EMITTER International Journal of Engineering Technology
Vol. 10, No. 1, June 2022, pp. 63~82
DOI : 10.24003/emitter.v10i1.628

Automating Test Case Generation for Android Applications using Model-based Testing

Usman Habib Khan[1], Muhammad Naeem Ahmed Khan[2], Aamir Mehmood Mirza[3], Muhammad Akram*[4], Shariqa Fakhar[5], Shumaila Hussain[6], Irfan Ahmed Magsi[7], Raja Asif Wagan[8]

[1,2] Independent Research Scholar, Islamabad, Pakistan,
[3,4,7,8] Balochistan University of Information Technology, Engineering and Management Sciences, Quetta, Pakistan
[5,6] Sardar Bahadur Khan Women's University, Quetta, Pakistan
Corresponding Author: akram.khan@buitms.edu.pk

Received January 6, 2022; Revised February 10, 2022; Accepted March 14, 2022

Abstract

Testing of mobile applications (apps) has its quirks as numerous events are required to be tested. Mobile apps testing, being an evolving domain, carries certain challenges that should be accounted for in the overall testing process. Since smartphone apps are moderate in size so we consider that model-based testing (MBT) using state machines and statecharts could be a promising option for ensuring maximum coverage and completeness of test cases. Using model-based testing approach, we can automate the tedious phase of test case generation, which not only saves time of the overall testing process but also minimizes defects and ensures maximum test case coverage and completeness. In this paper, we explore and model the most critical modules of the mobile app for generating test cases to ascertain the efficiency and impact of using model-based testing. Test cases for the targeted model of the application under test were generated on a real device. The experimental results indicate that our framework reduced the time required to execute all the generated test cases by 50%. Experimental setup and results are reported herein.

Keywords: Android app testing, Model-based testing, Functional testing, Smartphone app testing, Test case generation.

1. INTRODUCTION

Software testing is a nontrivial phase in the software development process and 50% of the total effort is consumed during this phase [1]. Software testing is the process of finding defects in the software. In manual testing, the tester explores the core functionality of the software and develops test cases accordingly. Automation of the testing process not only reduces the time and efforts of software testers but also ensures the correctness of the application. Software tools specifically designed to execute test scripts and compare the generated output with the expected output are used in the automated testing process. However, automated testing can computerize only some steps of the

overall testing process. Hence, human intervention is always required to perform automated testing.

Model Based Testing (MBT) is performed by using a model of the app that depicts the functional behavior of the software. A model describing software under test (SUT) is an abstract representation that describes how the software works. Test cases derived from a model are functional tests on the same level of abstraction as that of the model. Test cases are collectively known as an abstract test suite. When the model of an application is traversed end to end, it provides a potential test case of a specific module or branch of the application. Ideally, a model is traversed completely to ensure that the application is fully explored, and no path or branch of software is ignored during modeling of the application under test (AUT). An abstract test suite cannot be directly executed as abstract test cases require modifications to transform these into executable scripts. Executed scripts generate results of the testing process which can be analyzed manually or using the third-party tool as per requirements of the test plan.

Smartphone apps are getting popular – from entertainment to games and from utility to mission-critical systems, there is a huge pool of smartphone apps. The mission-critical smartphone apps are in the areas of health, banking, e-business/commerce, and transport management sectors, etc. Mobile apps are greatly exposed to users so there are chances of use case deviations and these apps have their quirks for testing as a high number of events are needed to be tested [2]. Security is an important aspect of smartphone apps [3]. Like web apps, mobile apps also communicate with live servers and cloud so the need for security, performance, and stress testing arises for the mobile platform [4]. Smartphone app testing carries certain challenges [5] which mainly pertain to the interpretability of mobile platforms with web, third party systems, and cloud. Other challenges relate to limited memory/space, UI/display, battery life, and storage [6]. Smartphones carry several features that general-purpose IT products normally do not possess like GPS, Bluetooth, and accelerometer. These features require additional considerations and efforts when testing mobile apps. Mobile apps are getting more complex in nature, bigger in size, and available for a multi-user environment, therefore, manual testing of such apps is no more viable.

There are several test automation techniques like MBT [7], Record and Replay [8], GUI-based App Testing [3], Robotic Testing [9], Model Checking [10], Targeted Testing and Conformance Testing [11], etc. Automation can be achieved at different stages of testing like test case generation, test data generation [12], test case execution, model generation [13] from requirement specifications, and code generation from the model. For MBT, unified modeling language (UML) diagrams are used to illustrate the functionality of SUT. UML diagrams are designed to portray specific angles or views of the system. MBT facilitates automated testing by making an abstract model of the SUT which can be translated into a tool readable format to facilitate test case generation of the SUT.

MBT has been successfully employed to test conventional software [14], [15]; therefore, we consider that the same approach can be exploited to automate the testing process for mobile apps. For automatic generation of test cases, activity, sequence, and statechart diagrams are ordinarily used to identify the functional behavior of the system. Yet there is no tool available to generate test cases for an AUT automatically. The existing techniques endure some drawbacks which serve as research impetus to overcome these issues. A major issue in adapting the MBT approach is the state explosion [14], [16]–[18] i.e., it becomes difficult to manage model space when the size of the app increases.

The focus of this study is to generate automated test cases of smartphone apps particularly for the Android platform using model-based approach. Android is the leading smartphone operating system (OS) followed by iOS. Also, it is a Java-based OS and the majority of mobile apps have been developed in Java language. For model creation, we use state machine diagrams or statecharts to model application behavior, functionality, and user interaction. State machine and activity diagrams are widely used modeling paradigms [13]. We use state machine or statechart diagrams for AUT modeling as it is a matured paradigm in the UML domain. Most of the smartphone apps are state-based i.e., these apps change states according to the input provided by the user. Also, statechart diagrams are easy to understand, analyze, and model.

