ECE655 Survey Report: Test Automation Tools for Mobile Apps

Xiyue Zhang (20601564)

***Abstract*—the objective of this paper is to compare popular test automation tools and analyze the problems to test GUI and other mobile special features. Then some strategies are given to help generating proper test case and improve testing efficiency. For GUI testing, there is model-based testing to build GUI model and generate test case based on this model. For other mobile features like location and sensors, there is contextual fuzzing approach to combine features into representative test case and meet the testing requirement.**

Index Terms – Automation Test, GUI, Model, Context

I. INTRODUCTION

With the increasing development of mobile product, there is more mobile software on app store available for user to download. As one of the software developing process, testing is important step to check whether the application meets functional and quality requirement. If user find bug the first time he used the application, he is very likely to uninstall this application. However developer may not have time to do manual testing because there are many test cases. So they prefer to use testing automation tools. According to the existing tools, most of them are focus on unit test. Developers need to create testing script first and the tools can run the script automatically. However it also takes time to design test case. Here are some cases as below to show it is in great need to generating test case automatically.

A. GUI and Integration

GUI (graphical user interface) is a type of user interface that allows user to interact with electronic device through graphical icons and visual indicators. Beyond computers, GUIs are used in handheld mobile devices [1]. GUI testing is the process of testing graphical user interface to ensure it meets its specifications. Before GUI testing developers need to design test case which can cover all the functionality of the system and fully exercise the GUI [2].

However GUI has many operations that need to be tested and it is common that small app contains more than 10 windows and 1000 possible testing path. Besides developer many miss some important test scenarios which can raise bug during test case design. According to the available Android automation testing tool, most of them generate GUI test case by writing script or converting action into script for certain UI logic script. Even though the tools can help developers to run the test case automatically, developers have to write many test case and sometimes rewrite test case just because a small UI/logic changes. As we can see that many apps are updated frequently just to change the layout without any functionality update, it is also common that developer need to do many manual work on GUI testing.

This paper introduces the model-based testing for GUI testing in chapter 3. The model is extract from the GUI structure and event, then used to generate test case. There will be two model-based testing strategies. The overall thought is similar except GUI model and the method to extract model.

B. Mobile Specific Features

There are many features for mobile testing such as mobility, location services, sensors or different gestures and inputs. In unit testing, each test case chose a feasible group of mobile feature parameters as input and check whether it meets the certain requirement. However there will be so many possible combinations of feature parameters to test. We do not know representative combination for running app, and which one is possible to throw exception or bug.

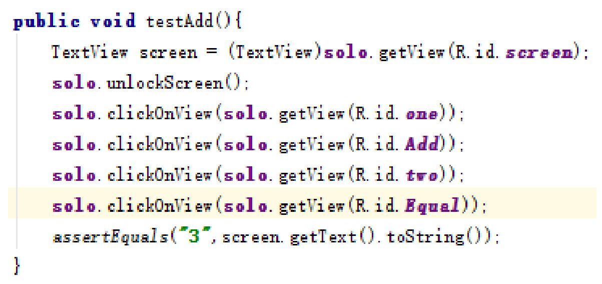
This paper introduces a contextual fuzzing approach in chapter 4. The approach describes the mobile feature as contexts. The testing for context is difficult because test input needs to be run with various contexts. In addition the range of values for each context can be very large such as location. To find a representative test case, the paper introduce a cloud service called Caiipa to test apps over an expanded mobile context space in a scalable way.

II. EXISTING AUTOMATION TOOL

There are many automation testing tools which is widely used by developer. In this chapter we pick up some popular tools and analyze their features and raise the reason why to improve GUI testing and mobile special feature testing.

A. Robotium

Robotium is an extension of the JUnit library for Android UI testing a useful method. It provides automated black box test cases for native and hybrid Android applications. Developers can write functional, system, and acceptance test scenarios on Robotium. The tests are written in Java. For its advantage, it is the basic tool to test android application, so it supports most of Android versions and sub-versions. For its disadvantage, it takes time to create a test case by writing source code. And it is unsuitable for interaction with system software, for example, it can’t lock and unlock a smartphone or a tablet. Robotium itself have no Recorder or screenshot function. However Robotium Recorder as an extended recording tool can create test scripts. By performing actual actions on your real device, it records every step on device and converts these operations to Javascript test script.[3]

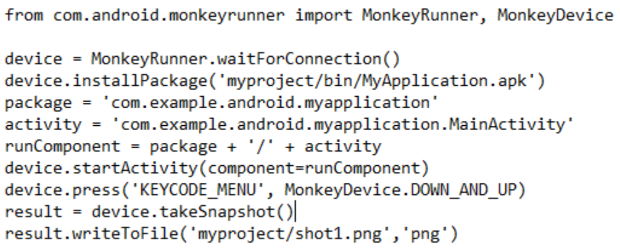


```
public void testAdd(){
    TextView screen = (TextView)solo.getView(R.id.screen);
    solo.unlockScreen();
    solo.clickOnView(solo.getView(R.id.one));
    solo.clickOnView(solo.getView(R.id.Add));
    solo.clickOnView(solo.getView(R.id.two));
    solo.clickOnView(solo.getView(R.id.Equal));
    assertEquals("3", screen.getText().toString());
}
```

Figure 2.1 Robotium test script

B. MonkeyRunner

MonkeyRunner is lower than Robotium and its test case are written in Python. It can run the test on the actual device connected to the PC or emulator. The tool has an API that allows it to control real device or emulator from outside Android code. It has recorder to create a test script but developers have to create script for each device and recreate test case due to UI changes[4].



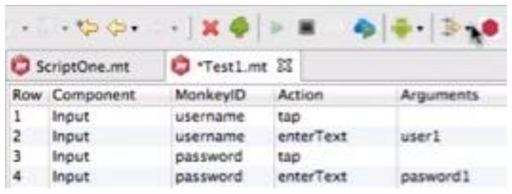
```
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice

device = MonkeyRunner.waitForConnection()
device.installPackage('myproject/bin/MyApplication.apk')
package = 'com.example.android.myapplication'
activity = 'com.example.android.myapplication.MainActivity'
runComponent = package + '/' + activity
device.startActivity(component=runComponent)
device.press('KEYCODE_MENU', MonkeyDevice.DOWN_AND_UP)
result = device.takeSnapshot()
result.writeToFile('myproject/shot1.png','png')
```

Figure 2.2 MonkeyRunner test script

C. MonkeyTalk

MonkeyTalk consists of three components: IDE, Agent, and script. The IDE creates a test script using logging and playback. Agent is a test tool library to which an application is linked. The MonkeyTalk script uses simple keyword syntax and an Ant or Java execution engine. Tests can be data-driven from a spreadsheet using the CVS format [5].



