computing@computingonline.net
www.computingonline.net

Print ISSN 1727-6209
On-line ISSN 2312-5381
International Journal of Computing

A GUI TESTING STRATEGY AND TOOL FOR ANDROID APPS

Moheb R. Girgis, Bahgat A. Abdel Latef, Tahany Akl

Department of Computer Science, Faculty of Science, Minia University, El-Minia, Egypt
moheb.girgis@mu.edu.eg, bahgat.ahmed@mu.edu.eg, thany_00_11@yahoo.com

Paper history:
Received 24 July 2019
Received in revised form 23 April 2020
Accepted 08 July 2020
Available online 27 September 2020

Keywords:
Mobile apps;
GUI testing;
Android app GUI testing;
Model-based testing;
Automated testing tools;
Event-based coverage criteria;
Robotium test framework.

Abstract: The increasing popularity of Android and the GUI-driven nature of its apps have motivated the need for applicable automated GUI testing techniques. This paper presents a proposed strategy and a supporting tool for GUI testing of Android apps. The strategy employs a model-based approach to capture the event-driven nature of Android apps. It includes two phases: Modeling Phase and Test Evaluation Phase. In the modeling phase, an event sequence diagram (ESD) is created for each activity in the app under test (AUT), which depicts its events and possible transitions between them, and used to generate event sequences (test cases). In the test evaluation phase, certain event-based coverage criteria are employed to measure the adequacy of the generated test cases. The proposed tool analyses the AUT, creates an ESD for each activity, and generates event sequences. It handles the event sequences explosion problem and ensures the event sequences feasibility. For each event sequence, the tool generates a test script and a corresponding Robotium test class, adds it to the AUT and executes it. The paper also presents a case study that illustrates the use of the proposed strategy and tool for testing a simple Android app.

1. INTRODUCTION

The number of mobile apps continues to grow at a rapid pace and the requirements for their performance grow as they become more advanced and are exposed to higher loads of users. This makes the subject of mobile app testing important and of continuing interest. Mobile apps heavily depend on graphical user interfaces (GUIs). Testing of these GUIs is very important, as it lets developers ensure that the app meets its functional requirements and achieves a high standard of quality such that it is more likely to be successfully adopted by users. Automating these tests is very useful since it saves a lot of time, but it is very difficult due to the complexity of mobile apps and the limited resources available in mobile devices.

Due to the widespread use of Android platform, the work in this paper focuses on testing the GUI of Android apps. The paper presents a proposed strategy for GUI testing of Android apps, and a supporting tool for analyzing the app under test (AUT), generating test cases based on certain event-based coverage criteria, adapted for Android app, and executing these test cases. The proposed strategy employs a model-based approach to capture the event-driven nature of Android apps. The employed model is the event sequence diagram (ESD), which depicts the events for an app and the possible transitions between them. The proposed tool collects the IO/Clickable views in each activity of the AUT and their events. Then, it generates an ESD for each activity, and uses it to generate a set of event sequences according to the specified event-based criteria. For each event sequence, the tool generates a test script, from which it generates a Robotium test class, adds it to the AUT and executes it. The paper also presents a case study that illustrates the use of the proposed GUI testing strategy and the supporting tool for testing a simple Android app.

The paper is organized as follows: Section 2 presents a review of related research in the area of model-based GUI testing of Android apps. Section 3 briefly describes the main components of Android app GUI. Section 4 gives an overview of Robotium Test Framework. Section 5 describes the proposed

GUI testing strategy for Android apps. Section 6 describes the proposed testing tool, which implements the proposed strategy. Section 7 presents an example of using the proposed approach and tool for testing a simple Android app. Section 8 presents the conclusion of the work presented in this paper.

2. RELATED WORK

For Android app testing, several approaches have been proposed that focus on test input generation, i.e., event generation. These approaches can be categorized as follows: random testing (see e.g. [1-3]); model-based testing (see e.g. [4-10]); symbolic execution testing (see e.g. [11-15]); and search-based testing and other testing approaches, which use more sophisticated techniques to generate events (see e.g. [16-20]).

Since this paper focuses on model-based GUI testing of Android apps, a review of related research in this area is given below.