This paper is organized into five sections and this section being the first section describes an introduction of the different testing approaches. The second section presents a critical review of the existing testing techniques for smartphone apps followed by the key challenges, motivation, and problem statement. The originality of work is described in the third section. We present our proposed framework for automated test case generation in the fourth section. In the next section, we provide details of the experimental setup and experimental results. Finally, we conclude in the last section.

2. RELATED WORKS

Mobile apps are event-driven systems that act according to the generated events, gestures, and context of operations. Several offline testing techniques are reported in the literature such as the model-based approach based on a UML activity diagram for testing context-aware smartphone apps [5]. Amalfitano et al. [1] identified different domains for mobile application testing e.g., the interface model and call graph model. These models generate test cases with the help of evolutionary techniques. Tong and Yan [2] proposed a hybrid technique to improve malware detection rate in Android mobile apps. It is a generic approach that caters for both static and dynamic approaches to detect malware in Android apps. Static analysis is done on the code and dynamic analysis is a runtime live analysis of AUT. This technique firstly performs dynamic analysis to collect system calls of an app to form patterns. This approach is based on evaluating the difference in malware and benign

apps in runtime system calls. For GUI testing, several techniques have been proposed in the literature including *Histogram, SURF,* and *Template matching*. Lin et al. [3] proposed a GUI based automated testing tool SPAG-C which uses a record-replay technique to perform GUI testing on Android devices. However, applications with non-deterministic GUI cannot be tested by SPAG-C as the AUT screen keeps changing for video or game apps. This approach works on image capturing mechanism where a camera is used to capture screenshots of all UI could be affected by external factors like light, exposure, etc., so a controlled environment is required to execute this tool.

FSM and statechart diagrams are formal yet powerful tools to model a SUT. Rauf et al. [4] presents a critical analysis of MBT techniques and states that UML diagrams are mature artifacts of software design to derive test cases. Activity and sequence diagrams are the focus of the modern researchers in MBT [14]. A drawback of sequence and activity diagram regarding mobile systems is the state skipping problem. The activity diagram exhibits the complete behavior of the system and is a modeling tool for the derivation of functional test cases. This offers new prospects to develop hybrid models that embed the merits of different modeling approaches like the completeness of FSM, coverage of statecharts, the concreteness of activity and class diagrams, and conciseness of sequence diagram. It is reported that the FSM/statechart is suitable to test mobile apps through MBT. UML design artifacts can help achieve significant state coverage, transition coverage, and transition-pair coverage to fulfill the boundary testing criterion. Swain et al. [7] used the statechart diagram to automatically generate test cases for object-oriented software. Whereas, Amalfitano et al. [9] presented a test automation technique and tool for mobile platform called *MobiGuitar* which is based on extracting test cases for the SUT by using its GUI widget. It creates a scalable state machine model using event-based test coverage criteria, which automatically creates test cases for SUT.

Manual testing has its own merits like completeness, device independence, and realism. But sometimes, especially for regression testing, it becomes a time consuming and tedious task. Mao et al. [10] developed a tool called *Axis* for automated black box testing of mobile apps. Idea is to automate the testing process that requires less human involvement. However, mobile apps contain several smart gestures such as the compass, swipe gestures, tapping, etc. that need to be carried out physically to test their correctness. Espada et al. [11] presented an MBT approach for Android apps to generate automated test cases. The approach is based on user interactions by modeling AUT behavior by composing state machines and then exhaustively exploring the acquired model using a model checking tool *SPIN*. All of the user behaviors eventually correspond to potential test cases. The model is obtained in a semi-automated manner using the *UIAutomatorViewer* tool. The generated test cases are in XML format which is converted into a specification language processing tool *Promela* to generate a model by analyzing system specifications. SPIN uses this model for generating test cases.

Azim and Neamtiu [20] present an approach that systematically explores Android apps without analyzing its source code. The proposed approach traverses AUT in terms of usability and explores it in a depth-first manner to ensure full coverage of the app. Jing et al. [17] present a conformance testing tool that systematically generates test cases from requirement specifications and performs a rigorous conformance testing. Farto et al. [21] analyze MBT in modeling, concretization, and execution of automated test cases of mobile apps. The study used the Event Sequence Graph to design the test model and used the Robotium framework to implement the test cases. Guiterezz et al. [22] present a model-driven approach for test case generation from the functional requirements. The proposed approach uses structural model coverage criteria to generate test cases based on transitions among use cases, particularly at variation points. As test cases are generated from functional requirements, so requirements should be crystal clear and non-redundant. To achieve standardization and precision while generating a UML model from the functional requirements, it would be more appropriate to employ Natural Language Processing techniques on the requirement document. Lamancha et al. [23] mechanize the process of test oracle by automatically producing expected output and comparing it with the actual output. The study uses state machine diagrams to generate a model of the (SUT) and model-driven testing. The study uses UML activity, sequence, and state machine diagrams to elaborate test oracles. Though state machine diagrams are used for model generation, but for a large-sized application or complex system, the state machine can be problematic as these can grow enormously in size.

GUI based testing is common for mobile app testing but sometimes it cannot be fully applied to those apps where UI heavily depends on user interaction e.g., multimedia apps. Ramler et al. [24] applied code level testing to test GUI-rich apps. This study is useful for testing UI gestures (e.g., mouse clicks, drag & drop, double click, swipe, and holding) of rich interactive systems. The generated test cases were executed through a testing framework called *jUnit*. Dev et al. [15] proposed a GUI testing approach for smartphone apps based on online MBT and offline MBT. Offline MBT is useful if SUT is well-conceived i.e., there is no ambiguity in requirements. Online MBT can record test cases continuously and helps in parallel testing as it can incorporate live changes in SUT behavior. A drawback of MBT is that it requires special skills for modeling, analyzing requirements, and scriptwriting. Espada et al. [25] presented a tool to identify abnormalities in mobile apps. The tool firstly catches user interactions with the app and then models those interactions using statecharts. The model is then translated into executable XML and Java scripts. Salva and Zafimiharisoa [26] presented an Android app security testing tool to detect intent-based vulnerabilities on Android components. Tao and Gao [27] presented the concept and design of mobile testing as a service that comprises eight different test features in which five are for cloud and three for generic mobile app testing.