Row	Component	MonkeyID	Action	Arguments
1	Input	username	tap	
2	Input	username	enterText	user1
3	Input	password	tap	
4	Input	password	enterText	pasword1

Figure 2.3 MonkeyTalk Recorder

D.Appium

Appium supports Android versions from 2.3 and later. It contains a NodeJs server which is implemented by Javascript. Appium supports testing on Android Chrome or its embedded "browser" app. Since Appium is based on WebView and WebView can interact with various elements of a web page. As most platforms have browser, Appium can support cross-platform testing. Besides Appium make a higher level packaging WebDriver for automated testing framework such as UiAutomator and Selendroid. The WebDriver specifies a set of client-server protocols that allow clients do HTTP requests regardless of the language they are written in. So Appium is a useful framework that developer can design their own testing environment, which can flexible to use other automation tools and different programming languages [6].

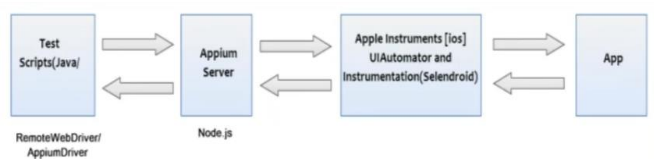
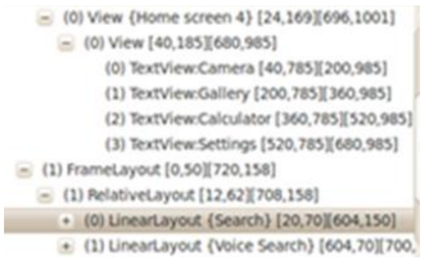

Test Scripts(Java/
RemoteWebDriver/ AppiumDriver
Appium Server
Node.js
Apple Instruments [ios] UIAutomator and Instrumentation(Selendroid)
App

Figure 2.4 Appuim Framework

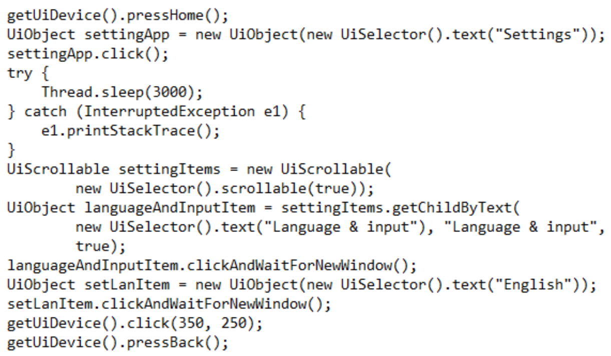
E. UIAutomator

UI Automator supports Android versions beginning from 4.1. It a testing framework provides a set of APIs to build UI tests that performs interactions on user apps and system apps. The APIs allows you to perform system operations such as opening the Settings menu or unlock the device for the app to launch. UI Automator testing framework has three main component which are UI Automator Viewer(to inspect layout hierarchy), UiDevice class (to access and perform operations on the device) and UI Automator APIs(to capture and manipulate UI components across multiple apps) [7]



```
(0) View {Home screen 4} [24,169][696,1001]
  (0) View [40,185][680,985]
      (0) TextView:Camera [40,785][200,985]
      (1) TextView:Gallery [200,785][360,985]
      (2) TextView:Calculator [360,785][520,985]
      (3) TextView:Settings [520,785][680,985]
(1) FrameLayout [0,50][720,158]
  (1) RelativeLayout [12,62][708,158]
    (0) LinearLayout {Search} [20,70][604,150]
    (1) LinearLayout {Voice Search} [604,70][700,
```

Figure 2.5 UI Automator Viewer



```
getUiDevice().pressHome();
UiObject settingApp = new UiObject(new UiSelector().text("Settings"));
settingApp.click();
try {
    Thread.sleep(3000);
} catch (InterruptedException e1) {
    e1.printStackTrace();
}
UiScrollable settingItems = new UiScrollable(
        new UiSelector().scrollable(true));
UiObject languageAndInputItem = settingItems.getChildByText(
        new UiSelector().text("Language & input"), "Language & input",
        true);
languageAndInputItem.clickAndWaitForNewWindow();
UiObject setLanItem = new UiObject(new UiSelector().text("English"));
setLanItem.clickAndWaitForNewWindow();
getUiDevice().click(350, 250);
getUiDevice().pressBack();
```

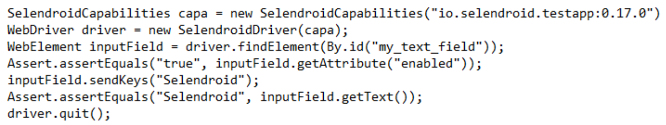
Figure 2.6 UI Automator Test Script

F. Selendroid

Selendroid drives off the UI of Android native app, hybrid apps and the mobile web. Test cases are written by Selenium

2 client API.[8]. Selendroid is based on Android instrument so there are lots of great features which are helpful for test script creation, execution and automation process:

- App under test requires no modifications
- Gestures like not-just-click are fully supported
- Large set of devices testing works fine with it
- UI elements can be found with varying locators



```
SelendroidCapabilities capa = new SelendroidCapabilities("io.selendroid.testapp:0.17.0")
WebDriver driver = new SelendroidDriver(capa);
WebElement inputField = driver.findElement(By.id("my_text_field"));
Assert.assertEquals("true", inputField.getAttribute("enabled"));
inputField.sendKeys("Selendroid");
Assert.assertEquals("Selendroid", inputField.getText());
driver.quit();
```

Figure 2.7 Selendroid Test Script

According to testing strategy above, we can summarize two common testing tools strategies. Firstly most of test cases are based on test script. The developer can write script code to design unit test for functional requirement. Secondly some of tools have recorder to convert device operation into test script. In this way developer don't need to write script for every unit test. Both of the strategy can save time on duplicate testing operation. However, they all need manual operation to generate test case by either writing script or record action.

For GUI testing, it takes time to analyze all possible event sequence so that test cases are too many to finish by manual work. For mobile feature testing, there are still many possible feature combinations as test cases. What is worse, the test case we design may not meet real environment requirement, for example the location is right and internet does not meet the location condition. So this paper will give testing strategy on how to automatically generate test case which is fit for testing requirement.