Amalfitano et al. [4] presented a technique for rapid crash testing and regression testing of Android apps. The technique is based on a crawler that automatically builds a model of the app GUI and obtains test cases that can be automatically executed. Amalfitano et al. [5] presented AndroidRipper, an automated technique that tests Android apps via their GUI. AndroidRipper is based on a user-interface driven ripper that automatically explores the app GUI with the aim of exercising the application in a structured manner. Yang et al. [6] presented a grey-box approach and a tool, for automatically extracting a model of a given mobile app. In this approach, static analysis extracts the set of events of the app GUI. Then, dynamic crawling reverse-engineers a model of the app, by exercising these events on the running app. Azim and Neamtiu [7] presented Android App Explorer (A^3E), an approach and tool that allows Android apps to be explored while running on actual phones, yet without requiring access to the app source code. They construct a high-level control flow graph from the app bytecode that captures legal transitions among activities, and use it to develop an exploration strategy named Targeted Exploration that permits fast, direct exploration of activities, including activities that would be difficult to reach during normal use. They also developed a strategy named Depth-first Exploration that mimics user actions for exploring activities and their constituents. Choi et al. [8] proposed an automated technique, called SwiftHand, for generating sequences of test inputs for Android apps. The technique uses machine learning to learn a model of the app during testing, uses the learned model to generate user inputs that visit unexplored states of the app, and uses the execution of the app on the generated inputs to refine the model. Amalfitano et al. [9] presented MobiGUITAR for automated GUI-driven testing of Android apps, which is based on observation, extraction, and abstraction of the run-time state of GUI widgets. The abstraction is a scalable state-machine model that, together with test coverage criteria, provides a way to automatically generate test cases. Su et al. [10] introduced Stoat, a guided approach to perform stochastic model-based testing on Android apps. Stoat operates in two phases: (1) Given an app as input, it uses dynamic analysis enhanced by a weighted UI exploration strategy and static analysis to reverse engineering a stochastic model of the app GUI interactions; and (2) it adapts Gibbs sampling to iteratively mutate/refine the stochastic model and guides test generation from the mutated models toward achieving high code and model coverage and exhibiting diverse sequences.

The proposed approach differs from the reviewed approaches in the following aspects: (1) it creates a simple model, ESD, to represent the events in the UI of each activity and possible transitions between them, and uses it to generate test cases; (2) it employs event-based coverage criteria, adapted for Android app, to measure the adequacy of the generated test cases; (3) it significantly reduces the number of generated event sequences by identifying subsumption between different event sequences and discarding any sequence that is a subsequence of another one, and by checking event sequences feasibility, i.e. their ability to be executed, and discarding any sequence that includes any illegal event subsequences; and (4) it utilizes the features of the Robotium Test Framework to extract the AUT activities' views and related information, and to execute the generated test classes.

3. ANDROID APPS UI COMPONENTS

The main components of an Android app, which dictate the UI and handle the user interaction to the mobile device screen, are activities [21]. An activity represents a single screen with a UI. An app may have more than one activity. For example, an email app might have one activity for showing new emails, another activity for composing an email, and another activity for reading emails. In this case, these activities can interact with each other, and the one, which is presented when the app is launched, is called the *main activity*.

The UI for each component (activity) of an app is defined using a hierarchy of View and ViewGroup objects. Each view group is an invisible container that organizes child views, while the child views

may be input controls or other widgets that draw some part of the UI. Input controls are the interactive components in the app UI. Android provides a wide variety of controls, such as TextView, EditText, Button, CheckBox, RadioButton, RadioGroup, and many more. UI inputs of an app include the input controls and their events (actions) for each activity. Events are a useful way to collect data about a user's interaction with interactive components of apps, such as button presses or screen touch, etc. When an event happens, the corresponding *Event Handler*, which is the method that actually handles the event, is called to perform any required task.

4. ROBOTIUM TEST FRAMEWORK

Robotium is an extension of the Android test framework and was created to make it easy to write UI test automation scripts for Android apps [22]. Robotium tests allow the tester to define test cases across Android activities. Robotium tests perceive the AUT as black box, i.e., it only interacts with the user interface and not via the internal code of the app.

The main class for testing with Robotium is *Solo*. Through a Solo object and its methods, we can set values in input fields, click on buttons and get results from other UI components. Methods of JUnits Assert class can then be used to check those results. Table 1 shows examples of the Solo methods.

Table 1. Examples of Solo objects methods

Method	Description
assertCurrentActivity (text, Activity)	Verify whether the current activity is the activity which is passed as the send parameter.
getCurrentViews()	Returns an ArrayList of Views currently displayed in focused Activity.
clickOnButton(int)	Click on button with specified index.
enterText(int, text)	Type *text* to editbox with specified index.
clickOnCheckbox(int)	Click on checkbox with given index
clickOnRadioButton (int)	Click on Radio button with given index
clearEditText(int)	Clear text in edit box with given index

5. THE PROPOSED ANDROID APPS UI TESTING STRATEGY

This section describes the proposed strategy for UI testing of Android apps. In this strategy, testing is conducted at two levels: activity level and app level. Firstly, all activities in the AUT are identified. This is a process to divide a large complex app into independent components that can be tested in isolation. Then, all views within each activity and their events are identified. In the activity level testing, each activity is tested in isolation to verify whether it works as expected. Then, in the app level testing, the app as a whole is tested to verify whether all of its activities can communicate with each other to complete the desired tasks. In this level, each activity will be treated as a trusted component because it has passed the activity level testing. An execution path of the app will be represented by a sequence of these trusted components.