State explosion is a key issue with MBT techniques as state-space increases substantially while modeling large size software systems. Arcaini and Gargintini [28] proposed a test case generation technique for distributed systems using Abstract State Machine (ASM) which is an extended version of FSM. A major drawback of using ASMs for large size software is the exploration of state space. Since distributed systems are huge so SUT is divided into smaller components and MBT is applied to these smaller chunks. This keeps the state space size under control.

Software is growing in terms of size, complexity, and many other aspects which make it difficult to test the SUT thoroughly. Generating feasible test paths for an app that satisfies the overall functionality of AUT is a sound technique to test mobile apps if test paths are minimal. Ahmad et al. [18] present an adaptation model for testing mobile apps through refactoring so that transitions and paths of the app are minimized. Code refactoring minimizes code size without affecting the functionality of the app and compromising the test case coverage. Yang et al. [29] proposed a grey-box approach to automatically reverse engineer GUI-models of mobile apps by using a testing tool called *Robotium*. The study performs a static analysis of the source code to extract the set of user actions supported by each GUI widget in the app. Morgado et al. [30] also proposed a technique to test mobile apps using reverse engineering through the identification of behavioral patterns of its GUI. Gudmundsson et al. [16] applied MBT to test mobile apps using a command-line tool *Kelevra* with *Appium*.

Test case preparation for a large and complex app is a tedious, time-consuming, and error-prone process due to a lot of human interventions. Sometimes various testing tasks in the manual testing are repetitive, thus making it a boring and tedious job for the testers. Automated testing helps overcome human error and makes the testing process reliable [31]. However, no tool is yet capable of full-fledged automatic testing of smartphone apps. The unique features of smartphones such as constantly changing context, interoperability, the unreliability of wireless networks, and limited bandwidth pose serious challenges for testing mobile apps. The inherent difficulties of testing mobile apps due to their peculiar nature served as a motivation for us to formulate an operational framework that will serve as a step forward to augment empirical investigations in the domain of mobile app testing. We propose systematically exploring AUT so that essential functions of the app get priority and path coverage is done based on the most frequently used app paths. Improving the efficiency of mobile apps testing through automation is an auxiliary motivation for our study.

3. ORIGINALITY

With the growing numbers and diversity of mobile device users, new testing techniques must be studied so that defects can be avoided, and the quality of mobile applications improved. The testing processes of mobile applications to improve the design and generation of test cases as well as the

availability of test automation tools should be given special consideration. In this context, model-based testing (MBT) is among the techniques which can be used to ensure software quality. MBT permits the automatic generation of test cases by a model based on the expected behavior of AUT. MBT is an approach that has several advantages reported in the literature, such as automatic test case generation, fault detection efficiency, and time and cost reduction for testing.

From the abridged literature review provided in Section 2, we observed that various modeling notations such as FSM, UML activity, and sequence diagrams have been used for test case generation. A novelty of our proposed framework is that we have used statecharts for developing test models. And to the best of our knowledge, no other research study has used statechart diagrams to generate test cases for mobile applications.

The key reason to choose statechart diagrams is that each mobile screen rendering represents a state in which an app is running. By considering every screen rendering in AUT as a state, we use statechart to model the app functionality. Statechart diagrams are a mature paradigm and have been used to model large and complex software systems so it also serves as a reliable source to model mobile app interaction and behavior. Furthermore, we have transformed abstract test cases into concrete test cases and executed these scripts and generate the test result reports in the *TestNG* tool*. Appium* is a popular open source tool for mobile app testing and we used it for automation of the testing process.

4. SYSTEM DESIGN

In this section, we present our proposed framework for automated test case generation of mobile apps using model-based approach. Figure 1 illustrates the functioning of the proposed framework.

We use model-based approach to automate the test case generation process. We first track the user behavior with AUT and transform all the possible use cases into abstract test cases. We use statechart diagrams for app modeling and consider every possible user interaction as a potential test case. Initial statecharts are obtained through the interaction of the user with the AUT. These interactions are then converted into tool readable format with the help of the *Yakindu* code generator tool. Yakindu supports the UML design and development environment.