III. GUI Test case Generation

A. GUI Testing

As shown in chapter 1, there are too many GUI test cases to be finished by manual work for developer. Moreover, each test case must meet certain app functionality. GUI test case is the possible sequences of events for the app. The sequence of event is based on GUI operations. So to design test case, developer must capture the right sequence of event from dynamic app. For example, the test case to open a file is that the user has to first click on the File Menu, select the open operation, use a dialog box to specify the file name, and focus the application on the newly opened window. These possible operations increase the sequencing problem.

B. Model-based testing

To capture the sequence of event, we need to build a GUI model and search the possible sequence based on that. There is a testing method called model-based testing to solve this problem. Model-based testing can support the GUI testing by building a proper test case automatically. It is an executing artifact to perform software testing or system testing. Models can be used to represent the desired behavior of a system under test or to represent testing strategies and a test environment [9].

According to the Figure 3.1, Models represent desired behavior of a system under test. The executable test case is generated based on model. The test case generation step is like this: Firstly abstract test suite derived from a model. Then executable test suite derived from abstract test suite. Finally executable test suite can communicate directly with the system under test. Because test suites are derived from models and not from source code, model-based testing is usually seen as one form of black-box testing

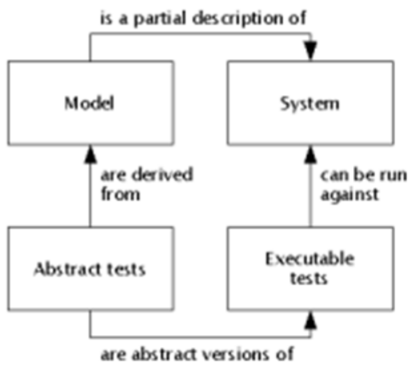


Figure 3.1 General model-based testing setting

According to GUI testing and its countless test case, the most important step for model-based testing is to build a proper model which can guide tools to generating test case. If a model is machine-readable and formal to the extent that it has a well-defined behavioral interpretation, test cases can in principle be derived mechanically.

There are some results of recent work on model-based testing. The popular models include Event-Sequence Graph, Event-Interaction Graph, Event-Flow Graph, and Finite State Machine. However, each model can only applied to certain GUI testing problem, for example, some model are fit for small app. Some model focus on user action to change UI states and ignore user input value. Even for the same model, there are different methods to build that. Here we introduce two models and their method to build mode.

1) GUI Ripper

GUI ripping is a dynamic process that is applied to executing software GUI. Starting from the software's first window, the GUI is traversed by opening all child windows. All the window's widgets (e.g., buttons, text-boxes), their properties (e.g., background-color, font), and values (e.g., blue, 12pt) are extracted [11].

a) GUI Ripper Model

The models extracted by GUI ripping include GUI forest, Event-flow graph and Integration tree. GUI forest shows windows structure, Event-flow graph shows the execution behavior and event states, integration tree can help generate event-flow graphs based on GUI forest.

Definition for GUI forest
GUI forest represents the structure of the GUI's windows and relationship between windows.

GUI forest is a triple < W, T ,E >

- W is the set of windows, the window as a node of GUI forest including the widgets (e.g., buttons, text fields), properties of widgets (e.g., color, size), and values of properties (e.g., red, 16pt).
- T ⊆ W is a set of windows called the top-level windows. Top-level window is the window presented to the user when app is launched.
- E is the set of directed edges: there is an edge from x to y, if window y is opened by performing an event in the window x.

Definition for Modal window:
Modal window is a GUI window. When the modal window start, it will restrict the focus of the user to a specific range of events within the window, until the window is explicitly terminated that. For example when the user opens a window, it terminates the interaction by either clicking OK or Cancel.

Definition for GUI component:
A GUI component C is an ordered pair (RF, UF), where RF represents a modal window in terms of its events and UF is a set whose elements represent modeless windows also in terms of their events. Each element of UF is invoked either by an event in UF or RF

Definition for Event-flow Graphs:
An event flow graph represents all possible interactions among the events in a component. An event-flow graph is created by identifying the events in a GUI component. A GUI component will exploit the GUI's hierarchy to identify GUI events sequence.

Event-flow graph for a component C is a tuple <V, E, B, I>

- V is a set of vertices representing all the events in the component
- E is a set of directed edges between vertices.
- B is a set of vertices representing events of C that are available to user when the component is first invoked.
- I is the set of restricted-focus events of the component.

There is an additional structure called integration tree which can identify event interactions among components. Then event-flow graphs along with the integration tree show the flow of events in the GUI. Once we have the event-flow graphs and integration tree, we can generate test cases by traversing these structures and enumerating the events encountered. A large number of test cases can be obtained quickly in this manner.

b) GUI Ripper Process

The process of GUI Ripping consists of two steps: Firstly GUI Ripper automatically traversed the GUI of the application extract its structure. Secondly the developer visually inspects the extracted GUI structure and makes corrections so that the structures confirm to software specifications.

(1) Structure Extraction

As shown in Figure 3.2. The algorithm is based on DFS. There are two procedures DFS-GUI and DFS-GUI-Recursive traversing the GUI and extract GUI structure.

```
$\mathcal{GUI}$ /* GUI tree of application under test */
PROCEDURE DFS-GUI(Application $A$)
  $\mathcal{T}$ = access-top-level-windows($A$)
  $\mathcal{GUI} = \mathcal{T}$
  /* $\mathcal{T}$ is set of top-level windows in the application */
  FORALL $t \in \mathcal{T}$ DO
    DFS-GUI-Recursive($t$)

PROCEDURE DFS-GUI-Recursive(Window $g$)
  $\mathcal{W}$ = get-widget-list-and-properties($g$)
  /* $\mathcal{W}$ is the set of all widgets in the Window */
  $\mathcal{E}$ = identify-executable-widgets ($\mathcal{W}$)
  /* From $\mathcal{W}$ identify executable widgets */
  FORALL $e \in \mathcal{E}$ DO
    execute-widget($e$)
    /* Execute the widget e */
    $\mathcal{C}$ = get-invoked-gui-windows($e$)
    $\mathcal{GUI} = \mathcal{GUI} \cup g$
    FORALL $c \in \mathcal{C}$ DO
      DFS-GUI-Recursive($c$)
```

Figure 3.2 Model Extracting Algorithm

In DFS-GUI, it returns the list of top-level windows, which become visible when the application is first launched. Then DFS-GUI-Recursive runs with these top-level windows.

In DFS-GUI-Recursive, a GUI window may contain several widgets and widgets can invoke other GUI window. Firstly the call to get-widget-list-and properties returns a list W of the constituent widgets in the GUI window. Then the function identifies a list E of executable-widgets that can invoke other GUI window. When executing e in E, e invokes a list C of GUI windows. Then DFS-GUI-Recursive runs with these GUI windows one by one.

