A System for GUI Testing of Android Apps with Multiple Activities

Moheb R. Girgis
Department of Computer Science
Faculty of Science
Minia University
El-Minia, Egypt

Bahgat A. Abdel Latef
Department of Computer Science
Faculty of Science
Minia University
El-Minia, Egypt

Tahany Akl
Department of Computer Science
Faculty of Science
Minia University
El-Minia, Egypt

ABSTRACT
The wide spread use of Android and the GUI-driven nature of its apps have risen the need for appropriate automated GUI testing techniques. This paper presents a proposed system for GUI testing of Android apps with multiple activities, which applies a model-based approach to capture the event-driven nature of Android apps. This approach comprises two phases: Modeling Phase and Test Evaluation Phase. In the modeling phase, for each activity in the app under test (AUT), an event sequence diagram (ESD) is built, which depicts the activity's 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 system analyses the AUT, builds an ESD for each activity, and generates event sequences. It handles the event sequences explosion problem and discards any unacceptable event sequences. For each event sequence, the system generates a test script and a corresponding Robotium test class, and executes the AUT with it. The paper also presents a case study that illustrates the use of the proposed system for testing an Android app with multiple activities, and the results of the experiments that have been conducted to evaluate the system's ability to expose some GUI errors that may occur in Android apps.

General Terms
Mobile Apps GUI Testing, Model-Based Testing, Automated GUI Testing.

Keywords
Android Apps GUI Testing, Model-Based Testing, Automated GUI Testing Tools, Event-Based Coverage Criteria, Robotium Test Framework.

1. INTRODUCTION
As mobile apps become more advanced and are exposed to higher number of users, the requirements on their performance grow, and the issue of their quality becomes very important. Mobile app testing is one of the most frequently used quality assurance techniques. Mobile apps are graphical user interface (GUI) driven apps, which makes GUI testing of such apps very important. GUI testing assures developers that their app meets its functional requirements with high quality such that it is more likely to be successfully accepted by users. It is very useful to automate these tests, as test automation 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 popularity of Android platform, the presented work focused on testing the GUI of Android apps. The paper presents a proposed system for GUI testing of Android apps with multiple activities, which analyzes the app under test (AUT), generates test cases based on certain event-based coverage criteria, and executes these test cases. The proposed system applies a model-based approach to capture the event-driven nature of Android apps. The model used in this approach is the event sequence diagram (ESD), which depicts the events for an app and the possible transitions between them. The proposed system collects the IO/Clickable views of each activity in the AUT and their events. Then, it generates an ESD for each activity, combines all ESDs in one ESD and uses it to generate a set of event sequences according to the specified event-based criteria. For each event sequence, the system generates a test script and a corresponding Robotium test class, then executes the AUT with it. The paper also presents a case study to illustrate the use of the proposed system for testing a simple Android app with three activities, and the results of the experiments that have been conducted to demonstrate the system's ability to expose GUI errors that may occur in Android apps.

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 presents background on Android app's Activities and the Robotium framework. Sections 4 and 5 describe the proposed GUI testing approach for Android apps, and the supporting system, respectively. Section 6 presents the case study. Section 7 presents the experimental results. Section 8 presents the conclusion of this work.

2. RELATED WORK
This section presents a review of related research in the area of model-based GUI testing of Android apps. Model-based GUI testing of Android apps is one of test input/event generation approaches for Android app testing.

Amalfitano et al. [1] presented a technique for rapid crash testing and regression testing of Android apps. It 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. [2] presented AndroidRipper, an automated technique that 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. [3] presented a grey-box approach and a tool, for automatically extracting a model of a given mobile app. They perform static analysis to extract the events of the app GUI, then, dynamic crawling to reverse-engineer a model of the app, by exercising these events on the running app. Azim and Neamtiu [4] presented Android App Explorer (A3E) that allows Android apps to be explored while running on actual phones. 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 that permits fast, direct exploration of activities. They also developed another exploration strategy that mimics user actions for exploring activities and their constituents. Choi et al. [5] proposed an automated technique, called SwiftHand, which uses machine learning to learn a model of the app during testing, then 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. [6] presented MobiGUITAR, 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, enables automatic generation of test cases. Su et al. [7] presented Stoat, which uses dynamic analysis enhanced by a weighted UI exploration strategy and static analysis to reverse engineer a stochastic model of the app GUI interactions, then 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.

The authors have previously presented a model-based approach for UI testing of Android apps that have only single activity [8]. The approach presented in this paper extends the authors' previous approach to test the UI of Android apps with multiple activities. The proposed approach differs from the reviewed approaches in the following aspects: (1) it builds a simple model, ESD, to represent the events in the UI of each activity and possible transitions between them, combines all ESDs in one ESD 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 filtering out any sequence that is a subsequence of another one, and any sequence that includes unacceptable event subsequences; (4) it automatically generates test scripts from event sequences and converts them to test classes; and (5) it utilizes the Robotium Test Framework features to extract the AUT activities' views and related information, and to execute the generated test classes.

