A Crawling Approach of Hierarchical GUI Model Generation for Android Applications

Chien-Hung Liu and Ping-Hung Chen
Dept. of Computer Science and Information Engineering,
National Taipei University of Technology
Email: {cliu, t100599001}@ntut.edu.tw

Abstract

As the number of Android applications has increased dramatically, there is a rising concern about their quality and reliability. In particular, the rich GUI interactions supported by Android should be thoroughly tested in order to ensure if the behavior of an Android application is correct. However, manually create a GUI state model can be tedious and error-prone, especially for a nontrivial application. This paper proposes a crawler that can automatically generate the GUI state model of Android applications. Particularly, a hierarchical state model is employed to represent the intra- and inter-activity GUI behavior of Android applications in order to increase the model readability. Empirical experiments were conducted to evaluate the proposed crawler and the generated model. The results show that the state model generated by the crawler has a promising accuracy as compared to the model created manually. Besides, the hierarchical state model can improve the readability of model and, hence, facilitate the model validation.

Keywords: Android Crawler; Android GUI model; GUI testing;

I. Introduction

In recent years, the number of Android applications has increased dramatically. According to the statistics [1], there have been over 2,100,000 Android applications available on Google Play since April, 2016. In particular, Android applications have been widely used in all aspects of our lives, such as doing business, communicating with friends and families, searching information, and playing games. Thus, it is very important to ensure that the behaviors of Android applications meet their design specifications and won't cause business loss, system outage, or any inconvenience.

In particular, Android applications are usually operated by using touch screen, as compared with traditional desktop applications in which user interactions are performed by using the keyboard and mouse. The rich UI interactions supported by Android through varied gestures, such as tapping, dragging, sliding, pinching, and rotating, should be thoroughly tested in order to ensure the correctness of Android applications.

To test the GUI behavior of an Android application, a promising approach is to model the application's GUI behavior using a finite state machine where the states represent the possible GUI screens and transitions represent the events that change the properties of the screens. The test event sequences then can be derived systematically by traversing the state machine based on selected coverage criteria. However, manually extracting the GUI behavior and creating the corresponding state machine may require considerable efforts since the possible number of states and transitions for a nontrivial Android application can be large. Moreover, the resulting state model could be incorrect and incomplete due to human errors.

To ease the problems of manually creating a GUI state model for a nontrivial Android application, an alternative method is to build the GUI state model automatically using a crawler [2]. The basic idea of such a crawler is to visit each reachable GUI state automatically from a given initial GUI state by attempting to invoke a list of potential triggerable events systematically. The crawling process will continue until the event list becomes empty or the stopping criterion of the crawling is met.

The concept of crawler has been successfully applied to web applications to explore different web pages automatically for various purposes, including the generation of GUI test model [3]. However, the GUI structure and the event processing of Android applications are quite different from those of web applications. For example, an Android application allows users to swipe left or right to move from one fragment view to another (screen slide). The crawler has to try both directions of the swipe event in order to make the fragment view (screen) visible and to retrieve and analyze the GUI information of the view for further crawling. Thus, the development of Android crawler needs to take into account the GUI characteristics introduced by Android.

Moreover, for an application with a large number of states and transitions, the GUI state model generated by a crawler can be hard to understand and difficult to validate. To automatically extract the GUI behavior of Android applications and increase the readability of the generated model, this paper presents a hierarchical GUI model generation approach based on the crawler. Particularly, the proposed crawler can automatically extract the GUI behavior of an Android application and generate a GUI state model by taking into account the rich GUI events supported by Android. In addition, a

hierarchical state model is employed to represent to intra- and inter-activity GUI behavior of Android applications in order to reduce the model complexity and, hence, facilitate the model analysis and validation.

To evaluate the effectiveness of the proposed Android crawler and hierarchical state model, several experiments are conducted. The experimental results suggest that the GUI state model generated by the proposed crawler can be much more accurate as compared to the state model created manually. Besides, the readability of the hierarchical GUI state model is much improved as compared to traditional "flat" state machine.

The rest of the paper is organized as follows. Section II briefly reviews the related work. Section III presents the hierarchical GUI state model and the crawling algorithm for model generation. Section IV depicts the design and limitations of the crawler. Section V describes and discusses the experimental results. Section VI summarizes the conclusions and describes possible future work.

II. Related Work

Although crawlers have been studied for many years, most of the existing researches focused on the web indexing. Currently, there is a few studies on the crawlers for the GUI model generation of Android applications. The following briefly reviews several researches related to our work.