When the procedure DFS-GUI-Recursive returns to DFS-GUI, the tree rooted at the top-level window is constructed and when the algorithm procedure finishes, we can get GUI forest of the app under test. During the traversal of the GUI, we also determine the event type by using low-level system calls. Once this information is available, we can create the event-flow graphs and integration tree

(2) Manual Inspection

The automated ripping process is not perfect and sometimes missing windows, widgets, and properties. For example, it cannot distinguish between modal and modeless windows; it cannot extract the structures of the Print dialog in

Java. So developer should detect the missing part by the ripper, and add it to the GUI forest at an appropriate location.

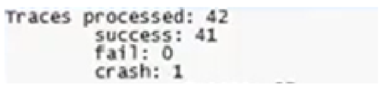
c) Implemented Tool

AndroidRipper extends GUI ripping to automatically and systematically traverse the Android app's UI, generating and executing test cases. It dynamically analyses the GUI to get sequences of events fireable through the GUI widgets. Each sequence of event provides an executable test case. AndroidRipper maintains a state machine model of the GUI, which we call a GUI Tree. The GUI Tree model contains the set of GUI states and state transitions encountered during the ripping process. The ripping technique is iterative and relies on the following concepts:

- An event is a user action performed on a GUI widget
- A task is a couple (action, GUI state) representing an action performed in a GUI state
- An action consists of a sequence of zero or more data input events followed by a single command input event.
- GUI exploration criterion is a logical predicate (composition of conditions) that establishes if the exploration of a given GUI must be continued (true value) or stopped (false value)

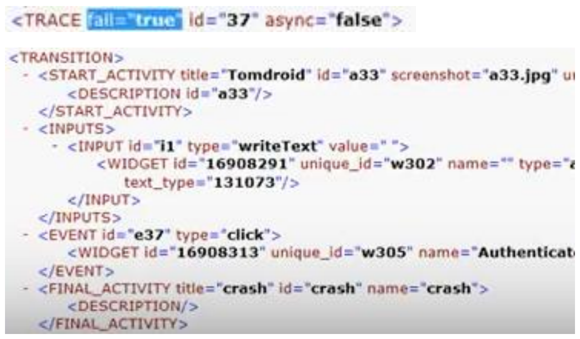
AndroidRipper design is based on executing tasks in a task list, initialized with tasks that are fireable in an initial GUI of the application, while the GUI Tree just contains a single state. The task list is iteratively updated with new tasks defined from the current GUIs, and new states and state transitions are added to the GUI Tree.

AndroidRipper is implemented by Robotium Framework and Android Instrumentation. Here are picture to show a testing result for a certain tested app. As shown in Figure 3.3 to Figure 3.5, Test Case State Record shows how many traces have been processed, in other word, how many possible test cases are generated. Also how many test cases are crashed or successful. Test Case Trace XML file shows the detain test case information about how to generate test case based on the possible transaction, where is the corresponding screenshot, and which test case is crashed. Test Case Script is the generated Junit test case script. As AndroidRipper is based on Robotium, the testing script is mainly Robotium style. To find the crash test, we can refer the xml file and script to see the crashed test case information. In this way every reachable app statement can be executed by running at least one of the event sequences included in the model.



```
Traces processed: 42
        success: 41
        fail: 0
        crash: 1
```

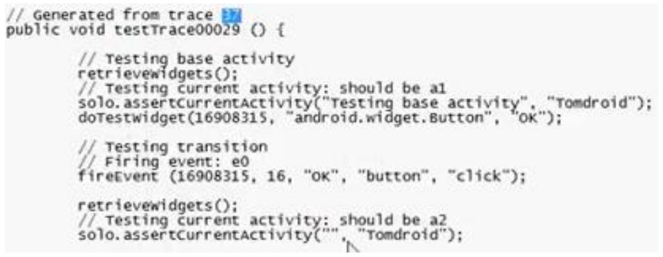
Figure 3.3 Test Case States Record



```
<TRACE fail="true" id="37" async="false">

<TRANSITION>
  - <START_ACTIVITY title="Tomdroid" id="a33" screenshot="a33.jpg" u
        <DESCRIPTION id="a33"/>
    </START_ACTIVITY>
  - <INPUTS>
      - <INPUT id="i1" type="writeText" value=" ">
            <WIDGET id="16908291" unique_id="w302" name="" type="
                text_type="131073"/>
        </INPUT>
    </INPUTS>
  - <EVENT id="e37" type="click">
        <WIDGET id="16908313" unique_id="w305" name="Authenticat
    </EVENT>
  - <FINAL_ACTIVITY title="crash" id="crash" name="crash">
        <DESCRIPTION/>
    </FINAL_ACTIVITY>
```

Figure 3.4 Test Case Trace XML



```
// Generated from trace 37
public void testTrace00029 () {

        // Testing base activity
        retrievewidgets();
        // Testing current activity: should be a1
        solo.assertCurrentActivity("Testing base activity", "Tomdroid");
        doTestWidget(16908315, "android.widget.Button", "OK");

        // Testing transition
        // Firing event: e0
        fireEvent (16908315, 16, "OK", "button", "click");

        retrievewidgets();
        // Testing current activity: should be a2
        solo.assertCurrentActivity("", "Tomdroid");
```

Figure 3.5 Test Case Script

However, the objective of GUIRipper and AndroidRipper is to stress-test the app rather than to create reusable model for future testing. We need to find a more useful model building method to fit for general test case generation. This approach is the following content about Grey-Box Approach.

2) Grey-Box Approach

a) Grey-Box Approach Model

Different from GUI forest model from GUI Ripper, Grey-Box Approach models the GUI behavior of app as a finite-state machine. However, the aim of this approach is to exploit the simple and intuitive GUI design of mobile apps to derive a compact high-quality model. It captures GUI components which support user actions, and ignore UI state resulting from data values input by the user. In this way the model avoids large, potentially infinite number of UI states.

b) Grey-Box Approach Process

There are mainly two steps to extract model, which are static analysis for source code and dynamic crawling for model. Static analysis is to extract the set of user actions supported by each widget in the GUI, and static method means to analyze app's source code without running the app. Dynamic crawler is used to reverse-engineer a model by exercising extracted actions on the running app.