Each testing level of the proposed strategy includes two phases: *Modeling phase* and *Test Evaluation phase*. In the modeling phase, a model is created for each activity/app that is used to generate test cases for testing the UI of the activity/app, and in the test evaluation phase, event-based coverage criteria are employed to determine whether the generated test cases have adequately tested the UI of the activity/app. In the next two subsections, the model used to represent each activity/app, and the UI test coverage criteria employed in the test evaluation phase, are described.

5.1 THE EVENT SEQUENCE DIAGRAM

All the possible execution paths in an app UI are represented by a model based on the Finite State Machine (FSM) model, called the *Event Sequence Diagram* (ESD) [23]. In an ESD, each node represents an event while a state transition is determined based on how the current node is responding to the inputs. An ESD is created for each app activity, then the ESDs of all app activities are grouped together to create an *App ESD*.

An ***Event Sequence Diagram*** D is a two-tuple $<N, E>$ where:

1. N is a set of nodes representing all the events for an activity/app. Each node $n \in N$ represents an event in D.
2. $E \subseteq N \times N$ is a set of directed edges between the nodes. Each edge $e \in E$ represents transition from one event to the next. An event e_2 is said to follow e_1 if and only if e_2 can be initiated after e_1.

The constructed ESDs are used in generating test cases (event sequences) for each activity, in the activity level testing, and then for the app as whole, in the app level testing, based on certain event-based coverage criteria.

5.2 THE UI TESTING COVERAGE CRITERIA

As described in the previous subsection, the events identified within each UI's activity are represented by an ESD, which is used to generate event sequences as test cases. In order to measure the test adequacy of test cases, Memon [24] has defined two kinds of event-based coverage criteria: (1) intra-component criteria for events within a component and (2) inter-component criteria for events across components. Intra-component criteria include: event coverage, event-interaction coverage, and length-n event-sequence coverage; and inter-component criteria include: invocation coverage, invocation-termination coverage, and length-n event-sequence coverage. We adapted these criteria for Android apps. We called the adapted criteria *Intra-activity criteria* and *Inter-activity criteria*, respectively, and defined them as follows:

Intra-activity criteria

– ***Event Coverage***: each event in the activity should be triggered at least once.
– ***Event-Interaction Coverage***: after an event e has been performed, all events that can interact with e should be executed at least once.
– ***Length-n Event-sequence Coverage***: all length-n event sequences within an activity should be executed at least once.

These criteria are employed in the test evaluation phase of the activity level testing.

Inter-activity criteria

– ***Invocation Coverage***: each event that starts a new activity must be performed at least once.
– ***Invocation-termination Coverage***: all length 2 event sequences consisting of an event followed by one of the invoked activity's termination events has to be tested.
– ***Length-n Event-sequence Coverage***: all length-n event sequences that start with an event in an activity and end with an event in another activity must be tested.

These criteria are employed in the test evaluation phase of the app level testing.

Having defined the ESD and the event-based coverage criteria for UI testing, the following steps are performed in order to apply the proposed UI testing strategy to test the UI of an Android app:

– Identify the app activities and create the corresponding ESDs.
– Using the activities ESDs, construct the App ESD.
– Generate test cases according to the defined coverage criteria.
– Execute the test cases.
– Analyze and evaluate the execution results.

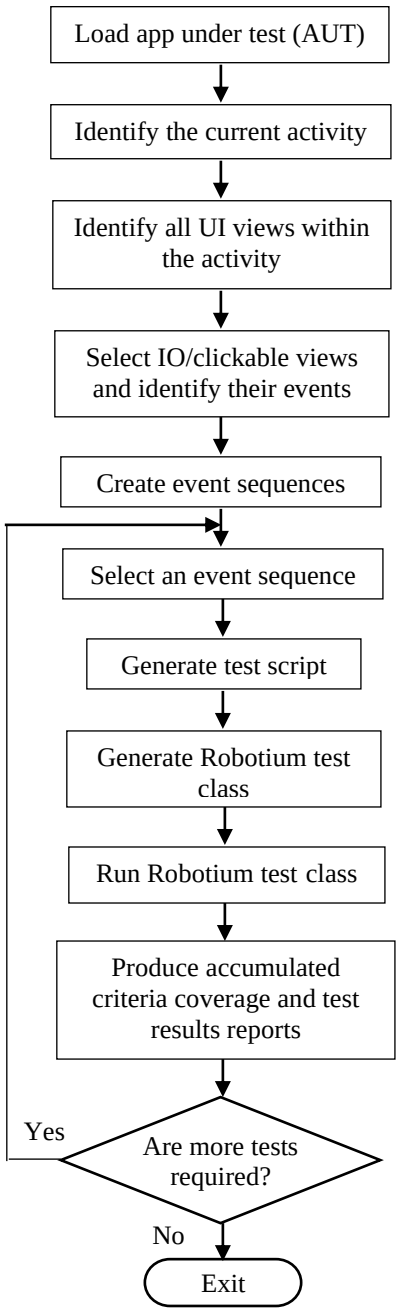

Figure 1 – The steps of the proposed GUI testing approach of Android Apps

6. THE PROPOSED ANDROID APPS UI TESTING TOOL

This section describes the proposed tool that implements the proposed UI testing strategy for Android apps.