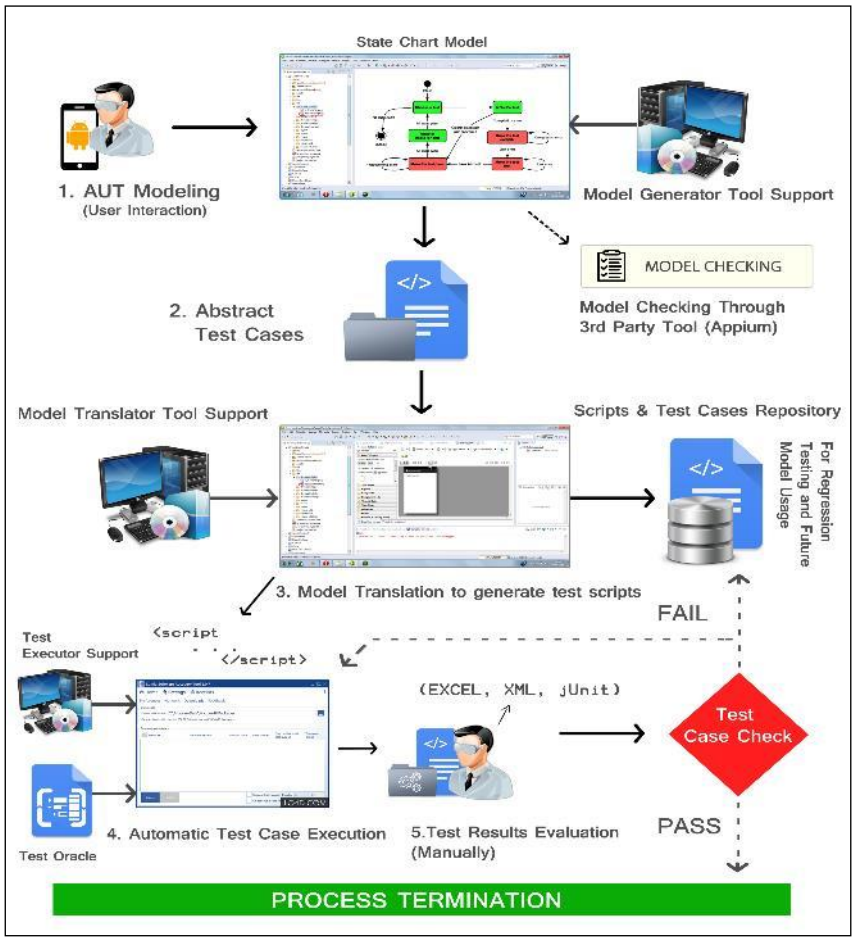

Figure 1**: Graphical representation of the proposed framework**

These models are then translated into a readable format for generating test suites by the model checking tool *Appium*. After the transformation of raw models into an executable test case, Test Executer is run to obtain XML or Java-based test scripts. For script execution, we integrate *Eclipse* extension to Test Executor to facilitate the execution of AUT. *Eclipse and TestNG* are Java-powered frameworks that offer test script execution. We use Outlook, an email client app for Android, to perform experiments. We validate the results from a state coverage perspective and the efficiency of the framework is assessed based on the number of valid test cases generated per unit time. Another validation parameter that we contemplate is effectiveness where we ensure that the generated test cases pertain to authentic functional paths of AUT and are unique and non-repetitive**.** A phase-wise detailed description of our proposed model is provided below.

4.1. AUT Modeling

Our framework is based on a model-based approach where a tester initially analyzes the functional working of the app by interacting with it in a user mode. For this purpose, we need a modeling paradigm to model the

behavior of the AUT. We use statechart diagrams for AUT modeling due to its previously defined merits. The conceived model of AUT functionality is manually analyzed by the tester to verify that the model is correct. A statechart modeling tool *Yakindu Suite* offers validation parameters of declaring interfaces, events, variables, and I/O mechanisms to ensure model correctness. We take Microsoft Outlook mobile app as our AUT and model it by using *Yakindu Suite* which allows us to model UML statechart diagrams in an integrated environment with *Eclipse*. An added advantage of the *Yakindu Suite* is that it allows code generation from the source model into different languages. Though the generated code is neither complete nor executable, yet it exhibits a structural view of the model of AUT. From the model of the AUT, it creates two code files; one is for its use with the ".sgen" extension and the other is a code file in Java programming language. Generating code in *Yakindu Suite* is not straightforward. We need to include libraries that support the functionality of code generation from the statechart model. The integration of the *Yakindu Suite* with *Eclipse* is the next step in our experimental setup so that we can create the model and generate code.

4.2. Abstract Test Cases Development

In the next step, we get abstract test cases in the form of model paths. Each module of the application is modeled separately which enhances the completeness and uniqueness of the model. For this purpose, we used a depth-first search (DFS) algorithm to explore the application model systematically. DFS model exploration takes the user from end-to-end functionality elements e.g., from the start of the app to any of its exit points, we can have a full path to test. DFS also helps to reduce the number of repeating and overlapping test paths.

4.3. Test Scripts Generation

These test paths or abstract test cases are then translated into a working code by using the *Yakindu model translator* plugin so that we can execute these test cases. This plugin generates a generalized code against the input model. It is important to mention that the acquired code is not a mature script that can be directly executed. In other words, the translated scripts are like the architectural view of the model which requires dynamics of user inputs and interactions like swipes, taps, double taps, hold and drag, etc. By using proper user inputs and interaction, we convert abstract test cases into concrete test cases with the help of a model translator.

4.4. Test Scripts Execution

As most of the smartphone apps are state-based so using statechart is a good choice to visualize the functionality of the AUT to understand what the app does, whether it is a game or a utility app etc. The overall functionality of an app is ascertained by either consulting requirements document or by playing around the app. Modeling is a crucial part of our framework as it

follows MBT. We model the app behavior using statecharts to generate abstract test code. Hence, the tester must have expertise in the tools that support code generation from a UML-based model. We used the *Yakindu* statechart modeling tool which allows code generation from a model through its code generating plug-in. These models are then translated into Java code which exhibits the general structure of the module or path. These scripts are in crude form and need to be manually edited so that these can be executed. We edit these scripts as per our requirements in *Eclipse* with *Appium*. We assert the changes and using these assertions we trigger the app UI including buttons, drop-down menus, and text boxes, etc. After editing the scripts manually, we execute these scripts and generate the test result reports in the *TestNG* tool*. Appium* is a popular open source tool for mobile app testing and we used it for automation of the testing process.

4.5. Test Results Evaluation

Evaluation of the test results is again a manual process which is done by the tester.

