigate’ and ‘LibClickButton’. Navigate method includes two arguments such as PackageName and MenuItem. LibClcikButton method includes one argument ButtonName. The attribute CF in first test case is true. It indicates this is a combined fragment. The Counter value greater than 0, indicates it is a loop. The second test case is scenario when loop condition is not met. In this case, it deletes all five alarms added earlier.

Menu tree DB for Alarm application has to be included using ‘Select MT DB’ button in VTE. In this case, ‘AlarmClock.db’ is included before APK is generated. When ‘Generate APK’ button is pressed in VTE, an APK known as ‘Alarm.apk’, is generated to handle this XML file. This APK sends XML name, Menu tree DB name to VTE service. VTE service parses XML test case and triggers the following command to invoke library function in ‘Library.jar’ file:

proc = java.lang.Runtime.getRuntime().exec("uiautomator runtest Library.jar -c com.uia.example.my.Library" + " -e command " + commandstr + " -e argument1name " + argument1name + " -e argument2name " + argument2name + " -e argument1value " + argument1value + " -e argument2value " + argument2value + " -e database " + databasestr);

Library function in Library.jar file generates events in android mobile to simulate user actions such as clicking menu items, buttons etc in order to execute these test cases. When Alarm.apk is executed, test cases are executed sequentially and an Excel based test report is generated automatically as shown in Figure 26.



Testcase number	Command	Argument1	Argument2	Test Resul
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
1	Navigate	MenuItem = New alarm	PackageName = com.lge.clock	Pass
1	LibClickButton	ButtonName = Save	NotApplicable = NotApplicable	Pass
2	Navigate	MenuItem = Delete	PackageName = com.lge.clock	Pass
2	LibDeleteAll	NotApplicable = NotApplicable	NotApplicable = NotApplicable	Pass
2	LibUIText	TextToCheck = add an alarm	NotApplicable = NotApplicable	Pass

Fig. 26. Test report for Five alarm test case

The test steps inside loop are executed repeatedly for count equal to loop count. In this case test steps such as Navigate and LibClickButton are executed for five times.

1) IF-ELSE test case

The second case study involves forwarding and replying a message designed as If-Else condition. A sequence diagram is drawn with following messages, as shown in Figure 27:

1. Navigate to new message
2. Type receiver mobile number (Self mobile number)
3. Type message ‘Hi’
4. Click ‘Send’ button
5. Wait for 1 minute
6. Verify message is received

If (condition=Forward)

7. Click Hot menu
8. Click menu item ‘Forward’
9. Select message to be forwarded (Hi)
10. Type receiver mobile number (Self mobile number)
11. Type message ‘Hi’
12. Click ‘Send’ button
13. Wait for 1 minute
14. Verify message is received

Else If (condition=Reply)

15. Type message ‘Hello’
16. Click ‘Send’ button
17. Wait for 1 minute
18. Verify message is received

End If

19. Click Hot menu
20. Delete All messages
21. Verify ‘New’ widget is appearing

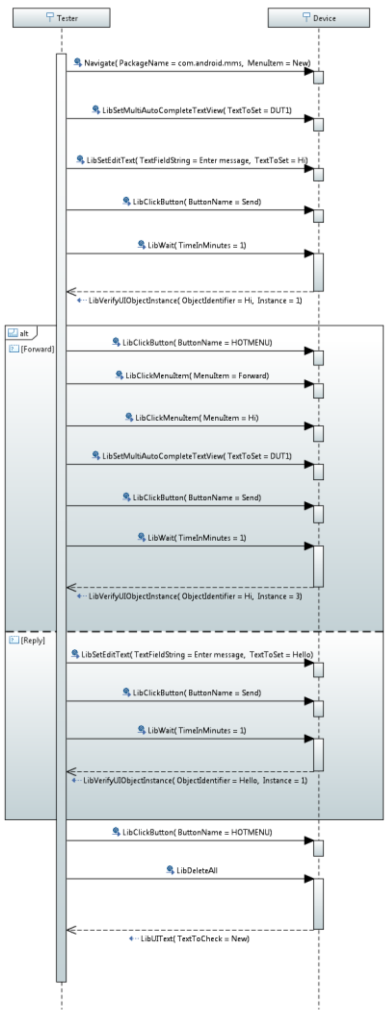
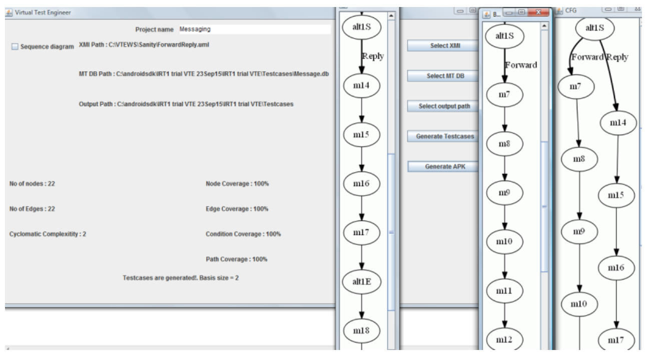