(1) Static analysis

Static analysis is used to infer actions. As the Android framework a user action is defined by either registering an appropriate event listener for it or inheriting the event handling method of an Android-framework component. We term the former as registered action and the latter as inherited action. For both these categories, identifying an action involves three basic steps:

- Step1: It identifies the place where action is instantiated or registered
- Step2: It locates the component on which action would be fired
- Step3: It extracts an identifier of the component that the crawler can later use to recognize the corresponding object and fire the action.

```
Input   : $\mathcal{A}$: app source code
Output: $\mathcal{E}$: action map
$ActionSet \leftarrow getAllActions()$
$EntryPoints \leftarrow getAllEntryPoints()$
foreach $\mathcal{P} \in EntryPoints$ do
    $CG \leftarrow makeCallGraph(\mathcal{A}, \mathcal{P})$
    foreach $\mathcal{X} \in ActionSet$ do
        $\mathcal{L} \leftarrow getEventRegMethod(\mathcal{X})$
        $PNodeSet \leftarrow getParentNode(CG, \mathcal{L})$
        foreach $PNode \in PNodeSet$ do
            $s \leftarrow findCallTo(PNode, \mathcal{L})$
            $v \leftarrow getCallingObject(s)$
            $i \leftarrow backLocate(v, \mathcal{A})$
            $ID \leftarrow getParameter(i)$
            $\mathcal{E}.add(ID, \mathcal{X})$
        end
    end
end
```

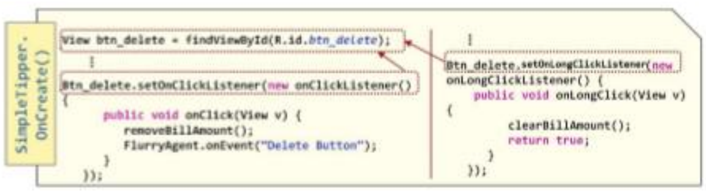
Figure 3.6 Registered Action Detection

Algorithm 1 describes the registered actions detection. It iterates over program-entry points (EntryPoints) and actions (ActionSet). For each entry point P and action X, it extracts the call graph of the app and locates a set of statements PNodeSet containing instances of a valid event-listener registering statement L for action X. Finally, for each statement PNode in PNodeSet it performs a backward slice on PNode to locate an initialization statement of the widget on which the instance of L was called. The backward slice is used to get an identifier ID of the component that is registered in the action map E with the action X. get ID-action pair map

```
Input   : $\mathcal{A}$: app source code
Output: $\mathcal{E}$: action map
$ActionSet \leftarrow getAllActions()$
$CH \leftarrow getClassHierarchy(\mathcal{A})$
$Klass \leftarrow getUserClass(CH)$ // Get user defined
foreach $Class \in Klass$ do
    foreach $\mathcal{X} \in ActionSet$ do
        $\mathcal{L} \leftarrow getActionHandlMethod(\mathcal{X})$
        $\mathcal{M} \leftarrow getDeclaredMethod(Class, \mathcal{L})$
        if $\mathcal{L} \in \mathcal{M}$ then
            $ID \leftarrow getNameOrID(\mathcal{L}, \mathcal{M})$
            $\mathcal{E}.add(ID, \mathcal{X})$
        end
    end
end
```

Figure 3.7 Inherited Action Detection

Algorithm 2 describes the inherited action detection. Frist we get class hierarchy CH of the whole app. Then, we use app's namespace to filter non-user-defined class e. For each of the user-defined classes, if the class overrides the action handling method L, we regard the action X as valid, then we extract the Activity name or registered ID of the class, and add the ID-action pair in the action mapping.



```
View btn_delete = findViewById(R.id.btn_delete);
    ⋮
Btn_delete.setOnClickListener(new onClickListener()
{
        public void onClick(View v) {
           removeBillAmount();
           FlurryAgent.onEvent("Delete Button");
        }
    });
```

```
    ⋮
Btn_delete.setOnLongClickListener(new
onLongClickListener() {
     public void onLongClick(View v)
{
          clearBillAmount();
          return true;
       }
   });
```

Figure 3.8 static analyze for action inference

As shown in Figure 3.8, developer defines click and longClick actions on the button DEL. To identify components on which to fire longClick, we first use the call graph to find the methods where setOnLongClickListener is called. It happens to be called in method onCreate of activity SimpleTipper. Then we locate the statement calling setOnLongClickListener in onCreate and get object btn delete that the listener is registered to. Finally we backslice to get the initialization statement of btn delete, get its ID btn delete, and add the ID-action pair to the action mapping. Thus, when the crawler encounters a screen with component btn delete, it fires a longClick on it.

Action-inference Algorithm presents the analysis to detect registered or inherited actions, and generates the action mapping based on call-graphs. The action mapping is used for Dynamic crawler to explore states graph and fire actions in action mapping.

(2) Dynamic crawler

The objective of the crawling algorithm is to exhaustively explore all the app's states by firing open actions. The crawling process ends if the model has no open states. This process can potentially be done through a simple DFS algorithm on the UI states. However, the key challenge here is the backtracking step.

```
Input   : $\mathcal{A}$: app under test, $\mathcal{E}$: action map
Output: $\mathcal{M}$: crawled model
begin
    $\mathcal{M} \leftarrow \emptyset$; $s \leftarrow getOpeningScreen(\mathcal{A})$
    while $s \neq null$ do
        $s \leftarrow forwardCrawlFromState(s, \mathcal{A}, \mathcal{M}, \mathcal{E})$ // forward crawl
        $s \leftarrow backtrack(s, \mathcal{A})$
        if $isInitialState(s)$ then $s \leftarrow findNewOpenState(s, \mathcal{M}, \mathcal{A})$
    end
end
```

Figure 3.9 Crawling Algorithm

Mobile provides a Back button to undo actions. But this button is designed for app navigation and is context-sensitive. Thus, it is not a reliable mechanism for backtracking to precisely the previous state. For example, on state 5d of Figure 3.10, pressing the Back button will not lead us back to previous state 5b or 5c, but to previous screen. Thus, the Back button need not take the navigation back to the immediately preceding state but to any of its ancestors. Hitting Back a finite number of times will take the app to the initial screen. Our crawler uses a modified depth-first search, which tries to crawl only "forward" as much as possible using the Back button to backtrack when needed.

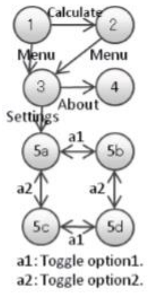
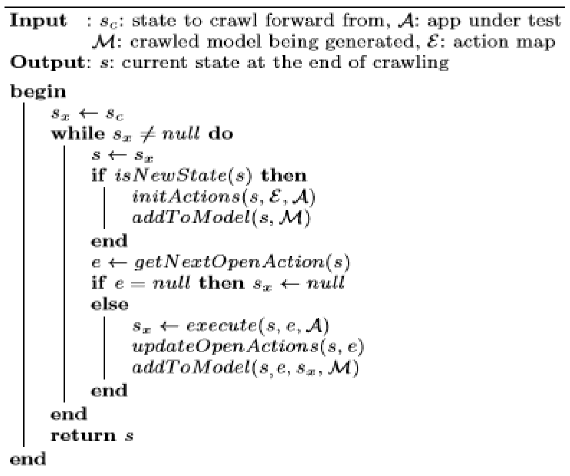


Figure 3.10 App State Graph

Crawling Algorithm describes this strategy. It repeats a sequence of three steps until it can make no further progress at terminate point.