5. EXPERIMENT AND ANALYSIS

To implement the framework, we model the working behavior of the app using state machine/chart diagrams. The model contains all the pertinent interactions of the user with AUT. For instance, a user needs to have a valid email address before send/receive an email. For a clear comprehension of the app behavior, we create a separate model for each module so that each path exhibits a unique functional workflow of the app. The statechart model needs to be redesigned using specialized tools so that it could be translated into abstract code. The abstract code is subsequently edited manually to execute it on a third-party tool. We chose the *Yakindu* modeling tool to model the authentication module by interacting with GUI elements of MS-Outlook. The authentication module contains two input fields: username and password. Authentication modules are generally based on AND operator i.e., preconditions for both the input fields should be true to get access to the app. On tapping the login button, the app takes the user to an interface to enter the username. If the username is already registered with the app, this module takes the user to the next step to enter the password. If both the inputs are matched, then this module provides user access to use the app or otherwise displays the message “incorrect username or password.”

On the "Enter Username & Password" state, we pause the state to 1000 milliseconds so that the user can enter username and password without any hassle. *Yakindu* allows a minimum of 1000 ms waits for a state. Since it is a state-based paradigm, we set a timespan to hold a state and after the lapse of that time, the state can revert to the previous state if no user interaction happens. *Yakindu* works on the principle of input and output pair. We need to create interfaces for every state and declare events and variables to be used within the state transitions.

When the user provides credentials of his/her email account, the state of the model is changed, and the app module goes into the state of the actual authentication of credentials. At this stage, the app checks whether the supplied credentials are valid or not. If these credentials are valid then the module takes the user to the HOME page of the app or otherwise displays an error message. Figure 2 shows a general statechart model of the UI of the authentication module of AUT.

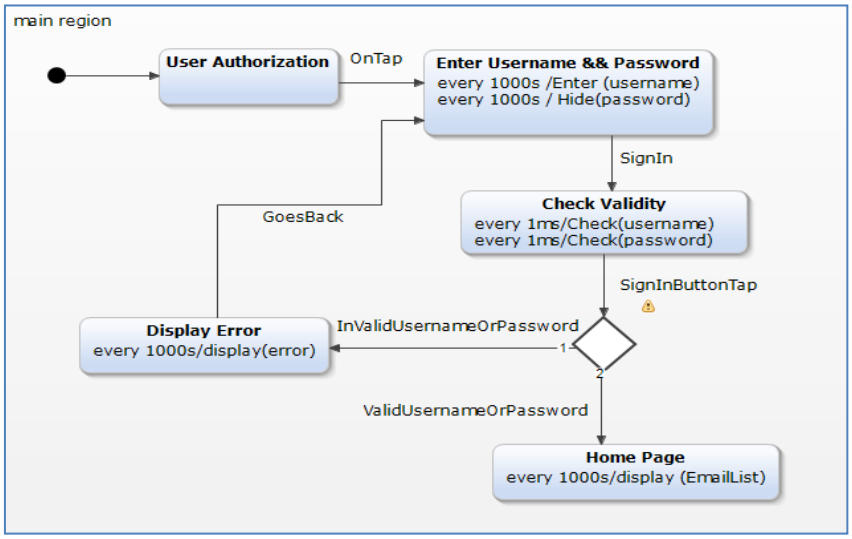

Figure 2**: Statechart model of the authentication module**

In general, the possible test combinations for 2 input fields are 7 satisfying $\mathbf{n^3 - 1}$ formula. Here, the input field **n** is 2 corresponding to username and password. Using this formula, we get the following 7 test cases about username and password.

TC-1: Correct username and correct password
TC-2: Incorrect username with the correct password
TC-3: Correct username with an incorrect password
TC-4: Incorrect username and incorrect password
TC-5: Empty username with non-empty password
TC-6: Non-empty username with an empty password
TC-7: Empty username and empty password

By grouping test cases into valid, invalid, and missing cases, there is only one app path TC-1 where app control is smoothly transferred to the next state. This blissful path comes under the valid case category. Next three cases (TC-2, TC-3, and TC-4) where either username or password is invalid pertain to the invalid scenario category. The last three test cases TC-5, TC-6, and TC-7 pertain to incomplete credentials where at least one of the input fields is missing. There is an important facet to describe; since we are exploiting GUI elements of the app, so it is not possible to trigger more than one tapping or clicking event on the screen. In *Eclipse*, we access GUI elements from top to bottom and *UiAutomator* can only trigger one input or UI element at a time. So, we run the

script sequentially so that every input/GUI element can be accessed one at a time. In this way, we get a complete test suite of a test scenario. Some other cases need to be verified e.g., the checkbox button for the "Remember Me" option.

All the test cases are executed through assertion made on the input fields and login button. We have an ID for every UI element like input fields, pushbuttons, menus, radio buttons, checkboxes, etc. These IDs are accessed by capturing screen snapshots through *UiAutomator* and then we declare these in the script generated by the *Yakindu* code generator plugin. We apply assertions on these UI elements and access these through ID. Once the required UI element is successfully called, we enter data values to determine the result of the assertion made on the UI elements. The obtained result is the ultimate result of the test case. After modeling app behavior, we transform these test paths into abstract test cases that hold information on the path transitions. These test cases are an architectural view of the test path and need adjustments in terms of user inputs and variants. *Yakindu* provides plugins for different programming languages but we generated abstract test cases in Java to transform our model into scripts

To run Outlook's *APK* file in real time, we need to connect the smartphone with the *Appium* tool as it acts as a server for hosting the APK file of the AUT. For this purpose, we define all the attributes or capabilities of the AUT *APK.* For example, the *APK* name, web-server name, ports to be opened, Android OS version, absolute path of the *APK* file, application package source, and the main activity through which we start capturing GUI elements. Next, we declare the remote *WebDriver* which is used to access the port and URL of the remote server. The generated script is quite generic and we shall refine these scripts before execution. It is necessary to define all the possible transitions that can lead to new states. We use assertions to call the input elements of GUI like input fields, radio buttons, checkboxes, etc. After modification and declaring all the required variables and their interactions, we transform the abstract test case into a concrete or executable test case.