Wang et al. [4] describe several challenges of exploring the GUI of Android applications, including the identification of Android GUI components, generations of interacting GUI events, and the crawling algorithm to achieve high coverage. A tool, called DroidCrawler, has been implemented. Basically, the DroidCrawler downloads the GUI information of the target Android application using ADB (Android Debug Bridge) [5] and identifies the corresponding GUI components. It then simulates user actions using Monkey [6] by triggering Key and GUI events related to the components of interest. A depth-first algorithm is presented to automatically explore the GUIs of target and generate a GUI tree where the nodes representing the GUIs and the edges representing the trigger events between the GUIs. A case study is presented to illustrate the GUI coverage of the DroidCrawler.

Amalfitano et al. [7], propose a GUI crawling method that can be used for crash testing and regression testing of Android applications. The proposed method basically explores the GUI of an Android application by automatically simulating real user events on the user interfaces and builds a GUI tree model. Each node of the tree represents a user interface of the application and each edge represents the event causing the change between the user interfaces. Test cases then can be derived from the tree model systematically. A supporting tool, called AndroidRipper [8], has been implemented and a case study is presented to illustrate the effectiveness of the method. Moreover, based on the proposed method, [9] presents a toolset to automate the generation of JUnit test cases for testing the GUI of Android applications.

Takala et al. [10] present a model-based testing (MBT) method for testing the GUI of Android applications. They adapt state machine to model the Android GUI. In particular, each model abstracts an individual view of the GUI with two separate state machines: an action machine and a refinement machine. The action machine describes the high-level functionality with action words and state verifications. The refinement machine implements action words and state verifications using keywords. Keyword-based test cases then can be generated from the model automatically through a supporting open source toolset, called TEMA. A case study is presented to demonstrate the effectiveness of the proposed method.

Yang et al. [11] propose an approach that combines both static analysis and dynamic crawling techniques to automate the GUI model generation for Android applications. Specifically, the approach extracts the set of user actions supported by each widget in GUI screens from the source code of the Android applications. The extracted user actions include the action registered to an event-listener or inherited from the event-handling method of an Android framework component. A dynamic crawler is then used to exercise the extracted actions systematically and generate a compact GUI model. A tool, called ORBIT, is implemented to support the proposed approach and experiments were conducted to demonstrate the efficiency of the proposed approach.

Machiry et al. [12] present a system, called Dynodroid, that can automatically generate relevant inputs to Android applications for the support of dynamic analysis and testing. The main principle of Dynodroid is an *observe-select-execute* cycle. In the observer stage, Dynodroid analyzes the widgets on the current screen and computes a set of relevant UI and system events. In the selector stage, Dynodroid will select an event to execute from the set of relevant events according to a randomized algorithm which supports three different selection strategies. In the executor stage, Dynodroid can execute the event chosen by the system automatically or provided by users manually. Experimental results indicate that, as compared with humans and Monkey, Dynodroid has satisfactory code coverage and is more efficient in generating input sequences.

Zhu et al. [13] present an approach called Cadage (Context-Aware Dynamic Android Gui Explorer) to generate a GUI model automatically for testing Android applications. Similar to [11], the basic test cycle of Cadage includes Inferencer, Selector, Executor, and Modeler which are responsible to extract fireable events of current GUI state, select an action event, execute the chosen event,

and construct the GUI model, respectively. Particularly, the goal of the approach is to explore the unexecuted events of the Android application under test as quickly as possible while constructing the approximate GUI model. To achieve this and solve the non-determinism problem introduced by the approximation of the model, a probabilistic selection algorithm that can increase the priority of unexecuted events is used when selecting an event to execute. Evaluation is provided to show the efficiency of the approach.

As compared with the aforementioned studies, our work focuses on the automatic generation of GUI state model for Android applications without source code. Moreover, to facilitate analysis and testing, the proposed approach aims to improve the readability and completeness of the GUI state model.

III. The GUI Model Generation Approach

This section describes the hierarchical GUI state model and the crawling approach used to construct the model automatically.

3.1 The Hieratical GUI State Model

An Android application usually consists of multiple activities that interact with each other. Each activity provides a container for UI widgets, such as buttons and text boxes. Users can interact with the application by navigating different activities using the UI widgets or physical keys of the mobile device. To represent the possible user interaction behavior of Android applications, a two-level hierarchical state model is employed. Specifically, the top level of the state model is called ATD (Activity Transition Diagram) that represents the navigations between the activities. The second level of the state model is called ASD (Activity Substate Diagram) that abstracts the state changes within an activity.

The ATD and ASD are finite state machines that can be represented as a 5-tuple FSM = (Q, Σ, q_0, δ, λ), where Q is a finite set of states, Σ is a finite set of UI and key events, q_0 is the initial state, δ:$Q\times\Sigma\rightarrow Q$ is a set of transitions, and λ:$Q\rightarrow\Sigma$ is a mapping function. Thus, let *ATD* be a FSM and *ASDs* be a finite set of FSMs. then the proposed GUI state model is formally defined as a triple G = (*ATD*, *ASDs*, μ), where μ:$Q\rightarrow$ *ASDs* is a mapping function that associates each state $q\in Q$ with a FSM in *ASDs*.