3. BACKGROUND

Activities are the main components of an Android app, which dictate the UI and handle the user interaction with the mobile device screen [9]. An activity represents a single screen with a UI. An app may have more than one activity, which can interact with each other. The one, which is presented when the app is launched, is called the main activity. The UI for each activity of an app is defined using a hierarchy of View and ViewGroup objects. Each view group is an invisible container that arranges child views, while the child views may be input controls or other widgets that draw some part of the UI. Input controls are the app UI interactive components. Android provides a wide variety of controls, such as EditText, TextView, Button, RadioButton, CheckBox, 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, a corresponding *Event Handler* is called to perform any required task.

The proposed system 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. 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 [10]. 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.

4. THE PROPOSED ANDROID APPS UI TESTING APPROACH

The proposed approach for testing the UI of Android apps with multiple activities is described as follows: Firstly, the AUT is statically analyzed to identify its activities and the views within each activity with their events. Then, testing is performed in two levels: activity level and app level. In the *activity level testing*, each activity is tested separately to verify whether it works as expected. Then, in the *app level testing*, the whole app is tested to verify whether all of its activities can work together to complete the desired functions. In this level, each activity is treated as a trusted unit as it has successfully passed the activity level testing. An execution path of the app is represented by a sequence of these trusted units.

Each of these testing levels consists of two phases: *Modeling phase* and *Test Evaluation phase*. In the modeling phase, a model is built for each activity/app to be used in generating test cases for the UI testing of the activity/app, while in the test evaluation phase, event-based coverage criteria are employed to determine whether the UI of the activity/app has been adequately tested by the generated test cases.

The possible execution paths in an activity/app UI are represented by a model called the *Event Sequence Diagram* (ESD) [11], which is based on the Finite State Machine model. In an ESD, each node represents an event, while a state transition is determined based on how the current node is responding to inputs. An ESD is built for each activity, then the ESDs of all activities are combined to build an App ESD.

An ESD D is a 2-tuple <N, E> where:

N is a set of nodes representing all the events for an activity/app. Each node $n \in N$ represents an event in D.

$E \subseteq N \times N$ 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 whole app, in the app level testing, based on certain event-based coverage criteria.

In order to measure the test adequacy of test cases, Memon [12] has defined two sets of event-based coverage criteria: intra-component criteria for events within a component; and inter-component criteria for events across components. In this work, these criteria were adapted for Android apps, and called: *Intra-activity criteria* and *Inter-activity criteria*, respectively. The first criteria are employed in the test evaluation phase of the activity level testing, while the second criteria are employed in the test evaluation phase of the app level testing. These criteria were defined as follows [8]:

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.

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.

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 approach to test the UI of an Android app: 1) Identify the app activities and build the corresponding ESDs; 2) Using the activities ESDs, construct the App ESD; 3) Generate test cases according to the defined coverage criteria; 4) Execute the test cases; 5) Analyze and evaluate the execution results.

5. THE PROPOSED ANDROID APPS UI TESTING SYSTEM

This section describes the proposed Android apps UI testing system that implements the proposed UI testing approach, described above. Figure 1 shows the steps that are followed by the system to generate and execute test cases for the AUT activities. The system 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. The system builds an ESD for each activity, and generates test cases based on the ESDs of the AUT and the coverage criteria, described in Sec. 4. The system takes as input the AUT, and produces as output: UI event sequences, Executable test cases, Criteria coverage report, and Test results report.

Generate_and_Run_Test_Cases Algorithm, shown in Figure 2, implements the steps shown in Figure 1. In this algorithm, three data structures are created for each activity act_i: Event list EL_i, which contains the activity's IO/clickable views with their events; Event Index List EIL_i, which contains for each view its index in EL_i, type, text, and id; and Event Sequences List SEQ_i, which contains all possible acceptable event sequences of the activity.

In this algorithm, the For loop (lines 1-14) performs the following actions for each activity act_i of app: Creates a ***Solo*** object, uses it to identify the current activity act_i and detects act_i's views and related information, which includes the view's type, event, text and id (lines 3-5); selects only IO/clickable views, saves the text of each view with its event in the Event List EL_i, and generates the Event Index List EIL_i (lines 6-7); then, in the inner loop (lines 8-13), for each event $e \in EL_i$, generates all possible acceptable sequences of e with all other events in EL_i, using the procedure ***Generate_Event_Sequences***, shown in Figure 3, and stores them in the Event Sequences List SEQ_i.