Now, this script is in its complete and final form and can be run on any external environment which supports Java. We execute this script on an allied tool *Eclipse*. We set parameters for the execution of test cases which determine whether the test is a "PASS" or "FAIL". However, we are not interested in determining whether the test gets PASS or FAIL as our major focus is to generate test cases only. But for obtaining the results, we generated test reports against every test case. We applied assertion on every input element and access it through its *Xpath*, *className,* or *ElementID*. We access these elements through the UiAutomator screen capture function which shows all the available UI elements on the screen. *Appium* sequentially executes the script and after script execution, it transfers control back to *Eclipse* to display whether the test case was a pass or fail. This report is generated by the *TestNG* tool that we integrated with *Eclipse* for reporting purposes. *TestNG* is a testing framework and requires JDK 7 or above to function properly. *TestNG* supports

easy annotations, log generation, and allows producing HTML and XML reports.

The next module is "Send/Receive Email" where we first compose an email. Firstly, we get the GUI elementID of the "compose" icon or button. Once the GUI element is accessed, we enter the receiver's valid email address. Email clients allow the user to enter up to 25 distinct email addresses in one go making it a potential test case. After entering the recipient's email address, the next input box is to enter the CC email address (es). This step is optional and can be left empty. The next field is to enter the subject of the email message, but it can also be empty. The next step of composing an email is to enter the content of the message. We can verify whether the text editor allows typing in the text/content making it a potential test case. We can also attach files with the email message. On tapping the attachment icon, we should have the ability to attach a file of a maximum of 25MB size from the mobile file directory. Tapping on the "Send" button should send an email message to the recipient(s). The sent folder should contain an entry if the email is sent to the receivers. The following test cases for this module can be generated:

- To check the recipient's address is present or not
- To check the recipient's address count is not more than 25
- To check that editor allows composing the message
- To check that an attachment can be attached
- To check that size of attachment should not be more than 25 MB
- To check that the Send button should deliver the message

The next module of the app is "View email" where users can view Inbox containing the received messages. On tapping a message, the user can view the sender's name, subject of the email, content of the message, attachment, etc. Besides, the user can also reply, forward, delete, print, and move the received email. From an implementation point of view, these potential test cases are triggered by the same procedure of acquiring element ID using the capturing screen in the *UiAutomator* tool and applying assertions over it. The following test cases for this module can be generated:

- Can a user view the received email?
- Can a user view the sender's email address/name?
- Can a user view the subject of the email message? (optional)*
- Can a user view the content of the message? (optional)*
- Can a user view/download the attachment with the message? (optional)*
- Can a user delete the message?
- Can a user recover the email message?
- Can a user forward the message?
- Can a user reply to the sender?
- Can a user print the email message?
- Can a user move the message to another folder?
- Can a user mark the email message for future use?

Another important function of the application is that the user should be able to logout/sign-out or remove his/her account from the app. The mobile email app does not have any UI element for logout/sign-out in the form of a button or icon. Users can only sign-out only by removing his/her account from the app. If a user taps on “Remove Account” in the app's main menu, the user session gets expired and the login screen appears. Thus, this feature leads to a potential test case.

Applying MBT on smartphone app testing can produce promising results in the controlled environment. To verify the efficiency of our framework, we compare our framework with manual testing and an automated MBT tool. For this purpose, a domain specialist manually explored the app in play around manner and generated test cases and we compared results obtained by the domain experts with the results obtained through our framework. Table 1 shows a comparison of both approaches. We model the app and traverse it in a targeted fashion so that test cases are unique, complete, and non-repetitive. This ultimately boosts automation in terms of resource utilization i.e. cost and time. We targeted those modules of the app that were critical and important from a functional point of view.

Table 1. Total effort size and steps of the proposed framework.

Proposed Framework			
Sr	**Tasks/Steps**	**Duration**	
1	Understanding the functionality of AUT	▪ Consulting documentation	08 Hrs
		▪ Preparing notes and briefs/Understanding AUT	08 Hrs
		▪ Going through App/ General use	08 Hrs
2	Research for the best suitable choice for tools for experimental setup	▪ Research for tools selection	24 Hrs
		▪ Compatibility checks	16 Hrs
3	AUT Modeling using statecharts	▪ AUT Modeling	40 Hrs
4	Experimental setup	▪ Tools Installation	04 Hrs
		▪ Removing Dependencies	12 Hrs
		▪ Server Configurations	06 Hrs
		▪ Smartphone Integration	02 Hrs
5	Model translation to code (Model to structural code generation)	▪ Model to code translation	01 Hrs
6	Concrete script generation (Code editing to make abstract scripts to concrete scripts)	▪ Manual code editing to make scripts executable	40 Hrs
7	**Script Execution**	▪ **Script (Test cases) execution**	**03 Min**
8	**Report Generation**	▪ **Report generation of executed scripts**	**02 Min**
Total effort size		**169 Hrs (approx.)**	

Our framework generates the necessary test cases for functional testing by eliminating repetitive and least critical test scenarios. Table 1 shows the total effort size of our approach by describing the breakdown of every task/step. The proposed approach took almost 169 hours for testing an app for the very first time; whereas, the manual approach took 42 hours to test the same app. Our approach takes additional time for the initial steps to develop the model, tools installation, and resolving constraints and dependencies. However, these steps are just a one-time activity. Our approach is faster and efficient as compared to the manual approach if we consider the execution time that our approach takes for the same AUT. The manual approach took 10 hours for a complete cycle and we must go through the same procedure and steps again and again to test the AUT. But we just have to run the script against AUT, and test reports can be generated within 5 minutes using our framework which is 120 times faster in terms of test cases execution time and is best suited for regression testing. In other words, our approach becomes faster in the second round onward. As the overall testing process involves various testing cycles, therefore, our framework not only automates the testing process but saves testing time significantly.