- Step1: The function forwardCrawlFromState() is to recursively visit states with open actions. It fires an open action and continues crawling until it reaches a state with no open actions.
- Step2: The function backtrack() is called to backtrack from the current state until another open state is found or one of the initial states of the crawl model is reached.
- Step3: In the former case forward crawling is resumed from this open state. In the latter case the function findNewOpenState() is used to find and crawl to a new open state and forward crawling is continued from there.



```
Input  : $s_c$: state to crawl forward from, $\mathcal{A}$: app under test
         $\mathcal{M}$: crawled model being generated, $\mathcal{E}$: action map
Output: $s$: current state at the end of crawling
begin
    $s_x \leftarrow s_c$
    while $s_x \neq null$ do
        $s \leftarrow s_x$
        if $isNewState(s)$ then
            $initActions(s, \mathcal{E}, \mathcal{A})$
            $addToModel(s, \mathcal{M})$
        end
        $e \leftarrow getNextOpenAction(s)$
        if $e = null$ then $s_x \leftarrow null$
        else
            $s_x \leftarrow execute(s, e, \mathcal{A})$
            $updateOpenActions(s, e)$
            $addToModel(s, e, s_x, \mathcal{M})$
        end
    end
    return $s$
end
```

Figure 3.11 forwardCrawlFromState()

Algorithm 4 implements forwardCrawlFromsState() for forward crawling from a given state. It iterates the same operation for the each current state. getNextOpenAction() is to obtain an open action e on s. if e is open action, then e is executed to reach another open state. updateOpenActions() is to update open actions of s. Then addToModel() is to add transition s→(e)→sx into the model.

To completely crawl the sub-graph formed by states 5a-5d in Figure 3.10, the standard DFS would need to backtrack several times whereas our algorithm would cover it in a single forward crawl through the sequence Menu → Settings → a1 → a2 → a1 → a2 → a2 → a1 → a2 → a1, by continuing to fire open actions.

c) Grey-Box Approach Implement

1 Action Detector:

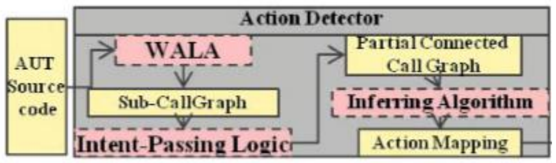


Figure 3.12 Action Detector Framework

WALA static-analysis framework: Android apps are event-driven and organized as a set of event-handler callback methods. This module is to search the app code and gives a set of partial, disconnected sub call-graphs.

intent-passing logic: This module builds a mapping of intent-sending methods and intent filters by analyzing the app's source code and manifest file. This mapping connects the sub call graphs into a partial connected call graph. It is partial because for some intent-passing mechanisms like intent broadcasting whose behavior is affected by the runtime state of the Android system, we are unable to infer this information statically.

Inferring algorithm: This module applies action-inference algorithm on the partial connected call graph to generate the action mapping

2 Dynamic Crawler

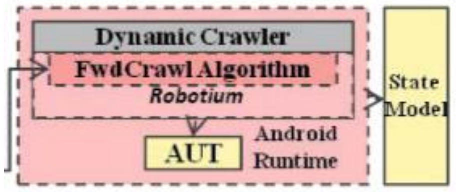


Figure 3.13 Dynamic Crawler Framework

Dynamic crawler is built on top of the Robotium. The crawler gets the list of actions from the Action Detector, it implements special handling for certain components such as dynamically-created GUI components and system-generated GUI components that are not statically declared.

For dynamically-created GUI components (ListView), crawler represents the container as one of the two abstract states: an empty list and a non-empty list. Further, it randomly chooses only one of the child components to crawl by firing actions defined in the container.

For system-generated GUI components (system-generated context menu), crawler identifies such components at runtime and systematically crawls each child, rather than treating it as a generic container.

IV. Mobile Specific Features Testing

In terms of widely used mobile context, there are three contexts capturing the most of context-related bugs, which are Wireless Network Conditions (Wi-Fi, 3G), Device Heterogeneity (memory, CPU) and Sensor Input (GPS, compass).

As shown in chapter 1, the test case of mobile feature is hard to generate. There are some techniques as below to analyze the comprehensive context and offer test strategy, but they are not fit for some condition as below.

- Tools for collecting and analyzing data logs from already deployed apps can send bug information to developer. However it lacks broad coverage because local condition of certain app is likely not representative of a public app release.
- Inducing issues via fault injection could be used to test consequences of specific situations. However, it needs to choosing which issues to inject
- There may be platform simulators to test the app under a specific GPS location and network type. However these simulators can't provide a way to systematically explore representative context combination in real environment.
- Crashes information of previous test case could guide the next test case generation, but the problem is to map previously observed scenarios well to performance analysis. Even if such data is available, one still has the problem to identify the specific problematic contexts to use for testing.

There is test case predict method called contextual fuzzing to solve the problems above. The Caiipa as a cloud server, combine contextual fuzzing with other service to generate test case for various mobile contexts.

A. System component

Caiipa runs the app under AppHost either in an emulator or on real hardware, while simulating various contexts using the Perturbation Layer. It continuously monitors the performance of the app under each context; and PerfAnalyzer outputs a report with all cases where it found the app to have a bug,

As running all possible combinations of contexts is not feasible. we propose two techniques. First, ContextLib uses machine learning techniques to identify representative contexts by determining combinations of contexts that are likely to occur in the real world, or removing redundant combinations of contexts. Second, ContextPrioritizer sorts different context combinations, and runs those with higher likelihood of detecting failures before others. This helps Caiipa to quickly discover the problematic scenarios. AppHost Dispatcher reads the prioritized test cases from ContextPrioritizer and sends each test to the appropriate AppHost.