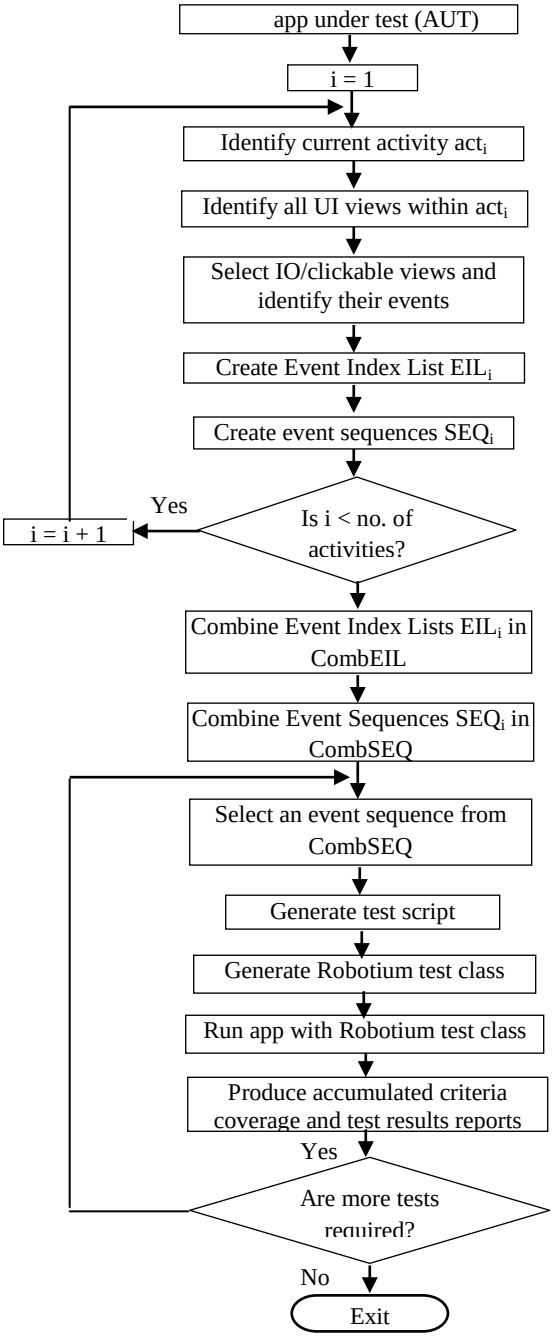


Fig 1: The steps of the proposed GUI testing approach for Android Apps with multiple activities

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 unacceptable event subsequences. Then, the algorithm (lines 14-15) combines all EIL_i's, using the procedure CombineEventIndexLists, shown in Figure 4, to get CombEIL, and combines all SEQ_i's, using the procedure ***CombineEventSequences***, shown in Figure 5, to get CombSEQ.

Generate_and_Run_Test_Cases Algorithm
Input: *app*, the AUT
Output: Test classes, Test results report, and Criteria coverage report
Begin
1. For each activity $act_i \in app$ (i = 1 … No. of activities)
2. Begin
3. Create a Solo object, *solo*.
4. Identify the current activity act_i in *app*, by using the method solo.getCurrentActivity().
5. Detect all UI views in act_i, by using the method solo.getCurrentViews().
6. Select from the detected views, only IO/clickable views and save the text of each view with its event in the event list EL_i.
7. Generate the Event Index List EIL_i, which contains for each event its index in EL_i, type, text, and id.
8. SEQ_i = [ ] // Initialize Event Sequences // List for activity act_i
9. For each event e $\in EL_i$
10. Begin
// generate all possible acceptable
// sequences of e with all other events
// in EL_i and store them in S_e
11. S_e = Generate_Event_Sequences(e, EL_i)
12. Add S_e to SEQ_i
13. End For
14. End For
15. CombEIL = CombineEventIndexLists() // Combine all EIL_i together
16. CombSEQ = CombineEventSequences() // Combine all SEQi together
17. testComplete = false
18. While not testComplete
19. Begin
20. Select next event sequence s ∈ CombSEQ
21. Generate_Test_Script(s, CombEIL) → testScriptFile
22. Create_Test_Class(testScriptFile) → Robotium test class testClassFile
23. add testClassFile to *app*
24. Run *app* with the Robotium test class
25. Produce accumulated Criteria Coverage Report and Test Results Report
26. If no more tests are required Then
testComplete = true
27. End While
End.

Fig 1: Generate_and_Run_Test_Cases Algorithm

Finally, the algorithm (in lines 17-24) repeats the following actions while more tests are required: selects an event sequence s ∈ CombSEQ; generates a test script for it, by using CombEIL and the procedure ***Generate_Test_Script***, shown in Figure 6. The test script includes, for each event in s, a line that contains the type of the corresponding view, its text and id. Then, it generates, from the generated test script, a Robotium test class, using the procedure ***Create_Test_Class***, shown in Figure 7, and adds it to *app*; executes *app* with the Robotium test class, and 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 test coverage criteria, to decide whether more tests are required or not.

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

Fig 2: Generate_Event_Sequences Procedure

Procedure CombineEventIndexLists()
Input: Event Index Lists EIL_i of all activities act_i in app (i = 1 … No. of activities)
Output: Combined Event Index List CombEIL
Begin
1. CombEIL = [ ]
2. For each activity $act_i \in app$ (i = 1 … No. of activities)
3. Begin
4. Add EIL_i to CombEIL.
5. End For
6. Return CombEIL
End.

Fig 4: CombineEventIndexLists Procedure

In procedure ***CombineEventSequences***, the For loop (lines 1-9) checks, for each activity, acti, other than the main activity, whether any of its event sequences, s_{ij}, does not end with BACK event (bk_i), and if so, it appends bk_i to that sequence. This ensures that, during execution, each sequence returns back to the main activity. Next, the For loop (lines 10-22) combines the event sequences of the main activity with event sequences of the other activities that can be reached from it. The presented automated GUI testing system has been 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.