Fig. 1 shows the steps that are followed by the tool to generate and execute test cases for each activity in the AUT. The tool utilizes the functionalities provided by the Robotium Test Framework in two of these steps: in analyzing the AUT activities to extract their views and related information, and in executing the generated test class for each event sequence.

```
Generate_and_Run_Test_Cases Algorithm
Input: app, the AUT
Output: Test classes, Test results report, and Criteria
        coverage report
Begin
  1. Create a Solo object, solo.
  2. Identify the current activity act in app, by
     using the method solo.getCurrentActivety().
  3. Detect all UI views in act, by using the
     method solo.getCurrentViews(), then select
     from them IO/clickable views and save the
     text of each view with its event in the event
     list L.
  4. Generate the Event Index List IL, which
     contains for each event its index in L, type,
     text, and id
  5. S = ∅      // Initialize Event Sequences Set
  6. For each event e ∈ L
  7. Begin
        // generate all possible legal sequences of e
        // with all other events in L and store them
        // in Se
  8.    Se = Generate_Event_Sequences(e, L)
  9.    S = S ∪ Se
  10. End For
  11. For each event sequence s ∈ S
  12. Begin
  13.    Generate_Test_Script(s, IL) →
                                        testScriptFile
  14.    Create_Test_Class(testScriptFile) →
                        Robotium test class testClassFile
  15.    add testClassFile to app.
  16.    Run app with the Robotium test class.
  17.    Produce accumulated Criteria Coverage
        Report and Test Results Report.
  18. End For
End.
```

Figure 2 – Generate_and_Run_Test_Cases Algorithm

The procedural details of the tool steps are described in the *Generate_and_Run_Test_Cases* algorithm, shown in Fig. 2. In this algorithm, three data structures are created: *Event list L*, which contains, for each activity, its IO/clickable views with their events; *Event Index List IL*, which contains for each view its index in L, type, text, and id; and *Event Sequences Set S*, which contains all possible legal event sequences.

The tool builds an ESD for each activity, and generates test cases based on the ESDs of the AUT and the coverage criteria, described in Sec. 5. The input to the tool is the AUT, and the outputs produced by the tool are: UI event sequences, Executable test cases, Criteria coverage report, and Test results report.

```
Procedure Generate_Event_Sequences(e, L)
Input: an event e; Event List L
Output: Event sequences list for event e, Se
Begin
  1. Se = ∅;
  2. While there are possible event sequences
     from e to other events in L
  3. Begin
  4.    Generate a possible event sequence s from
        e to other events in L;
  5.    If s is a subsequence of another generated
        sequence in Se or it includes any illegal
        event subsequences Then
  6.       Discard s;
  7.    Else
  8.       Add s to list Se;
  9.    End If
  10. End While
  11. Return Se;
End.
```

Figure 3 – Generate_Event_Sequences Procedure

The tool works as follows: Firstly, it uses a Solo object to identify the current activity and detect its views. From the detected views and related information, which includes the view's type, event, text and id, the tool selects only IO/clickable views and saves the text of each view with its event in the Event List *L*, and generates the Event Index List *IL*. Then, for each event $e \in L$, the tool generates all possible legal sequences of e with all other events in *L*, using the procedure *Generate_Event_Sequences*, shown in Fig. 3, and stores them in the Event Sequences Set *S*. To overcome the event sequences explosion problem, the procedure identifies subsumption between different event sequences, and discards any sequence that is a subsequence of a previously generated sequence. Also, to ensure the feasibility of event sequences, i.e., their ability to be executed, the procedure discards any sequence that includes any illegal event subsequences. Next, for each event sequence $s \in S$, the tool generates a test script by using IL and the procedure *Generate_Test_Script*, shown in Fig. 4. For each event in s, the test script includes a line that contains the type of the corresponding view, its text and id. From the generated test script, the tool generates a Robotium test class, using the procedure *Create_Test_Class*, shown in Fig. 5, and adds it to *app*. Finally, *app* with the Robotium test class is executed, and the tool produces test results and criteria coverage reports. These reports provide the tester with information about the detected errors, if any, and the fulfilment of the specified testing coverage criteria, to decide whether more tests are required or not.