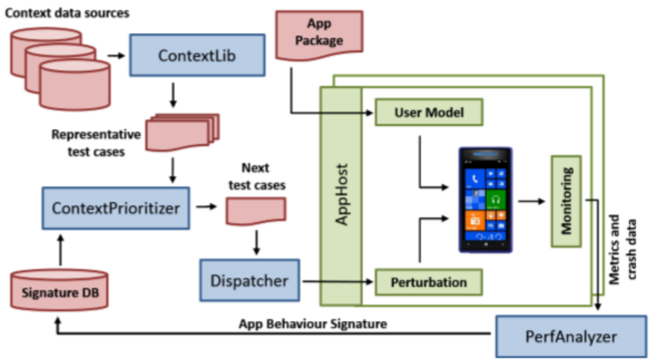


Figure 4.1 Caiipa Server Framework

ContextLib performs a two-stage test case generation process that first filters redundant contexts that do not significantly impact app behavior such as resource consumption), then generating test cases not from just individual contexts such as network condition, but also based on context transitions such as 3G to 4G and the co-occurrence contexts of different types such as fast network with low memory.

ContextPrioritizer receive contexts from ContextLib to determine the contexts order and communicates this ordering to AppHost Dispatchers. App behavior like crashes and resource use is used by ContextPrioritizer to build and maintain app similarity measurements that determine this prioritization. The result from prioritization process:

- The redundant or irrelevant contexts are ignored, for example an app is discovered to never use the network, so network contexts are not used
- The contexts that negatively impact similar apps are prioritized, for example an app that behaves similarly to streaming apps would have network contexts prioritized.

AppHost Dispatcher makes decision to run test case on a VM or on real hardware for AppHost. For In order to scale testing, most test cases are run on a VM, which AppHost Dispatcher selects from a AppHost pool. However there are two situations for AppHosts running on real devices:

- The app is native or mixed code and unable to be run on an emulator, or when an app uses specific hardware that is difficult to emulate
- when ContextPrioritizer selects any of the small number of device-related test cases included in ContextLib .

AppHost runs apps based on AppHost Dispatcher selection. The AppHost has three main functions:

UI Automation: This module uses a GUI Model to generates user events based on weights. The procedure is similar to GUI Ripper in chapter 3.

Perturbation: This module simulates conditions, such as CPU performance levels, amount of available memory, GPS location report, different network parameters to simulate different network interfaces such as Wi-Fi, GPRS, network quality levels, network transitions such as 3G to Wi-Fi or handoffs between cell towers. This layer is extensible and new contexts can be added as required.

Monitoring: During test execution AppHost closely records app behavior in the form of a log of system-wide and per app performance counters (e.g., network traffic, disk I/Os, CPU and memory usage) and crash data. To accommodate the variable number of logging sources (e.g., system built in services and customized loggers), AppHost implements a plug-in architecture where each source is wrapped in a monitor. Monitors can either be time-driven (i.e., logging at fixed intervals), or event-driven (i.e., logging as an event of interest occurs, like specific UI events).

PerfAnalyzer. This component identifies crashes, and possible bugs (e.g. battery drain, data usage spikes, long latency) in the large pool of monitoring data generated by AppHosts. To identify failures that do not result in a crash, it uses anomaly detection that assumes norms, based on previous behavior of (1) the target app and (2) an app group that are similar to the target app. Crashes by themselves provide insufficient data to identify their root cause. Interpretation of them is key to providing actionable feedback to developers. PerfAnalyzer processes crash data across crashes to provide more focused feedback and helps narrow down the possible root cause. Also, if the app developer provides debug symbols, it can find the source code location where specific issues were triggered. PerfAnalyzer augments the bug data with relevant contextual information to help identify the source of problems. The generated report includes resource consumption changes prior to crash, the set of perturbed conditions, and the click trace of actions taken on the app along with screenshots context, thus documenting its internal computation state and how the app got there; which is missing in regular bug tracking systems or if only limited data from the crash moment is available.

B. Prioritizing Test Case

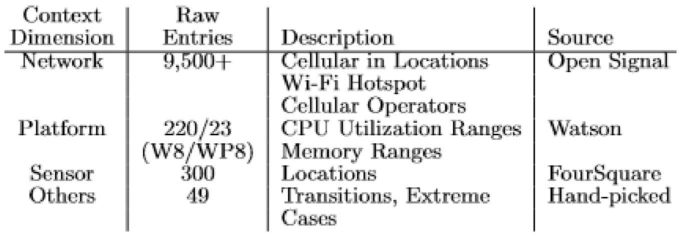
ContextLib maintains a test case collection that runs into the thousands. At this scale, the latency and computational overhead of performing the full test suite becomes prohibitive to users. To address this challenge, ContextPrioritizer finds a unique per-app sequence of test cases that increases the marginal probability of identifying crashes (and performance issues) for each test case performed. The key benefit is that exercising only a fraction of the entire ContextLib can still discover important context-related crashes and performance bugs.The role of ContextPrioritizer is to determine the next batch of test cases (chosen from ContextLib) to be applied to the user provided test app.

The underlying approach of ContextPrioritizer is to learn from prior experience when prioritizing test cases for a fresh unseen test app. ContextPrioritizer first searches past apps to find a group that should perform similarly to the current test app (referred to as a AppSimSet). And then, it examines the test case history of each member of the AppSimSet, identifying potentially "problematic" test cases that may result in faults. These problematic test cases are then prioritized ahead of others in an effort to increase the efficiency by which problems in the current app are discovered.

While we informally refer to apps in this description (e.g., test app, or apps in a AppSimSet), more precisely this term points to an app package that includes both (1) a mobile app and (2) an instance of a User Interaction Model (see §4). This pairing is necessary because the code path and resource usage of an app are highly sensitive to the user input. Conceptually, an app package represents a particular usage scenario within an app.

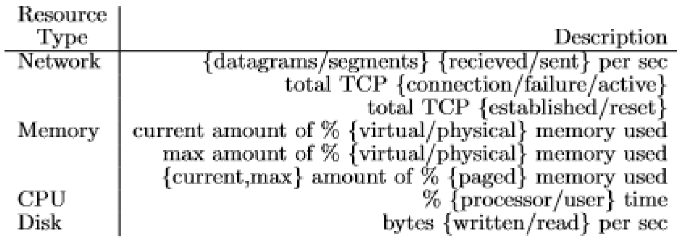
```
Algorithm 1: ContextPrioritizer
Input  : Prioritization Request for App$_i$
         Len$_i$: Length of required TestSeq$_i$
Output: TestSeq$_i$ for App$_i$
AppSimSet ⟵ {}                          /* compute AppSimSet */
For ∀App$_j$ ∈ AppHist
    ResSimSet ⟵ {}
    For ∀Res$_k$ ∈ ResSet
        If KS.ResTestIsTrue(App$_i$, App$_j$,Res$_k$)
            ResSimSet ⟵ Res$_k$
        EndIf
    End
    If |ResSimSet| / |ResSet| $\geq$ KS$_{thres}$
        AppSimSet ⟵ App$_j$
    EndIf
End
ResCrash ⟵ {}                           /* compute ResCrash */
For ∀App$_k$ ∈ AppSimSet
    For ∀Crash$_l$ ∈ AppHist[App$_k$]
        ResCatSet ⟵ ResCat(Crash$_l$, CAT$_{thres}$)
        For ∀Res$_j$ ∈ ResSet
            If Res$_j$ ∈ ResCatSet
                ResCrash[Res$_j$] ⟵ Crash$_l$
            EndIf
        End
    End
End
TestSeq$_i$ ⟵ {}    /* select Test Case Sequence */
While |TestSeq$_i$| $\leq$ Len$_i$
    CrashVote ⟵ {}              /* voting - Tier One */
    For ∀ Res$_j$ ∈ ResCrash
        Temp ⟵ TopN(ResCrash[Res$_j$],RES$_{thres}$)
        CrashVote ⟵ Temp
        ResCrash[Res$_j$] − Temp
    End
    TestSeq$_i$ ⟵ TopN(CrashVote,1)  /* Tier Two */
End
```