Procedure CombineEventSequences()
Input: Event sequences list mainSEQ of *main_activity* of *app*
Event sequences lists SEQ_i of all other activities act_i in *app* (i = 2 … No. of activities)
Output: Combined event sequences CombSEQ
Begin
1. For each activity $act_i \in app$ (i = 2 … No. of activities)
2. Begin
3. For each sequence $s_{ij} \in SEQ_i$ (j = 1 … No. of sequences in SEQ_i)
4. Begin
5. If s_{ij} does not end with BACK event (bk_i) Then
6. Append bk_i to s_{ij}
7. End If
8. End For
9. End For
10. CombSEQ = [ ]
11. For each sequence s ∈ mainSEQ
12. Begin
13. If s includes an event e that triggers another activity act_i
14. pos = index of e in s
15. Randomly select a sequence s_{ij} from SEQ_i
16. Set s' = s
17. Insert s_{ij} at position pos+1 in s'
18. Add s' to CombSEQ
19. Else
20. Add s to CombSEQ
21. End If
22. End For
23. Return CombSEQ
End.

Fig 5: CombineEventSequences Procedure

Procedure **Generate_Test_Script (**s, EIL**)**
Input: an event sequence s
Event Index List EIL
Output: A test script file for the event sequence s, *testScriptFile*
Begin
1. For each e ∈ s
2. Begin
3. Get the view type that corresponds to event e, with its text and id, from EIL
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.

Fig 6: 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.

Fig 7: Create_Test_Class Procedure

The system provides users with the GUI interface shown in Figure 8. The system interface consists of:: seven buttons: "Browse", "Load AUT and Get Views and Sequences", "Generate Test Script, Test Class", "Run test class", "Get Number of Nodes", "Get Number of Edges", and "Generate Report"; three EditTextBox; controls, two TextView controls and one ListBox control. Firstly, the user selects an app for testing by clicking "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 system generates all possible acceptable event sequences of views. Next, a cycle starts: when the user clicks "Generate Test Script and Test Class" button, the system 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 system asks the

user whether he/she wants to continue, if the answer is no, the system stops, otherwise, the system allows the user to do more tests by clicking "Generate Test Script and Test Class" button, which repeats the above cycle, (see Figure 1). When the user clicks "Generate Report" button, the system loads the file that contains the executed paths of the ESD of the AUT and generates the coverage report.