Procedure **Generate_Test_Script (***s, IL***)**
Input: an event sequence *s*; Event Index List *IL*
Output: A test script file for the event sequence *s*, *testScriptFile*
Begin
1. For each $e \in s$
2. Begin
3. Get the view type that corresponds to event *e*, with its text and id, from *IL*
4. Add a line representing the action of this view, which contains this information, to the test script.
5. End
6. Save the generated test script in *testScriptFile*

End.

Figure 4 – Generate_Test_Script Procedure

Procedure Create_Test_Class (testScriptFile)
Input: The test script for an event sequence, *testScriptFile*
Output: A Java test class file, *testClassFile*
Begin
1. Insert the following lines into testClassFile:
   public void setUp() throws Exception {
   solo = new Solo(getInstrumentation(), getActivity());
   }
   public void testRun() {
2. While ! testScriptFile.EOF()
3. Begin
4. Read a line *ln* from testScriptFile
5. From *ln*, get view_type, text, and id
6. If view_type == "RadioButton" || view_type == "Button" Then
7. Insert the following instruction into testClassFile:
   solo.clickOnView(solo.getView(id));
8. Else If view_type == "TextView" Then
9. Insert the following instructions into testClassFile:
   TextView textField = (TextView) solo.getView(id);
   assertEquals((String)textField.getText(), text);
10. Else If view == "EditText" Then
11. Insert the following instructions into testClassFile:
    EditText vEditText = (EditText) solo.getView(id);
    solo.enterText(vEditText, some text);
12. Else If …
    ….
13. End;
14. Insert "}" into testClassFile

End.

Figure 5 – Create_Test_Class Procedure

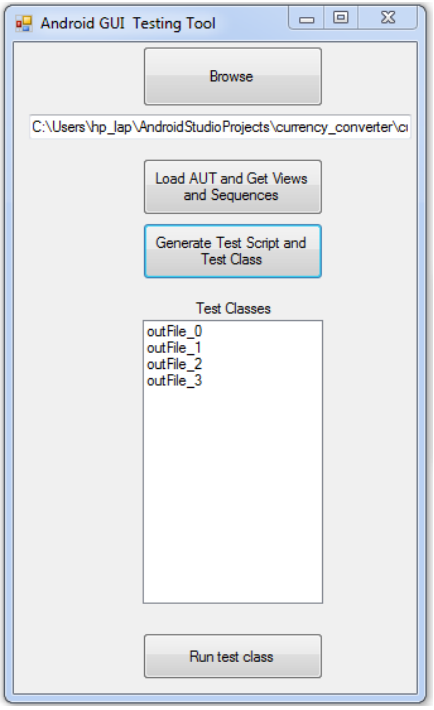

Figure 6 – The interface of the proposed tool

The presented automated GUI testing tool was developed using Android Studio 3.0.1 and Microsoft Visual Studio 2010 on a Laptop with processor: Intel Core i5 – 4300U CPU – 2.50 GHz and RAM: 8 GB. The AUT tests are executed using an Android emulator. The tool provides users with the GUI interface shown in Fig. 6. The main components of the tool interface are: four buttons, "Browse", "Load AUT and Get Views and Sequences", "Generate Test Script and Test Class", "Run test class"; one EditTextBox; and one ListBox. Firstly, the user selects an app for testing by clicking on "Browse" button. Then, when the user clicks "Load AUT and Get Views and Sequences" button, the selected app is loaded, list of all the clickable/IO views of each activity of this app and their events are extracted, and from this list the tool generates all possible legal event sequences of views. Next, a cycle starts: when the user clicks "Generate Test Script and Test Class" button, the tool selects an event sequence and generates a test script for it, then generates a Robotium test class for the generated test script, and shows its file name in the ListBox. This test class is added to the AUT. Each test class contains calls to Robotium functions through a Solo object that correspond to lines in the test script. When the user clicks "Run test class" button the test class is executed. Then, the tool asks the user whether he/she wants to continue, if the answer is no, the tool stops, otherwise, the tool allows the user to do more tests by clicking "Generate Test Script and Test Class" button, which repeats the above cycle, (see Fig. 1).

7. CASE STUDY

This section presents an example of using the proposed approach and tool for testing a simple Android app called Currency_Converter. Fig. 7 shows the window (main activity) of this app. Its UI includes 3 buttons ("Compute", "Clear", and "Exit"), 3 Radio buttons ("Egypt", "Canada", and "Japan"), one EditText control, 4 TextView controls.

When the app is launched, its initial window/activity, shown in Fig. 7, appears. The app takes as input an amount of money in United States Dollars (USD) and outputs the equivalent amount in Egyptian Pounds, Canadian Dollars, or Japanese Yen, according to the selected country: Egypt, Canada, or Japan, respectively. It produces an error message if no country is selected or no USD amount is entered before clicking the "Compute" button. At any time, the user can clear the window, by clicking the "Clear" button, which returns the AUT to its initial state, or close the window, by clicking the "Exit" button, which quits the app.