Figure 4.2 ContextPrioritizer Algorithm



Context Dimension	Raw Entries	Description	Source
Network	9,500+	Cellular in Locations Wi-Fi Hotspot Cellular Operators	Open Signal
Platform	220/23 (W8/WP8)	CPU Utilization Ranges Memory Ranges	Watson
Sensor	300	Locations	FourSquare
Others	49	Transitions, Extreme Cases	Hand-picked

TABLE I. : CONTEXT LIBRARY



Resource Type	Description

Network	{datagrams/segments} {recieved/sent} per sec
	total TCP {connection/failure/active}
	total TCP {established/reset}

Memory	current amount of % {virtual/physical} memory used max amount of % {virtual/physical} memory used {current,max} amount of % {paged} memory used
CPU	% {processor/user} time
Disk	bytes {written/read} per sec

TABLE II. SYSTEM RESOURCES

Algorithm 1 describe that ContextPrioritizer has the three main operation which are AppSimSet computing, ResCrash computing and test case sequence selection

Context fuzzing is complicated by the fact that a given app will likely only be sensitive to a fraction of all test cases in ContextLib. To choose important test cases for any particular test app. ContextPrioritizer identifying correlated system resource usage metrics as a deeper means to understand the relationship between two apps. Because two apps that have correlated resource usage (such as memory, CPU, network) are likely to have shared sensitivity to similar context-based test cases.

Compute AppSimSet

KS.ResTestIsTrue() is to compare similarity between a new test app, and a previously tested app This is done for each system resource metric while both apps were exposed to the same context. ResSet is a collection of j resource metrics (listed in Table 2). For each resource metric, the pairwise tests are performed between the current test app and all prior test apps in AppHist. In many cases, ContextPrioritizer is able to compare the same pair of apps and same resource metrics more than once (for example, under different test cases). For the prior test app to be included in AppSimSet, it must pass the statistical comparison test a certain percentage of times (KSthres). We empirically set KSthres = 65%, as it is not too loose to differentiate cases such as image-centric and video-centric news apps. ContextPrioritizer uses the above described pairwise comparison multiple times to construct a unique AppSimSet for each new test app.

Compute ResCrash

As many crashes are caused by abnormal resource usage such as excessive memory, crashes can be tied to resource metrics. The motivation for categorization of test case history is two-fold. First, it allows non-context related crashes(such as division by zero) to be ignored during prioritization. Second, the frequency of crashes fluctuates significantly between resource categories like (e.g., network-related crashes are much more common that hardware-related ones). Procedure ResCat() tied the crash to the resource metrics

Select Test Case Sequence

ContextPrioritizer uses a two-tiered voting process to arrive at a sequence of test cases (TestSeqi) to be applied to the test app (Appi). The tiered process ensures no single resource category dominates test case selection.

At each tier the voting setup is the same. Each time a test case results in a crash is treated as a vote, any crash that is categorized as being non-context related is ignored. TopN() compute The order of test cases determined by the popularity in terms of weighted crashes observed in past test apps contained within AppSimSet

The differences between the two tiers are in the pools of context-related crashes considered. For each past test app, the first tier looks at which test case crashes happened for each resource metric (ResCrash). Then, the some top popular test cases is picked for each app, and aggregated at the second tier. A single test case is chosen, but this process is repeated for a test case sequence.

V.CONCLUSION

In this paper, we have compared Android automation testing tool. The test case generation strategy of special testing condition strategy is given. According to the method, we can see that it is possible to generate test case automatically by testing tools. However the method is not fit for any condition, especially to model-based testing, because there will be many useless test case is generated. It is the future work to find a more general testing strategy such as half-automation testing tool.

REFERENCES

[1] GUI Wiki: available at https://en.wikipedia.org/wiki/Graphical_user_interface

[2] GUI testing Wiki: available at https://en.wikipedia.org/wiki/Graphical_user_interface_testing

[3] Robotium Wiki: available at https://github.com/robotiumtech/robotium/wiki

[4] MonkeyRunner: https://developer.android.com/studio/test/monkeyrunner/index.html

[5] MonkeyTalk : https://intensetesting.wordpress.com/2014/03/14/monkeytalk-tool-and-mobile-testing/

[6] UIAutomator : https://developer.android.com/topic/libraries/testing-support-library/index.html

[7] Appium: http: //appium.io/

[8] Selendroid : http://selendroid.io/

[9] Model-based testing Wiki: available at https://en.wikipedia.org/wiki/Model-based_testing#cite_note-6

[10] Android GUI Ripper Wiki: available at http://wpage.unina.it/ptramont/GUIRipperWiki.htm

[11] Memon, Atif M., Ishan Banerjee, and Adithya Nagarajan. "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing." WCRE. Vol. 3. 2003.

[12] Amalfitano, Domenico, et al. "Using GUI ripping for automated testing of Android applications." Proceedings of the 27th IEEE/ACM

International Conference on Automated Software Engineering. ACM, 2012.

[13] Yang, Wei, Mukul R. Prasad, and Tao Xie. "A grey-box approach for automated GUI-model generation of mobile applications." International Conference on Fundamental Approaches to Software Engineering. Springer Berlin Heidelberg, 2013.

[14] Liang, Chieh-Jan Mike, et al. "Caiipa: automated large-scale mobile app testing through contextual fuzzing." Proceedings of the 20th annual international conference on Mobile computing and networking. ACM, 2014.

[15] Liang, Chieh-Jan Mike, et al. "Contextual fuzzing: automated mobile app testing under dynamic device and environment conditions." Microsoft. Microsoft, nd Web (2013).