Tester
Device
Navigate(PackageName = com.android.mms, MenuItem = New)
LibSetMultiAutoCompleteTextView(TextToSet = DUT1)
LibSetEditText(TextFieldString = Enter message, TextToSet = Hi)
LibClickButton(ButtonName = Send)
LibWait(TimeInMinutes = 1)
LibVerifyUIObjectInstance(ObjectIdentifier = Hi, Instance = 1)
alt
[Forward]
LibClickButton(ButtonName = HOTMENU)
LibClickMenuItem(MenuItem = Forward)
LibClickMenuItem(MenuItem = Hi)
LibSetMultiAutoCompleteTextView(TextToSet = DUT1)
LibClickButton(ButtonName = Send)
LibWait(TimeInMinutes = 1)
LibVerifyUIObjectInstance(ObjectIdentifier = Hi, Instance = 3)
[Reply]
LibSetEditText(TextFieldString = Enter message, TextToSet = Hello)
LibClickButton(ButtonName = Send)
LibWait(TimeInMinutes = 1)
LibVerifyUIObjectInstance(ObjectIdentifier = Hello, Instance = 1)
LibClickButton(ButtonName = HOTMENU)
LibDeleteAll
LibUIText(TextToCheck = New)

Fig. 27. Forward-Reply Sequence diagram

A combined fragment construct ‘alt’ is added to include messages 7 to 14 under ‘If’ construct. Messages 15 to 18 are under ‘Else’ construct. The messages under ‘If’ part, are meant for ‘Forwarding’ a SMS. The messages under ‘Else’ part, are meant for ‘Replying’ a SMS. The messages outside combined fragment are added for both forward and reply test cases. VTE parses this diagram and generates two basis path test cases, one for forwarding and another for replying a SMS. VTE displays these test cases in graphical representation as shown in Figure 28.


Virtual Test Engineer

Project name Messaging

Sequence diagram XMI Path : C:\VTEWS\Sanity\ForwardReply.uml

MT DB Path : C:\androidsdk\IRT1 trial VTE 23Sep15\IRT1 trial VTE\Testcases\Message.db