The tool detects the IO/clickable views and saves the text of each view with its event in the events list, L, as shown in Table 2. Fig. 8 shows the events index list, IL, which contains, for each event, its index in L, its type, text, and id. Fig. 9 shows the corresponding ESD. From the list L and the ESD the tool generates all possible legal sequences of views.

Table 3 shows some of the generated test cases (event sequences). For each sequence the tool generates a test script as the one shown in Fig. 10, which corresponds to the event sequence [Idle -1-3-7-5] (Test case T1). Each line in the test script contains the view type, text, and id, separated by commas. If a view does not have text, e.g., EditText, the text positon is left empty. Then, the tool generates a Robotium test class for the generated test script, as shown in Fig. 11, and adds it to the AUT. Finally, the app with the test class is executed.

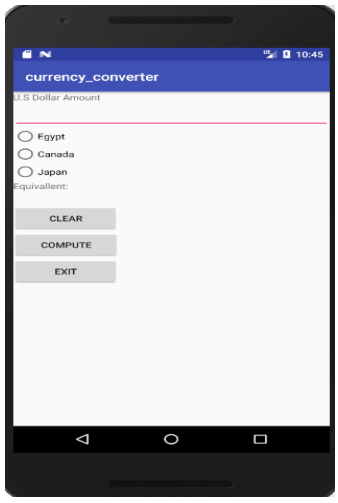

Figure 7 – The Currency_Converter App window (main activity)

Table 2. Event List of the example app

Index	Text	Event
1	""	enterText
2	"Egypt"	Click
3	"Canada"	Click
4	"Japan"	Click
5	""	Output equivalent amount
6	"Clear"	Click
7	"Compute"	Click
8	"Exit"	Click
9	""	Show error message

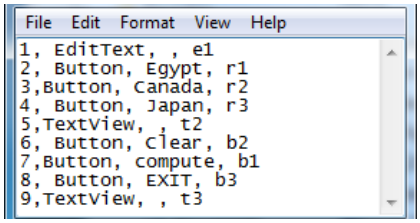

Figure 8 – The Event Index List of the example app

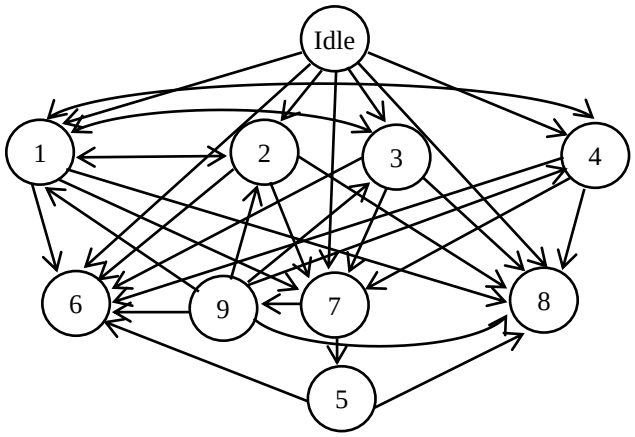

Figure 9 – The ESD of the GUI of the example app (10 nodes and 35 edges)

Table 3. Some of the test cases generated for the main activity of the example app

Test Case No.	Test Case
T1	Idle -1-3-7-5
T2	idle-1-2-7-5
T.3	Idle -1-7-9
T4	Idle -1-8
T5	Idle – 4-7-9
T6	Idle -3-6
T7	Idle -4-7-9-6
T8	idle-1-7-9-6
T9	Idle -1-7-9-2
T10	Idle -4-1-7-5
T11	Idle -7-9-1-7-9
T12	Idle -7-9-1-2-7-5
T13	Idle -1-4-7-5
T14	Idle -7-9-1-7-9-1-6
T15	Idle-8
...	

Fig. 12 shows part of the Criteria Coverage Report produced by tool for the test cases, shown in Table 3. It shows for each test case: the event sequence; the Event Coverage, which includes: the newly covered events, and the accumulated event coverage percentage; the Event-Interaction Coverage, which includes: the newly covered edges, and the accumulated event-interaction coverage; and finally, Length-n Event-sequence Coverage.