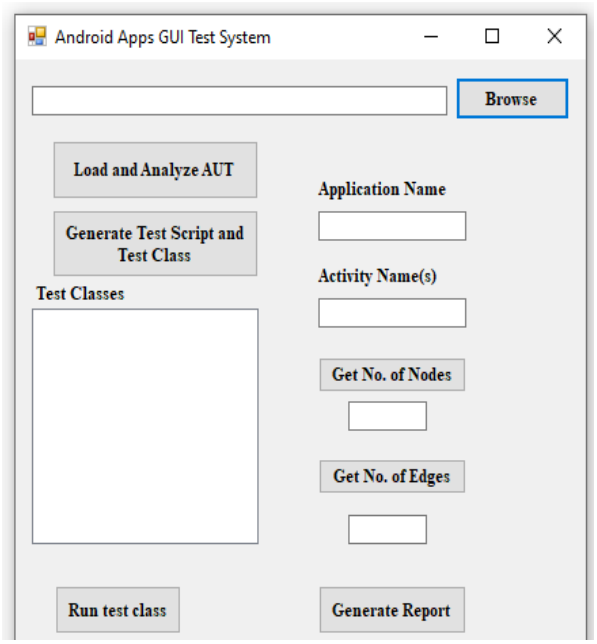


Fig 8: The interface of the proposed system

6. CASE STUDY

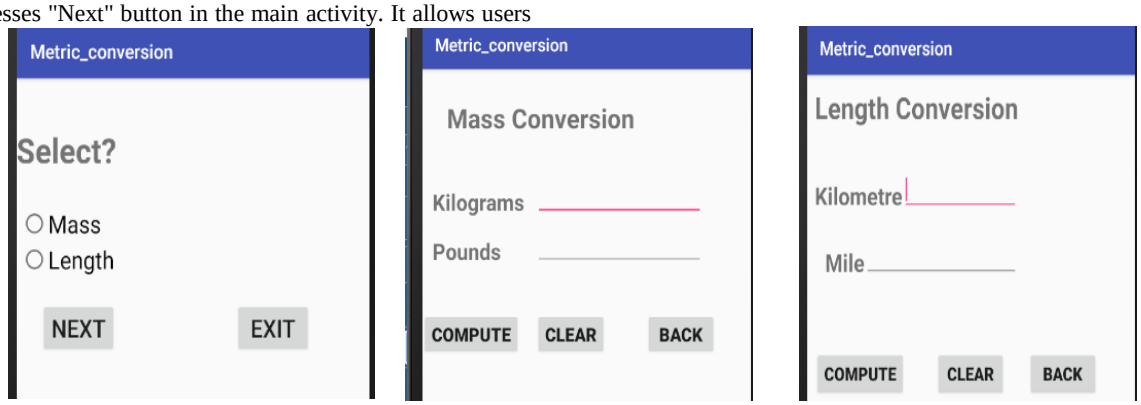
This section presents an example of using the proposed approach and system for testing a simple Android app called Metric_Conversion. It allows users to convert Kilograms to Pounds or Kilometers to Miles and vise-versa. As shown in Figure 9, this app has 3 activities. The main activity, shown in Figure 9(a), includes a TextView control that displays the message "Select?", 2 Radio buttons ("Mass" and "Length"), and 2 buttons ("Exit" and "Next"). It allows users to select either Mass units conversion or Length units conversion. The second activity, shown in Figure 9(b), includes a TextView control that displays the title "Mass Conversion", 2 TextView controls that display the labels "Kilograms" and "Pounds", 2 EditText controls, and 3 buttons ("Compute", "Clear" and "Back"). It appears when the user selects "Mass" Radio button and presses "Next" button in the main activity. It allows users to enter a mass amount in Kilograms/Pounds in the EditText control labeled "Kilograms"/"Pounds", then click "Compute" button to convert the input amount to Pounds/Kilograms and display it in the EditText control labeled "Pounds"/"Kilograms", respectively. The third activity, shown in Figure 9(c), is similar to the second one except that it is used for length units conversion. It appears when the user selects "Length" Radio button and presses "Next" button in the main activity. It allows users to enter a length amount in Kilometers/Miles in the EditText control labeled "Kilometers"/"Miles", then click "Compute" button to convert the input amount to Miles/Kilometers and display it in the EditText control labeled "Miles"/"Kilometers", respectively. In the second and third activities, pressing "Clear" button clears the contents of the EditText controls, and pressing "Back" button causes the app to return to the main activity. In the main activity, pressing "Exit" button terminates the app.

The system detects the IO/clickable views and saves the text of each view with its event in the events list, L, as shown in Table 1. Figure 10 shows the events index list, IL, which contains, for each event, its index in L, its type, text, and id. Figure 11 shows the corresponding ESD. Using the list L and the ESD, the system generates all possible acceptable sequences of views. In this example, only the interactions between the main activity and the second activity are considered, as the third one is similar.

Table 2 shows some of the generated test cases (event sequences). For each sequence the system generates a test script as the one shown in Figure 12, which corresponds to the event sequence [Idle-1-3-16-13-15-Idle-4] (Test case T7). 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 system generates a Robotium test class for the generated test script, as shown in Figure 13, and adds it to the AUT. Finally, the app with the test class is executed.

Figure 14 shows part of the Criteria Coverage Report produced by the system for the test cases shown in Table 2. It consists of two parts:

Part (I): Intra-Activity Criteria Coverage. It shows for each test case: the event sequence, the Event Coverage that includes: the newly covered events and the accumulated event coverage percentage, the Event-Interaction Coverage that includes: the newly covered edges and the accumulated event-interaction coverage, and Length-n Event-sequence Coverage;



(a) Main Activity **(b) Second Activity** **(c) Third Activity**

Fig 9: The Metric_Conversion app UI

```
0, TextView, Select?, tview
1, RadioButton, Mass, rr1
2, RadioButton, Length, rr2
3, Button, Next, button
4, Button, Exit, b1
10, TextView, Mass Conversion, tt
11, TextView, Kilograms, e3
12, TextView, Pounds, e2
13, Button, Compute, m1
14, Button, Clear, c8
15, Button, Back, a6
16, EditText, , e33
17, EditText, , e22
18, TextView, Length Conversion, t
19, TextView, Kilometre, a3
20, TextView, Mile, a2
21, Button, Compute, m1
22, EditText, , a33
23, EditText, , a22
24, Button, Clear, button2
25, Button, Back, button3
```

Fig 10: The Event Index List of the example app

Part (II): Inter-Activity Criteria Coverage. It shows Invocation Coverage that includes only one event (Event 3), which starts the second activity, Invocation-Termination Coverage that includes all length-2 sequences from an event to the main activity termination event (Event 4) and from an event to the second activity termination event (Event 15), and Length-n Event-Sequence Coverage that includes all length-n event sequences that start with an event in the main activity and end with an event in the second activity.

7. EXPERIMENTS

This section presents the results of the experiments, which were carried out to demonstrate the error exposing ability of the proposed system and the effectiveness of the GUI testing coverage criteria employed in it. The materials of the experiments were 10 Android apps. In these experiments, different errors were seeded in each app one at a time. In selecting the errors to be seeded, GUI errors were only considered, i.e., those errors that are manifested on the visible GUI at some point of time during the app's execution. The seeded errors are classified into three categories, *Input Errors*, *Code Errors* and *Android App Errors*. The *input errors category* includes errors that may occur by the user during his/her interaction with the app. The *code errors category* includes errors that may occur, by the developer, in the code of the UI controls event handlers. The *Android app errors category* includes errors that may occur in the GUI of android apps [13]. Table 3 describes the types of seeded errors and their frequency in the experiments, and Figure 15 shows the frequency of seeded errors in each error category.