Output Path : C:\androidsdk\IRT1 trial VTE 23Sep15\IRT1 trial VTE\Testcases

No of nodes : 22

No of Edges : 22

Cyclomatic Complexitity : 2

Node Coverage : 100%

Edge Coverage : 100%

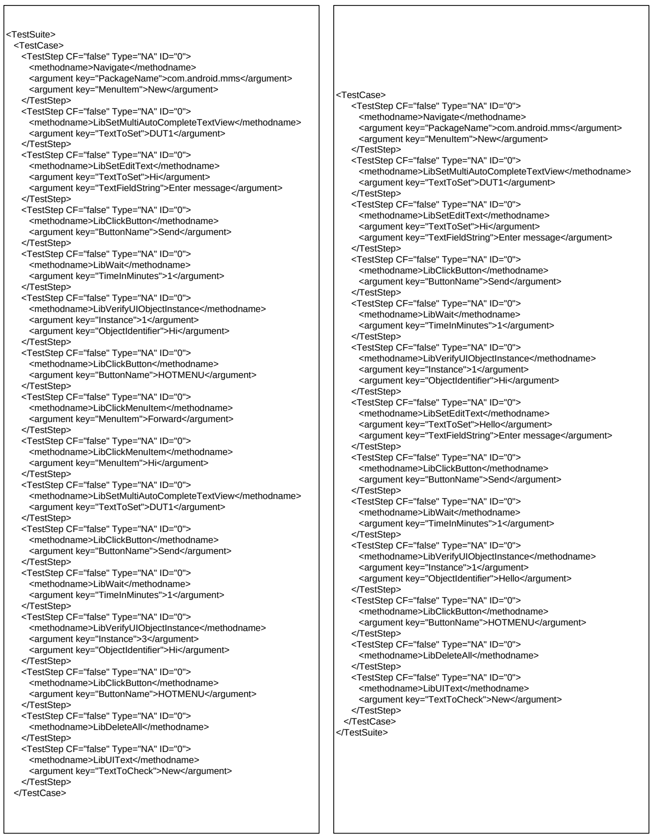
Condition Coverage : 100%

Path Coverage : 100%

Testcases are generated!. Basis size = 2

Fig. 28. Forward-Reply Basis Path test cases

On pressing 'Generate Testcases' button, a XML file is generated from Basis Path test cases as shown in Figure 29.


```
<TestSuite>
  <TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.android.mms</argument>
      <argument key="MenuItem">New</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetMultiAutoCompleteTextView</methodname>
      <argument key="TextToSet">DUT1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetEditText</methodname>
      <argument key="TextToSet">Hi</argument>
      <argument key="TextFieldString">Enter message</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">1</argument>
      <argument key="ObjectIdentifier">Hi</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">HOTMENU</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickMenuItem</methodname>
      <argument key="MenuItem">Forward</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickMenuItem</methodname>
      <argument key="MenuItem">Hi</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetMultiAutoCompleteTextView</methodname>
      <argument key="TextToSet">DUT1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">3</argument>
      <argument key="ObjectIdentifier">Hi</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">HOTMENU</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibDeleteAll</methodname>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibUIText</methodname>
      <argument key="TextToCheck">New</argument>
    </TestStep>
  </TestCase>
```

```
<TestCase>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>Navigate</methodname>
      <argument key="PackageName">com.android.mms</argument>
      <argument key="MenuItem">New</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetMultiAutoCompleteTextView</methodname>
      <argument key="TextToSet">DUT1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetEditText</methodname>
      <argument key="TextToSet">Hi</argument>
      <argument key="TextFieldString">Enter message</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">1</argument>
      <argument key="ObjectIdentifier">Hi</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibSetEditText</methodname>
      <argument key="TextToSet">Hello</argument>
      <argument key="TextFieldString">Enter message</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">Send</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibWait</methodname>
      <argument key="TimeInMinutes">1</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibVerifyUIObjectInstance</methodname>
      <argument key="Instance">1</argument>
      <argument key="ObjectIdentifier">Hello</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibClickButton</methodname>
      <argument key="ButtonName">HOTMENU</argument>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibDeleteAll</methodname>
    </TestStep>
    <TestStep CF="false" Type="NA" ID="0">
      <methodname>LibUIText</methodname>
      <argument key="TextToCheck">New</argument>
    </TestStep>
  </TestCase>
</TestSuite>
```

Fig. 29. XML generated for Forward-Reply test cases

XML file has a top level tag called Testsuite, which includes two test cases. The first test case includes test steps with method names such as 'LibClickButton', 'LibClickMenuItem' and 'LibSetMultiAutoCompleteTextView'. LibClcikButton is used to click buttons such as Hot menu button, Send button. LibClickMenuItem is used to click menu items such as Forward and Hi. LibSetMultiAutoCompleteTextView is used to type receiver mobile number under 'To' field. This method reads mobile number of DUT from 'Preconditions.db' file. User has to set DUT mobile number in 'Preconditions.db' before starting test execution, by using an APK known as 'Precondition.apk'. The second test case is reply scenario. At the end of each case, all messages are deleted.

Menu tree DB for Messaging application has to be included using 'Select MT DB' button in VTE. In this case, 'Message.db' is included before APK is generated. When 'Generate APK' button is pressed in VTE, an APK, in this case 'Messaging.apk', is generated to handle this XML file. This APK sends XML name, Menu tree DB name to VTE service. VTE service parses XML test case and triggers the commands to invoke library functions in 'Library.jar' file.

Library function in Library.jar file generates events in android mobile to simulate user actions such as clicking menu items, buttons etc in order to execute these test cases. When Messaging.apk is executed, test cases are executed sequentially and an Excel based test report is generated automatically as shown in Figure 30.