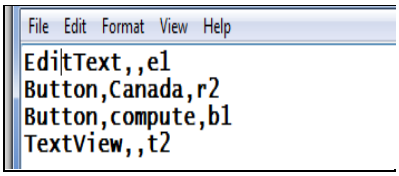

Figure 10 – The test script for the event sequence [Idle -1-3-7-5] (Test case T1)

```
public void setUp() throws Exception {
      solo = new Solo(getInstrumentation(),
      getActivity());
}
public void testRun() {
      EditText vEditText1 =
                  (EditText)solo.getView(R.id.e1);
      solo.enterText(vEditText1, "10 ");
      solo.clickOnView(solo.getView("r2"));
      solo.clickOnView(solo.getView("b1"));
      TextView textField1 = (TextView)solo.
                                    getView("t2");
      assertEquals((String)textField1.getText(),
                                          "20");
}
```

Figure 11 – The test class generated for the test script shown in Fig. 10

8. CONCLUSION

This paper presented a proposed strategy for testing the GUIs of Android apps. This strategy employs a model-based approach to capture the event-driven nature of Android apps. The employed model is the ESD, which depicts the events for an app and the possible transitions between them. The proposed strategy includes two phases: Modeling Phase and Test Evaluation Phase. In the modeling phase, an ESD is created for each activity in the AUT and used to generate test cases (event sequences). In the test evaluation phase, certain event-based coverage criteria, adapted for Android app, are employed to measure the adequacy of the generated test cases for testing the GUI of the AUT.

Then, the paper presented a supporting tool for analyzing the AUT, generating test cases, and executing these test cases. The proposed tool collects the IO/Clickable views in each activity of the AUT and the associated events. Then, it generates an ESD for each activity, and uses it to generate a set of event sequences according to the specified coverage criteria. The tool handles the event sequences explosion problem, by discarding any sequence that is a subsequence of another sequence; and ensures the feasibility of event sequences, by discarding any sequence that includes any illegal event subsequences. By considering these two situations, the number of generated sequences is significantly reduced.

```
Criteria Coverage Report
App Name: Currency_Converter
Activity Name: Main Activity
ESD: 10 nodes, 35 edges
Test Case No.: T1
      Event Sequence: Idle-1-3-7-5
      Event Coverage: Newly covered events: Idle, 1, 3, 5, 7,
            Accumulated Event Coverage: 50%
      Event-Interaction Coverage: Newly covered edges:
      Idle-1, 1-3, 3-7, 7-5
      Accumulated Event-Interaction Coverage: 11.43%
      Length-n Event-sequence Coverage: n = 5
Test Case No.: T2
      Event Sequence: idle-1-2-7-5
      Event Coverage: Newly covered events: 2,
            Accumulated Event Coverage: 60%
      Event-Interaction Coverage: Newly covered edges: 1-2,
      2-7,
      Accumulated Event-Interaction Coverage: 17.14%
      Length-n Event-sequence Coverage: n = 5
Test Case No.: T3
      Event Sequence: Idle-1-7-9
      Event Coverage: Newly covered events: 9,
            Accumulated Event Coverage: 70%
      Event-Interaction Coverage: Newly covered edges: 1-7,
      7-9,
      Accumulated Event-Interaction Coverage: 22.86%
      Length-n Event-sequence Coverage: n = 4
Test Case No.: T4
      Event Sequence: Idle-1-8
      Event Coverage: Newly covered events: 8,
            Accumulated Event Coverage: 80%
      Event-Interaction Coverage: Newly covered edges: 1-8,
      Accumulated Event-Interaction Coverage: 25.71%
      Length-n Event-sequence Coverage: n = 3
Test Case No.: T5
      Event Sequence: Idle-4-7-9
      Event Coverage: Newly covered events: 4,
            Accumulated Event Coverage: 90%
      Event-Interaction Coverage: Newly covered edges:
      Idle-4, 4-7,
      Accumulated Event-Interaction Coverage: 31.43%
      Length-n Event-sequence Coverage: n = 4
Test Case No.: T6
      Event Sequence: Idle-3-6
      Event Coverage: Newly covered events: 6,
            Accumulated Event Coverage: 100%
      Event-Interaction Coverage: Newly covered edges:
      Idle-3, 3-6,
      Accumulated Event-Interaction Coverage: 37.14%
      Length-n Event-sequence Coverage: n = 3
```

Figure 12 – Part of the Test Coverage Report produced by tool for the test cases, shown in Table 3

For each event sequence, the tool generates a test script, then generates a corresponding Robotium test class, adds it to the AUT and executes it. The tool utilizes the functionalities provided by the Robotium framework for extracting information about the views in each activity in the AUT, and for executing the generated test class of each event sequence.

Finally, the paper presented a case study that illustrates the use of the proposed GUI testing strategy and the supporting tool in testing the UI of a simple Android app.

9. REFERENCES

[1] *UI/Application Exerciser Monkey — Android Developers*. [Online]. Available at: http://developer.android.com/tools/help/monkey.html.

[2] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," *Proceedings of the 6th International Workshop on Automation of Software Test, AST'11*, Waikiki, Honolulu, HI, USA, May 23-24, 2011, pp. 77–83.

[3] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for android apps," *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russia, August 18–26, 2013, pp. 224-234.

[4] D. Amalfitano, A.R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for Android mobile application testing," *Proceedings of the IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11)*, Berlin, Germany, 21-25 March, 2011, pp. 252–261.