For illustrating the ATD and ASD, let's consider a trivial Android application shown in Fig. 1. The application has three activities: ItemList, About, and Setting. The GUI of the ItemList activity contains two buttons and a group of radiobuttons. The GUIs of two others contain only one button, respectively. The application allows users to navigate between the activities by clicking on the buttons. Such GUI behavior is modeled in the ATD as shown in Fig. 2, where the state of the ATD represents an activity, and the transition of the ATD represents an event between the activities.

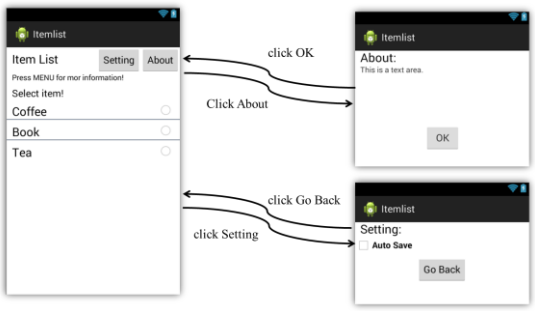

Fig. 1 The activity snapshots of a trivial Android app

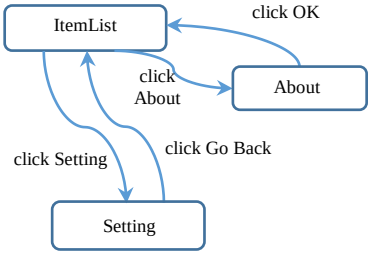

Fig. 2 The ATD of the trivial Android app

Moreover, in the ItemList activity, users can interact with the radiobuttons and change the GUI state as shown in Fig. 3. However, this won't change the activity of the application. Such GUI behavior is abstracted in the ASD shown in Fig. 4, where the state of the ASD represents the values of a set of GUI properties within an activity, and the transition of the ASD represents an event between the states.

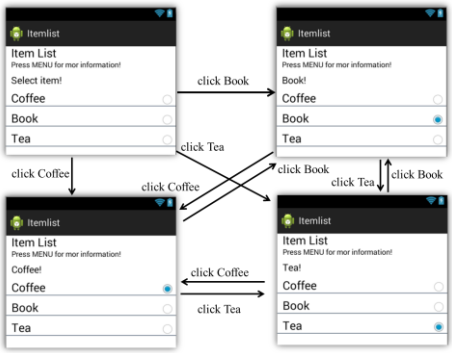

Fig. 3 The snapshots of interacting radio buttons

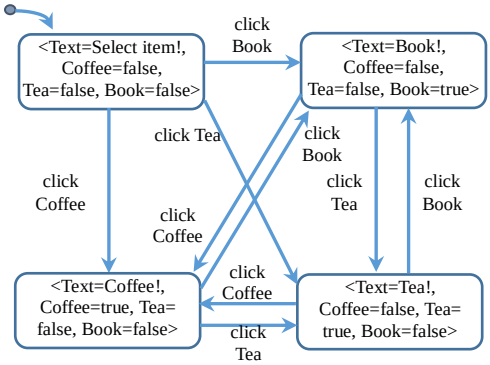

Fig. 4 The ASD corresponding to ItemList activity

3.2 The Crawling Algorithm of Model Generation

To generate the proposed hierarchical GUI state model automatically, a dynamic analysis approach is employed. The approach will crawl the possible GUI states of an Android application and generate the corresponding ATD and ASDs dynamically. Fig. 5 shows the crawling algorithm. The algorithm is based on the breadth-first search (BFS) traversal strategy to explore the possible GUI states of an Android application starting from the main activity.

```
CrawlingAlgorithm (Input: config, App; Output: ATD, ASD)
begin
  stoppingCriteria ← config
  start(App)
  a.activity ← App.MainActivity
  task.state ← s ←getGUIState()          //get current GUI state
  task.eventList ← all fireable events of s
  a.taskList ← task
  activityList ← a
  while activityList ≠ null or isFalse(stoppingCriteria) do
    a_i ← get an activity from activityList
    while a_i.taskList ≠ null do
      t_j ← get a task tk from a_i.taskList where tk.state = s
      while t_j.eventList ≠ null do
        e_k ← get an event from t_j.eventList
        fire e_k
        s ← getGUIState()                //get current GUI state
        if isChangeActivity(s) then
          if isNewATDState(s) then
            a.activity ← the new activity
            t.state ← s
            t.eventList ← all fireable events of s
            a.taskList ← t
            add a to activityList
          end if
          update ATD(e_k, s)             // update the ATD
          s ← t_j.state                  // backtrack to previous state
          restart(App) and forwardCrawling(a_i, s)
        else
          if isNewASDState(s) then
            t.state ← s;
            t.eventList ← all fireable events of s
            add t to a_i.taskList
            update ASD(a_i, e_k, s)      // update the ASD of a_i
            s ← t_j.state                // backtrack to previous state
            restart(App) and forwardCrawling(a_i, s)
          end if
          update ASD(a_i, e_k, s)        // update the ASD of a_i
        end if
        remove e_k from t_j.eventList
      end while
      remove t_j from a_i.taskList
      s ← fromState(t_j.state)           // backtrack to parent state
      restart(App) and forwardCrawling(a_i, s)
    end while
    remove a_i from activityList
  end while
  return (ATD, ASD)
end
```