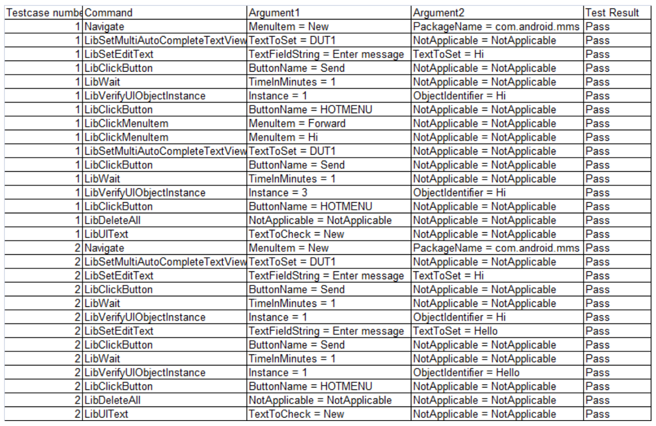

Testcase numbe	Command	Argument1	Argument2	Test Result
1	Navigate	MenuItem = New	PackageName = com.android.mms	Pass
1	LibSetMultiAutoCompleteTextView	TextToSet = DUT1	NotApplicable = NotApplicable	Pass
1	LibSetEditText	TextFieldString = Enter message	TextToSet = Hi	Pass
1	LibClickButton	ButtonName = Send	NotApplicable = NotApplicable	Pass
1	LibWait	TimeInMinutes = 1	NotApplicable = NotApplicable	Pass
1	LibVerifyUIObjectInstance	Instance = 1	ObjectIdentifier = Hi	Pass
1	LibClickButton	ButtonName = HOTMENU	NotApplicable = NotApplicable	Pass
1	LibClickMenuItem	MenuItem = Forward	NotApplicable = NotApplicable	Pass
1	LibClickMenuItem	MenuItem = Hi	NotApplicable = NotApplicable	Pass
1	LibSetMultiAutoCompleteTextView	TextToSet = DUT1	NotApplicable = NotApplicable	Pass
1	LibClickButton	ButtonName = Send	NotApplicable = NotApplicable	Pass
1	LibWait	TimeInMinutes = 1	NotApplicable = NotApplicable	Pass
1	LibVerifyUIObjectInstance	Instance = 3	ObjectIdentifier = Hi	Pass
1	LibClickButton	ButtonName = HOTMENU	NotApplicable = NotApplicable	Pass
1	LibDeleteAll	NotApplicable = NotApplicable	NotApplicable = NotApplicable	Pass
1	LibUIText	TextToCheck = New	NotApplicable = NotApplicable	Pass
2	Navigate	MenuItem = New	PackageName = com.android.mms	Pass
2	LibSetMultiAutoCompleteTextView	TextToSet = DUT1	NotApplicable = NotApplicable	Pass
2	LibSetEditText	TextFieldString = Enter message	TextToSet = Hi	Pass
2	LibClickButton	ButtonName = Send	NotApplicable = NotApplicable	Pass
2	LibWait	TimeInMinutes = 1	NotApplicable = NotApplicable	Pass
2	LibVerifyUIObjectInstance	Instance = 1	ObjectIdentifier = Hi	Pass
2	LibSetEditText	TextFieldString = Enter message	TextToSet = Hello	Pass
2	LibClickButton	ButtonName = Send	NotApplicable = NotApplicable	Pass
2	LibWait	TimeInMinutes = 1	NotApplicable = NotApplicable	Pass
2	LibVerifyUIObjectInstance	Instance = 1	ObjectIdentifier = Hello	Pass
2	LibClickButton	ButtonName = HOTMENU	NotApplicable = NotApplicable	Pass
2	LibDeleteAll	NotApplicable = NotApplicable	NotApplicable = NotApplicable	Pass
2	LibUIText	TextToCheck = New	NotApplicable = NotApplicable	Pass

Fig. 30. Test report for Forward-Reply test cases

The test steps for 'Forward' test case, inside IF part, are executed one time and listed under test case number1. The test steps for 'Reply' test case, inside ELSE part, are executed one time and listed under test case number2. The test steps outside 'alt' combined fragments are executed in both test cases.

VIII. COMPARISON WITH OTHER METHODS

In [3], a method to parse XMI exported from Sequence diagram is illustrated. A Control Flow Graph (CFG) is constructed from nodes, edges and guards. This method is not extended to generate test cases and test scripts. In our approach, XML based basis path test cases are generated from CFG. Also APK based test scripts are generated to simulate user actions in Android phones.

In [4], XML based test cases are created manually and then Java based test scripts are generated automatically. As XML is not supporting programming constructs such as loop, option, alternative and break, generating reusable test cases is challenging. In our approach, XML based test cases are generated automatically from Sequence diagram. As UML2 based Sequence diagram supports programming constructs, reusable test cases can be created.