Steps 1 through 6 in Table 1 are the tasks that are just one-time tasks and are performed once in the overall testing process. If there will be any change or tweak in the app, then we just have to modify the already written scripts and do not have to again perform complete testing of AUT. The same app was tested manually and Table 2 shows results in terms of effort size.

Table 2. Total effort size and steps in the manual approach.

Manual Testing Approach			
Sr	**Tasks/Steps**	**Duration**	
1	Understanding the functionality of AUT	▪ Consulting documentation	08 Hrs
		▪ Comprehending AUT	08 Hrs
		▪ Going through App/ General use	08 Hrs
2	Executing test cases for AUT	▪ Test cases preparation	08Hrs
		▪ Test cases execution	08 Hrs
3	Reporting	▪ Analyzing test results and reporting the results of executed test cases.	02 Hrs
		Total effort size	**42 Hrs (approx.)**

There is always a chance of human error in manual testing. Automation of the testing process enhances product quality and takes lesser time by streamlining the entire testing process. Table 3 shows a comparison of both approaches in terms of effort size.

Table 3. Comparison of manual approach with the proposed framework in terms of effort size.

Proposed Framework			Manual Approach	
Sr	**Task**	**Duration**	**Task**	**Duration**
1	Script (test cases) execution	03 Minutes	Test cases execution for AUT	08 Hours
2	The analysis report of the executed scripts	02 Minutes	Report generation of executed test cases	02 Hours
	Total effort	**05 Minutes**	**Total effort**	**10 Hours**

It can be observed that our framework takes lesser execution time. The manual approach is an ad-hoc or monkey testing as no model is used in the testing process and testers do not follow specialized protocol. Manual testing is the verification of specifications outlined in the requirement document. Since no usage model is employed in manual testing so computing requirements coverage becomes difficult. The effort required to implement the MBT approach is initially quite expensive. The results show that in terms of man-hours, manual testing performs better in the first go. Contemplating it from the regression testing or sanity testing perspective, manual testing is generally considered expensive as testers have to explore every path and transition of AUT. Yet our framework would require less time to perform regression and sanity testing, thus reducing the testing time of AUT and ensuring maximum coverage of paths and transitions.

Table 4. Overall comparison of the proposed framework with the manual approach.

Outlook Email Application	Model Coverage	Test Case Execution Time	Test Case Generated	Test Case Executed	Total effort size (Man Hours)
Proposed Framework	100 %	03 Minutes	28	28	169 Hours
Manual Approach	-	08 Hours	25	25	42 Hours

Initially, MBT takes considerable time to set up the environment, but if we analyze performance and results in the long run, we can observe that MBT performs well in regression and sanity testing in terms of testing time and cost. In manual testing, there is always a risk that the tester may overlook some aspects which were not in the previous release and are added in the current release due to fixing bug. Whereas in such scenarios, MBT takes minimal time with low resource requirements and provides better testing in terms of path coverage and product quality.

Smartphone apps are smart and respond accordingly as per user interactions. These apps are context-aware and react to the input and environment, therefore testing such applications is not an easy task. Through

experimentation, we tried to answer our research questions. Our first research question is to know to what extent MBT can be applied for smartphone app testing. We found that we can effectively apply MBT on smartphone apps and can generate test cases automatically.

MBT supports automation but there is as such no complete automation of the entire testing process. Nonetheless, manual editing is involved in this process, but we can achieve maximum automation by reducing human intervention. By analyzing the experimental results, we found that initially, MBT is substantially expensive. The app can't reach its final form in a single sprint as it takes several test cycles to ensure the quality of the app. MBT works better than the manual approach in regression testing. Due to the repetitive tasks, testers often overlook working functionalities of the app in the post-bug fixing releases and this could have serious consequences. MBT ensures end-to-end testing every time and provides reliable testing. The experimental results show that time to execute all the generated test cases was reduced by more than 50%, which saved project time and resources. MBT offers rigorous testing and is helpful in regression testing and ensures maximum path coverage. However, MBT requires ample modeling knowledge about UML. MBT can also be problematic if the size of the AUT is large as space explosion is a major problem with MBT.

6. CONCLUSION

We presented an MBT framework for smartphone app testing. Our framework ensures maximum state coverage of the AUT. We used statechart diagrams to model the functional behavior of AUT. Statecharts are suitable to model state behavior and events that keep changing according to user interactions. We validated our framework on the Microsoft Outlook smartphone app and the results were promising. We automatically generated test cases for AUT using MBT and this automation process may facilitate regression testing. Statecharts facilitate modeling apps with minimum functional redundancy and ensure maximum path coverage of AUT. The test cases generated through our framework are comprehensive and cover the overall end-to-end functionality of the app. Though MBT provides a systematic way to test an app in an automated fashion, still some manual steps are indispensable e.g., modeling part using a third-party tool is essentially a manual process. However, this step can be automated by using NLP tools and techniques, which can translate the functional requirements of the AUT into a UML-based model. Employing NLP techniques for translating functional requirements to UML notations is envisaged as a future dimension.

REFERENCES

[1] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, **A general framework for comparing automatic testing techniques of Android mobile apps**, *J. Syst. Softw.*, vol. 125, pp. 322–343, 2017.