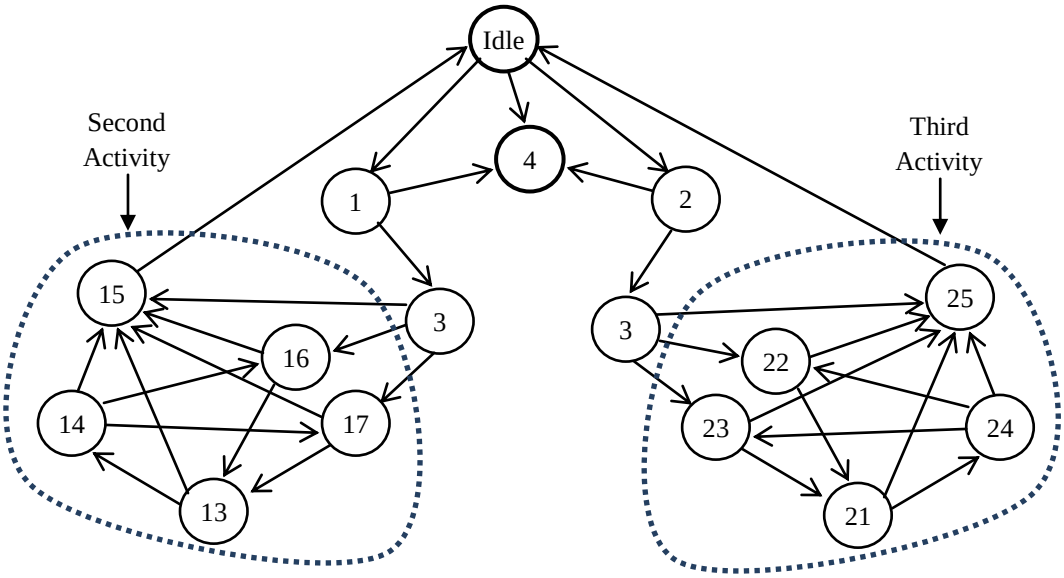


Fig 11: The ESD of the GUI of the example app (15 nodes and 33 edges)

Table 1. Event List of the example app

Index	Text	Event	Index	Text	Event
1	"Mass"	Click	16	" "	"enterText"
2	"Length"	Click	17	" "	"enterText"
3	"Next"	Click	21	"Compute"	"enterText"
4	"Exit"	Click	22	" "	"enterText"
13	"Compute"	Click	23	" "	"enterText"
14	"Clear"	Click	24	"Clear"	Click
15	"back"	Click	25	"back"	Click

Table 2. Some of the test cases generated for the main activity with the second activity of the example app

Test Case No.	Test Case
T1	Idle-1-3-15-Idle
T2	Idle-1-3-16-15-Idle
T3	Idle-1-3-16-13-14-16-15-Idle
T4	Idle-4
T5	Idle-2-4
T6	Idle-1-4
T7	Idle-1-3-16-13-15-Idle-4
T8	Idle-1-3-16-13-14-15-Idle
T9	Idle-1-3-17-15-Idle
T10	Idle-1-3-17-13-14-15-Idle
T11	Idle-1-3-17-13-14-16-15-Idle
…	

```
RadioButton,Mass,rr1
Button,Next,button
EditText,,e22
Button,Compute,m1
Button,Clear,c8
Button,Back,a6
Button,Exit,b1
```

Fig 12: The test script for the event sequence [Idle-1-3-16-13-15-Idle-4] (Test case T7)

```
public void setUp() throws Exception {
   solo = new Solo(getInstrumentation(), getActivity());
}
@Override
public void tearDown() throws Exception {
   solo.finishOpenedActivities();
   super.tearDown();
}
public void testRun() {
   solo.clickOnView(solo.getView("rr1"));
   solo.clickOnView(solo.getView("button"));
   EditText vEditText1 =
                    (EditText)solo.getView(R.id.e22);
   solo.enterText(vEditText1, "11 ");
   solo.clickOnView(solo.getView("m1"));
   solo.clickOnView(solo.getView("c8"));
   solo.clickOnView(solo.getView("a6"));
   solo.clickOnView(solo.getView("b1"));
}
```