In [5], State diagram based model is created to represent test scenario, and then low level events are generated using Robot framework. State diagram can be either action machine or refinement machine. Action machine contains high level keywords called action words. Refinement machine is implementation of the action machine, which contains low level keywords such as type word, press key etc. In our approach, Sequence diagram is used to capture user scenarios. XMI exported from the Sequence diagram is parsed to extract key words and then XML file is generated with a test case schema, which includes these keywords as part of test steps. Low level events are generated in the mobile device by parsing this XML file.

In [6], Hierarchy viewer is used to capture UI layout information. To preserve security, Hierarchy Viewer can only connect to devices running a developer version of the Android system. This framework can be used for emulator or device with debug binary. In our approach, UI automator is used for simulating low level device events such as touch events and UI content verification. UI automator enables testing of Android mobile with both debug binary and release binary.

In [7], Pattern Based GUI Testing (PBGT) approach relies on mapping UI elements with User Interface Test Patterns (UITP). Model is built on UITPs to represent test cases. Whenever UI layout is changing, corresponding model also needs to be changed. Test coverage is reduced as some UI elements are not recognized by Selendroid. In our approach, menu tree data base is generated for each model. This enables re-usability of same test suite across devices with different form factors.

In [8], context event patterns are considered along with user event patterns to test Android application. This is white box approach, mainly useful for debugging Android application. Context events are recognized either by listeners through their handlers or by using notification of corresponding intent messages. In our approach, a service is running in background, parsing XML to extract events, inject events through UI automator based library functions. This is black box approach and no need to change Android application code.

In [18], a GUI crawling algorithm needs instrumenting the source code of the application under test, in order to detect runtime crashes. In our approach, instrumentation is not required. It is fully black box approach. In [18], the algorithm is based on ‘Robotium’ framework. ‘Robotium’ is useful for only debug binaries. It needs extra permission to perform click from one application to another. In our approach, UI automator tool is used. UI automator tool is provided by Android, capable of extracting UI objects during run time and also it can generate click events. This helps to use this algorithm not only for debug binary, release binary too. In [18], the proof of concept is done with small size application. But in our case, the size of the application does not matter. It can handle complex scenarios such as tabs, popups, and different widgets such as radio button, checktextview etc.

In [19], the low level Android event is generated using Monkey runner. To generate touch event, coordinates of widgets are used. Using coordinates, is not fool proof. When widgets are dislocated in the page, the coordinate based click results in wrong action. In our approach, UI object properties such as name of the widget, description of the button, class name, and resource id are used. Even though the widget is dislocated in the current screen, the click action happens without fail.

The comparison between different test automation frameworks is shown in Table III.

TABLE III. COMPARISON OF TEST AUTOMATION FRAMEWORKS

Attributes	VTE	Selenium	TEMA	PBGT	Extended Ripper	Autoval	A^2T^2	APBF
Reference	This paper	[4]	[5,6]	[7]	[8]	[16]	[18]	[19]
Model based testing(MBT)	√	X	√	√	X	X	X	X
Programming constructs(If else, loop, etc)	√	X	√	X	√	√	X	X
Test design automation (Basis path etc)	√	X	√	X	X	X	X	X
Test case generation automation	√	X	√	√	√	X	√	√
Test script generation automation	√	√	√	√	√	X	√	√
Handling UI changes	√	X	√	√	X	X	√	X
PC connectivity Independence	√	X	X	X	X	X	X	X
Test Execution Automation	√	√	√	√	√	√	√	√
Image comparison	X	X	X	X	X	√	√	√
Text comparison	√	√	√	√	√	X	√	X
Log verification	√	X	X	X	X	X	X	X
UI Object based verification	√	√	√	√	√	X	√	X
Coordinate based automation	X	X	X	√	X	√	X	√
Same script for Form factors	√	√	√	√	√	X	√	X
Portability (maintenance for OS/Navigation changes)	√	X	X	X	X	X	X	X
Scalability (Extension Ex: Keyword/Data driven)	√	X	√	X	X	X	X	X
Code instrumentation	X	X	X	X	X	X	√	X

Test automation frameworks are compared with respect to various attributes to assess the capability and reusability of frameworks. The capability attributes are support of programming constructs, handling UI changes, test generation automation, text/log/UI object verification, portability and scalability. The reusability attributes are Model based testing (MBT) support, same script for different form factors, menu tree generation to handle navigation changes.