[5] D. Amalfitano, A.R. Fasolino, P. Tramontana, S. De Carmine, and A.M. Memon, "Using GUI ripping for automated testing of Android applications," *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, ASE'12*, Essen, Germany, September 3-7, 2012, pp. 258–261.

[6] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13*, Rome, Italy, March 16-24, 2013, pp. 250–265.

[7] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13*, Indianapolis, IN, USA, October 26-31, 2013, pp. 641–660.

[8] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13*, Indianapolis, IN, USA, October 26-31, 2013, pp. 623–640.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: automated model-based testing of mobile apps," *IEEE Software*, vol. 32, issue 5, pp. 53–59, 2015.

[10] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," *Proceedings of the Symposium on the Foundations of Software Engineering ESEC/FSE'17*, Paderborn, Germany, September 4–8, 2017, pp. 245-256.

[11] N. Mirzaei, S. Malek, C.S. Păsăreanu, N. Esfahani, and R. Mahmood, "Testing Android apps through symbolic execution," *ACM SIGSOFT Software Engineering Notes*, vol. 37, issue 6, pp. 1-5, 2012.

[12] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," *Proceedings of the 20th ACM SIGSOFT Symposium on the Foundations of Software Engineering, SIGSOFT/FSE'12*, Cary, NC, USA, November 11-16, 2012, pp. 59-69.

[13] C. S. Jensen, M. R. Prasad, and A. Møller, "Automated testing with targeted event sequence generation," *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA'13*, Lugano, Switzerland, July 15-20, 2013, pp. 67–77.

[14] N. Mirzaei, H. Bagheri, R. Mahmood, and S. Malek, "SIG-Droid: Automated system input generation for android applications," *Proceedings of the 26th IEEE International Symposium on Software Reliability Engineering, ISSRE'15*, Gaithersbury, Maryland, USA, November 2-5, 2015, pp. 461-471.

[15] N. Mirzaei, J. Garcia, H. Bagheri, A. Sadeghi, and S. Malek, "Reducing combinatorics in GUI testing of android applications," *Proceedings of the 38th International Conference on Software Engineering, ICSE'16*, Austin, TX, USA, May 14-22, 2016, pp. 559–570.

[16] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: segmented evolutionary testing of android apps," *Proceedings of the 22nd ACM*

SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE'14, Hong Kong, China, November 16-22, 2014, pp. 599–609.

[17] K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for android applications," *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA'16*, Saarbrucken, Germany, July 18-20, 2016, pp. 94–105.

[18] A. Darvish and C. K. Chang, "Black-box test data generation for GUI testing," *Proceedings of the 14th International Conference on Quality Software*, 2-3 October 2014, Dallas, TX, USA, pp. 133-138.

[19] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, D. Poshyvanyk, "Mining Android app usages for generating actionable GUI-based execution scenarios," *Proceedings of the 12th IEEE/ACMWorking Conference on Mining Software Repositories, MSR'2015*, Florence, Italy, May 16-17, 2015, pp. 111–122.

[20] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI testing for Android applications," *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 30 October-3 November 2017, Urbana, IL, USA, pp. 27-37.

[21] Android – Application Components. [Online]. Available at: https://www.tutorialspoint.com/android/android_application_components.htm.

[22] Android user interface testing with Robotium – Tutorial. [Online]. Available at: http://www.vogella.com/tutorials/Robotium/article.html.

[23] P. Li, T. Huynh, M. Reformat, and J. Miller, "A practical approach to testing GUI systems," *Empirical Software Engineering*, vol. 12, issue 4, pp. 331–357, 2007.

[24] A. M. Memon, *A Comprehensive Framework for Testing Graphical User Interfaces*, PhD Thesis, Department of Computer Science, University of Pittsburgh, July 2001.

***Moheb R. Girgis**, received his B.Sc. degree from Mansoura University, Egypt, in 1974, M.Sc. degree from Assuit University, Egypt, in 1980, and Ph.D. degree from the University of Liverpool, England, in 1986. He is a professor of computer science at Minia University, Egypt.*

His research interests include software engineering, software testing, information retrieval, evolutionary algorithms, image processing, and computer networks.

***Bahgat A. Abdel Latef,** received his B.Sc. and M.Sc. degrees from Assuit University, Egypt, in 1975 and 1983, respectively, and Ph.D. degree from the University of Liverpool, England, in 1989. He is a professor of computer science at Minia University, Egypt. His research interests include software engineering, information retrieval, and computer networks.*

***Tahany Akl,** received her B.Sc. and M.Sc. degrees from Minia University, Egypt, in 2005 and 2014, respectively. She is a Ph.D. Student at Computer Science Department, Minia University, Egypt. Her research interests include software engineering and GUI testing.*