Fig 13: The test class generated for the test script shown in Figure 12

In these experiments, each app was presented to the system to generate test cases for it, as described above. Then, errors were seeded one at a time in the app, and the system was used to execute the generated test cases on each erroneous version of the app. The system succeeded in detecting all the seeded errors, which demonstrates its effectiveness. This also demonstrates that the test cases generated to fulfill the adopted GUI testing coverage criteria were very effective. During test case execution, an error showed up, when an assertion was not verified, UI control event caused unexpected action or the action was not done, or incorrect output was produced.

```
Criteria Coverage Report
App Name: Metric_conversion
Activity Name: MainActivity, Second_Activity
ESD: 10 nodes, 18 edges
Part I: Intra-activity criteria coverage
Test Case No.: T1
        Event Sequence: Idle-1-3-15-Idle
        Event Coverage:     Newly covered events: Idle,1,3,15     Accumulated Event Coverage: 40 %
        Event-Interaction Coverage:     Newly covered edges: Idle-1,1-3,3-15,15-Idle
                                        Accumulated Event-Interaction Coverage: 22.22 %
        Length-n Event-sequence Coverage: n = 5
Test Case No.: T2
        Event Sequence: Idle-1-3-16-15-Idle
        Event Coverage:     Newly covered events: 16     Accumulated Event Coverage: 50 %
        Event-Interaction Coverage:     Newly covered edges: 3-16,16-15
                                        Accumulated Event-Interaction Coverage: 33.33 %
        Length-n Event-sequence Coverage: n = 6
Test Case No.: T3
        Event Sequence: Idle-1-3-16-13-14-16-15-Idle
        Event Coverage:     Newly covered events: 13,14     Accumulated Event Coverage: 70 %
        Event-Interaction Coverage:     Newly covered edges: 16-13,13-14,14-16
                                        Accumulated Event-Interaction Coverage: 50 %
        Length-n Event-sequence Coverage: n = 9
Test Case No.: T4
        Event Sequence: Idle-4
        Event Coverage:     Newly covered events: 4     Accumulated Event Coverage: 80 %
        Event-Interaction Coverage:     Newly covered edges: Idle-4
                                        Accumulated Event-Interaction Coverage: 55.56 %
        Length-n Event-sequence Coverage: n = 2
```

Fig 14: Part of the Test Coverage Report produced by the system for the test cases shown in Table 2

Test Case No.: T5
Event Sequence: Idle-2-4
Event Coverage: Newly covered events: 2 Accumulated Event Coverage: 90 %
Event-Interaction Coverage: Newly covered edges: Idle-2,2-4
Accumulated Event-Interaction Coverage: 66.67 %
Length-n Event-sequence Coverage: n = 3
Test Case No.: T6
Event Sequence: Idle-1-4
Event Coverage: Newly covered events: None Accumulated Event Coverage: 90 %
Event-Interaction Coverage: Newly covered edges: 1-4
Accumulated Event-Interaction Coverage: 72.22 %
Length-n Event-sequence Coverage: n = 3
Test Case No.: T7
Event Sequence: Idle-1-3-16-13-15-Idle-4
Event Coverage: Newly covered events: None Accumulated Event Coverage: 90 %
Event-Interaction Coverage: Newly covered edges: 13-15
Accumulated Event-Interaction Coverage: 77.78 %
Length-n Event-sequence Coverage: n = 8
Test Case No.: T8
Event Sequence: Idle-1-3-16-13-14-15-Idle
Event Coverage: Newly covered events: None Accumulated Event Coverage: 90 %
Event-Interaction Coverage: Newly covered edges: 14-15
Accumulated Event-Interaction Coverage: 83.33 %
Length-n Event-sequence Coverage: n = 8
Test Case No.: T9
Event Sequence: Idle-1-3-17-15-Idle
Event Coverage: Newly covered events: 17 Accumulated Event Coverage: 100 %
Event-Interaction Coverage: Newly covered edges: 3-17,17-15
Accumulated Event-Interaction Coverage: 94.44 %
Length-n Event-sequence Coverage: n = 6
Test Case No.: T10
Event Sequence: Idle-1-3-17-13-14-15-Idle
Event Coverage: Newly covered events: None Accumulated Event Coverage: 100 %
Event-Interaction Coverage: Newly covered edges: 17-13
Accumulated Event-Interaction Coverage: 100 %
Length-n Event-sequence Coverage: n = 8
Test Case No.: T11
Event Sequence: Idle-1-3-17-13-14-16-15-Idle
Event Coverage: Newly covered events: None Accumulated Event Coverage: 100 %
Event-Interaction Coverage: Newly covered edges: None
Accumulated Event-Interaction Coverage: 100 %
Length-n Event-sequence Coverage: n = 9
…
Part II: Inter-Activity Criteria Coverage
- Invocation coverage: Event 3
- Invocation-termination coverage:
Main Activity Termination: Idle-4, 2-4, 1-4
Second Activity Termination: 3-15, 16-15, 13-15, 14-15, 17-15
- Length-n event-sequence coverage

3-15	n=2	Idle-1-3-16-13-15	n=6
3-16-15	n=3	Idle-1-3-16-13-14-15	n=7
Idle-1-3-15	n=4	Idle-1-3-17-13-14-15	n=7
Idle-1-3-16-15	n=5	Idle-1-3-16-13-14-16-15	n=8
Idle-1-3-17-15	n=5	Idle-1-3-17-13-14-16-15	n=8

Fig 14: Part of the Test Coverage Report produced by the system for the test cases shown in Table 2 (Continued)

8. CONCLUSION

This paper presented a proposed system that applies a model-based approach for testing the GUIs of Android apps with multiple activities. The employed model is the ESD, which depicts the events for an app and the possible transitions between them. The proposed approach consists of two phases: Modeling Phase and Test Evaluation Phase. In the modeling phase, an ESD is built for each activity in the AUT, then all ESDs are combined to form the App ESD, which is used to generate test cases (event sequences). In the test evaluation phase, certain event-based coverage criteria are employed to measure the adequacy of the generated test cases for testing the GUI of the AUT.