IX. CONCLUSIONS

In this paper, an automated method to generate functional tests from Sequence diagram is proposed for testing Android mobiles. Testing activities such as requirement analysis, test case design, test case generation, test script generation and test execution are automated. The scenarios are captured in the form of Sequence diagram. XMI file exported from Sequence diagram is parsed to get CFG. Basis path test cases are generated from CFG in the form of XML file. A menu tree database is generated, which helps to navigate through menu tree of Android mobile. An APK is generated to handle this XML and menu tree database to simulate user interface events such as click menu item, click button etc in target device. XL based test reports are generated. To execute multiple APKs in sequence, a test scheduler is developed. The objective is to reduce the test effort by automating test engineering activities throughout the test cycle.

In this method, a Sequence diagram is created to capture test scenarios. In future, Activity diagram and State diagram based models will be involved to capture test scenarios.

In this method, algorithms to generate CFG and basis path test cases are developed. In future, algorithms to consider other test design techniques such as data flow testing, orthogonal optimization, LCSAJ testing etc will be developed.

In future, a suitable algorithm will be developed using genetic algorithm and AI (Artificial Intelligence) planning, to generate test data.

This framework is a kind of keyword/action driven framework and scalable. In future, it will be extended with data driven and design pattern frameworks.

This method will be more generalized, so that this method will be useful not only for Android platform, other embedded systems with different platforms also will get benefited.

REFERENCES

[1] R. V. Binder, Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.

[2] Anirban Basu, Software Quality Assurance, Testing and Metrics, PHI Learning, 2015.

[3] D. Kundu, D. Samanta, and R. Mall "An Approach to Convert XMI Representation of UML 2.x Interaction Diagram into Control Flow Graph", in International Scholarly Research Network(ISRN) Software Engineering, Volume 2012.

[4] Ruifeng Chen and Huaikou Miao "A Selenium based Approach to Automatic Test Script Generation for Refactoring JavaScript Code", in IEEE/ACIS 12th International Conference on Computer and Information Science (ICIS), 2013.

[5] Tuomas Pajumen, Tommi Takala and Mika Katara. “Model-Based Testing a General Purpose Keyword-Driven Test Automation Framework”, International Conference on Software Testing, Verification and Validation Workshops, 2011.

[6] Tommi Takala, Mika Katara, and Julian Harty, “Experiences of system-level model-based GUI testing of an Android application,” in Proceedings of the 4th IEEE International Conference on Software Testing, Verification, and Validation (ICST 2011). Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2011, pp. 377–386..

[7] P.Costa, A.C.R. Paiva, and M. Nabuco, "Pattern Based GUI testing for Mobile Applications", In Proc. of 9th International Conference on the Quality of Information and Communications Technology (QUATIC), IEEE Computer Society, 2014.

[8] D. Amalfitano, A. R. Fasolino, P. Tramontana, N. Amatucci, "Considering Context Events in Event-Based Testing of Mobile Applications", IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2013.

[9] M. Fewster and D. Graham, Software Test Automation: Effective use of test execution tools. Addison–Wesley, 1999.

[10] https://eclipse.org/papyrus/.

[11] http://simple.sourceforge.net/.

[12] http://www.graphviz.org/Documentation/dotguide.pdf.

[13] Android Developers. UI automator. Available at: http://developer.android.com/tools/help/uiautomator/index.html. Last accessed Nov. 29, 2014.

[14] OMG, “XML Metadata Interchange (XMI),v2.1”,2004.

[15] Anbunathan R and Anirban Basu. "A Recursive Crawler Algorithm to Detect Crash in Android Application", IEEE International Conference on Computational Intelligence and Computing Research(ICCIC), 2014.

[16] Anbunathan R and Anirban Basu. "Automation framework for testing Android mobiles", International Journal of Computer Applications, Vol. 106, No. 1, 25-31, November 2014.

[17] Anbunathan R and Anirban Basu. "An Event based Test Automation Framework for Android Mobiles", IEEE First International Conference on Contemporary Computing and Informatics(IC3I), 2014.

[18] D. Amalfitano, A. R. Fasolino, P. Tramontana, "A GUI Crawling-based technique for Android Mobile Application Testing", IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011.

[19] Lu Lu, Yulong hong, Kai Su, Yuping Yan, "Activity Page based Functional Test Automation for Android Application", IEEE Third World Congress on Software Engineering (WCSE), 2012.

[20] Google Code. Robotium. Available at: https://code.google.com/p/robotium/. Last accessed Nov. 29, 2014.

[21] Android Developers. Monkeyrunner. Available at: http://developer.android.com/tools/help/monkeyrunner_concepts.html. Last accessed Nov. 29, 2014.