[2] F. Tong and Z. Yan, **A hybrid approach of mobile malware detection in Android**, *J. Parallel Distrib. Comput.*, vol. 103, pp. 22–31, 2017.
[3] Y.-D. Lin, J. F. Rojas, E. T.-H. Chu, and Y.-C. Lai, **On the accuracy, efficiency, and reusability of automated test oracles for android devices**, *IEEE Trans. Softw. Eng.*, vol. 40, no. 10, pp. 957–970, 2014.
[4] R. Abdul, **An Improved Model for Model based Software Testing**, *Int. J. Comput. Sci. Netw. Secur.*, vol. 17, no. 1, pp. 151–154, 2017.
[5] A. M. Mirza and M. N. A. Khan, **An automated functional testing framework for context-aware applications**, *IEEE Access*, vol. 6, pp. 46568–46583, 2018.
[6] M. Aamir Mehmood, K. Muhammad Naeem Ahmed, and I. Saleem, **Identifying Test Complexity Metrics for Multicore, Cloud, Mobile, Ubiquitous and Context Aware Computing**, *Int. J. Comput. Sci. Inf. Secur.*, vol. 14, no. 10, 2016.
[7] R. Swain, V. Panthi, P. K. Behera, and D. P. Mohapatra, **Automatic test case generation from UML state chart diagram**, *Int. J. Comput. Appl.*, vol. 42, no. 7, pp. 26–36, 2012.
[8] J. Lee and H. Kim, **QDroid: Mobile Application Quality Analyzer for App Market Curators**, *Mob. Inf. Syst.*, vol. 2016, 2016.
[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, **MobiGUITAR: Automated model-based testing of mobile apps**, *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, 2014.
[10] K. Mao, M. Harman, and Y. Jia, **Robotic testing of mobile apps for truly black-box automation**, *IEEE Softw.*, vol. 34, no. 2, pp. 11–16, 2017.
[11] A. R. Espada, M. del M. Gallardo, A. Salmerón, and P. Merino, **Using model checking to generate test cases for android applications**, *ArXiv Prepr. ArXiv150402440*, 2015.
[12] M. A. Mehmood, M. Khan, and W. Afzal, **Automating Test Data Generation for Testing Context-Aware Applications**, in *IEEE 9th International Conference on Software Engineering and Service Science (ICSESS)*, 2018, pp. 104–108.
[13] M. Aamir Mehmood, M. N. A. Khan, and W. Afzal, **Transforming context-aware application development model into a testing model**, in *8th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, Beijing, 2017, pp. 177–182.
[14] A. Kaur and V. Vig, **Systematic review of automatic test case generation by UML diagrams**, *Int. J. Eng. Res. Technol.*, vol. 1, no. 7, 2012.
[15] R. Dev, A. Jääskeläinen, and M. Katara, **Model-based GUI testing: Case smartphone camera and messaging development**, in *Advances in Computers*, vol. 85, Elsevier, 2012, pp. 65–122.
[16] V. Gudmundsson, M. Lindvall, L. Aceto, J. Bergthorsson, and D. Ganesan, **Model-based Testing of Mobile Systems–An Empirical Study on QuizUp Android App**, *ArXiv Prepr. ArXiv160600503*, 2016.
[17] Y. Jing, G.-J. Ahn, and H. Hu, **Model-based conformance testing for android**, in *International Workshop on Security*, 2012, pp. 1–18.

[18] M. Ahmed, R. Ibrahim, and N. Ibrahim, **An Adaptation Model for Android Application Testing with Refactoring**, *Int. J. Softw. Eng. Its Appl.*, vol. 9, no. 10, pp. 65–74, 2015.
[19] OMG, **About the Unified Modeling Language Specification Version 2.0.** https://www.omg.org/spec/UML/2.0/About-UML/ (accessed Jun. 16, 2020).
[20] T. Azim and I. Neamtiu, **Targeted and depth-first exploration for systematic testing of android apps**, in *ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641–660.
[21] G. de Cleva Farto and A. T. Endo, **Evaluating the model-based testing approach in the context of mobile applications**, *Electron. Notes Theor. Comput. Sci.*, vol. 314, pp. 3–21, 2015.
[22] J. Gutiérrez, M. Escalona, and M. Mejías, **A model-driven approach for functional test case generation**, *J. Syst. Softw.*, vol. 109, pp. 214–228, 2015.
[23] B. P. Lamancha, M. Polo, D. Caivano, M. Piattini, and G. Visaggio, **Automated generation of test oracles using a model-driven approach**, *Inf. Softw. Technol.*, vol. 55, no. 2, pp. 301–319, 2013.
[24] R. Ramler, G. Buchgeher, and C. Klammer, **Adapting automated test generation to GUI testing of industry applications**, *Inf. Softw. Technol.*, vol. 93, pp. 248–263, 2018.
[25] A. R. Espada, M. del M. Gallardo, A. Salmerón, and P. Merino, **Performance analysis of Spotify® for Android with model-based testing**, *Mob. Inf. Syst.*, vol. 2017, 2017.
[26] S. Salva and S. R. Zafimiharisoa, **APSET, an Android Application Security Testing tool for detecting intent-based vulnerabilities**, *Int. J. Softw. Tools Technol. Transf.*, vol. 17, no. 2, pp. 201–221, 2015.
[27] C. Tao and J. Gao, **On building a cloud-based mobile testing infrastructure service system**, *J. Syst. Softw.*, vol. 124, pp. 39–55, 2017.
[28] P. Arcaini and A. Gargantini, **Test generation for sequential nets of Abstract State Machines with information passing**, *Sci. Comput. Program.*, vol. 94, pp. 93–108, 2014.
[29] W. Yang, M. R. Prasad, and T. Xie, **A grey-box approach for automated GUI-model generation of mobile applications**, in *International Conference on Fundamental Approaches to Software Engineering*, 2013, pp. 250–265.
[30] I. C. Morgado, A. C. Paiva, and J. P. Faria, **Automated pattern-based testing of mobile applications**, in *9th International Conference on the Quality of Information and Communications Technology*, 2014, pp. 294–299.
[31] Sami-Ul-Haq, Khan, M. N. A., Mirza, A. M., Saif Ur Rehman, Raja Asif Wagan, & Saleem, I. **Addressing Communication, Coordination and Cultural Issues in Global Software Development Projects**. EMITTER

International Journal of Engineering Technology, 9(1), 13-30, 2021. https://doi.org/10.24003/emitter.v9i1.558