The proposed system, which applies the proposed approach, analyzes the AUT, generates test cases, and executes these test cases. It collects the IO/Clickable views in each activity of the AUT and the associated events. Then, it generates the App ESD, and uses it to generate a set of event sequences according to the specified coverage criteria.

The system handles the event sequences explosion problem, and ensures the feasibility of event sequences. By considering these two issues, the number of generated sequences is significantly reduced.

Table 3. Types and frequency of seeded errors in 10 Android apps

Error Category	Error Type	Description	Frequency
Input Errors	Empty field	No input is entered in a TextBox	4
	Incorrect format	Input is entered in a TextBox in incorrect format	5
	No Item Selected	No RadioButton is checked, or no item is selected in a ComboBox	9
Code Errors	Wrong action (Incorrect assignment)	Wrong action for a UI control event due to incorrect assignment to a variable or UI control property in the event handler	1
	Wrong action (Incorrect statement)	Wrong action for a UI control event due to an error in a statement in the event handler	34
	Action not done (Missing statement)	An action of a UI control event is not done due to missing statement in the event handler	19
	Action not Done (Incorrect statement)	An action of a UI control event is not done due an error in any statement in the event handler	31
	Action not done (Incorrect assignment)	An action of a UI control event is not done due to incorrect assignment to a variable or UI control property in the event handler	12
	Wrong arithmetic operator	A computation includes a wrong arithmetic operator.	3
	Incorrect constant value	A computation includes an incorrect constant value.	1
Android App Errors	Intent payload replacement	Replace the actual value, in the key-value pair in intent.putExtra() method, by the default value	1
	Intent target replacement	Replace the target of an Intent with one of the possible classes within the same package of the current class	14
	OnClick event replacement	Replace an OnClick event handler with another compatible handler	33
	Button widget deletion	Delete a button from the XML of the UIlayout	25
	EditText Widget Deletion	Removes a EditText widget.	7
	TextView widget deletion	Remove a TextView widget	8
	Incorrect button caption	Change the caption of a Button or RadioButton widget	35
	Button widget switch	Switch the locations of two buttons on the same screen	16
Total			**258**

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

Next, the paper presented a case study that illustrated the use of the proposed GUI testing system in testing the UI of a simple Android app with three activities. Finally, it presented the results of the experiments that have been conducted to evaluate the system's ability to detect some types of errors that may occur in Android apps. The system succeeded in detecting all the seeded errors, which demonstrates the effectiveness of the system and the test cases generated to

fulfill the adopted GUI testing coverage criteria.

In the experiments, simple Android apps were used, but in future work, an empirical validation of the proposed approach will be performed by conducting experiments involving real world apps with larger size and complexity, with the aim of evaluating its effectiveness and scalability in a real testing context.

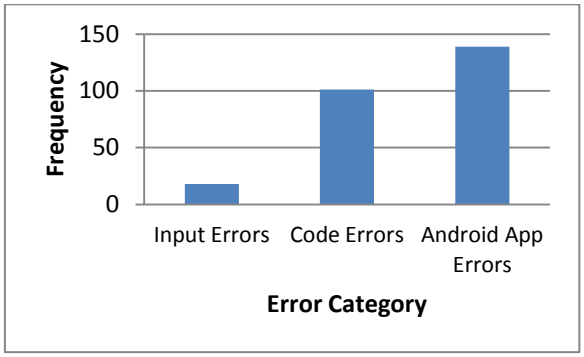


Fig 15: Frequency of seeded errors in each error category

9. REFERENCES

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

[2] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., Memon, A. M. 2012. Using GUI ripping for automated testing of Android applications. In: IEEE/ACM International Conference on Automated Software Engineering, ASE'12, 258–261. Essen, Germany.

[3] Yang, W., Prasad, M. R., Xie, T.: A grey-box approach for automated GUI-model generation of mobile applications. In: Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering, FASE'13, 250–265. Rome, Italy, (2013).

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

[5] Choi, W., Necula, G. C., Sen, K. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In: Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA'13, 623–640. Indianapolis, IN, USA.

[6] Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., Memon, A. M. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. IEEE Software, 32(5), 53–59.

[7] Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z. 2017. Guided, Stochastic Model-Based GUI Testing of Android Apps. Symposium on the Foundations of Software Engineering (ESEC/FSE'17), pp. 245-256. Paderborn, Germany.

[8] Girgis, M. R., Abdel Latef, B. A., and Akl, T. 2020. A GUI Testing Strategy and Tool for Android Apps, International Journal of Computing, 19(3), 355 – 364.

[9] Android - Application Components, https://www. tutorialspoint.com/android/android_application_ components.htm, last accessed 2018/8/12.

[10] Android user interface testing with Robotium – Tutorial, http://www.vogella. com/tutorials/Robotium/article.html, last accessed 2018/9/20.

[11] Li, P., Huynh, T., Reformat, M., Miller, J. 2007. A practical approach to testing GUI systems. Empirical Software Engineering, 12(4), 331–357.

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

[13] Deng, L., Offutt, J., Ammann, P., Mirzaei, N. 2017. Mutation operators for testing Android apps. Information and Software Technology, 81, 154-168.