Fig. 5 The crawling algorithm of model generation

In the crawling algorithm, each activity has a list of exploring tasks (i.e., taskList). An exploring task is composed of a GUI state and a list of events that can be executed from the state. When the crawler visits the GUI of an activity at the first time, the GUI state is identified and a list of fireable events associate with that GUI state is obtained. With the GUI state and events, a task is then created and added into the taskList of the activity. The crawler then explores the possible GUI states by getting a task from the taskList and firing the task's events iteratively.

After firing an event, if the current activity is changed to an activity unexplored before, a new activity is created and added into the list of activities to be explored (i.e., activityList). Moreover, if the executed event changes the activity, this means that there is an inter-activity GUI state change and the ATD of the application will be updated accordingly. To backtrack to previous GUI properly, the crawler will re-start the application and forward traverse to the previous GUI state from the initial state.

If the activity is not changed after firing an event, but a new GUI state occurs, then a task is created based on the new GUI state and is added into the taskList of the current activity. This means that there is an intra-activity GUI state change. Thus, the ASD of the current activity is updated accordingly. The crawler will backtrack if a new ASD state is encountered or all the events of the current task have been fired. The crawling process continues until the events of each task for every activity have been executed or the stopping criteria are satisfied. Currently, the supported stopping criteria include the timeout limit of crawling, the depth of the BFS, and the number of GUI states explored.

IV. Design and Implementation of the Crawler

To support the proposed approach, a tool called Android Crawler is developed. Fig. 6 shows the system architecture of the tool that consists of four major subsystems, including the GUI Extractor, Crawler Controller, Event Executor, and Model Builder. The GUI Extractor is used to extract and analyze the GUI information of an Android application. It will identify the widgets of the GUI and compute a list of fireable events. The Crawler Controller will examine the current GUI state, control the traversal of the crawler, and select an event to execute The Event Executor is responsible to fire the selected event using Monkey with a configurable default think time. The Model Builder will construct the ATD and ASDs for the application dynamically.

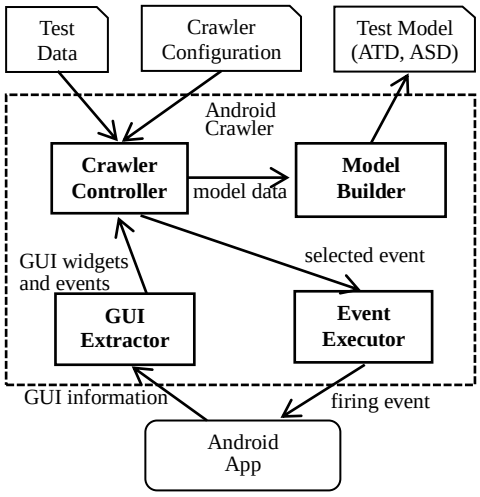

Fig. 6 The system architecture of Android Crawler

Notice that to obtain a list of fireable events for a GUI state, the crawler will dump the GUI

information using Uiautomator [14] and analyze the widgets of the GUI. The fireable events corresponding to each widget are identified, including click, double-click, long-click, swipe, scroll, edit-text, menu key, and back key events. The screen coordinates required to fire the events are also computed. The number of coordinates and how to compute the coordinates can be dependent of the event types. For example, the scroll event requires the beginning and ending coordinates. The gesture directions of the event can scroll from right to left, from left to right, from bottom to top, and from top to bottom. The event coordinates can be computed using the top-left and bottom-right corners of the screen.

V. Experiments

To assess the correctness and completeness of the generated GUI state model, we compare the GUI models automatically generated by the crawler with the referenced models manually generated by humans. To minimize the bias, five applications are chosen for this evaluation. The referenced GUI models of these five applications are created by three software engineer professionals manually. They interact with the applications and produce their own GUI models individually. They then review the models and derive the referenced models together according to the best of their understanding.

Table 1 shows the results of the accuracy of the GUI models generated by the crawler as compared with the referenced models. The results indicate that the average accuracies of GUI states and transitions are 87.7% and 64.4%, respectively. The crawler seems to have a better state coverage than transition coverage. The possible reasons could be that the events currently supported by the crawler are limited to some UI and key events. The transitions introduced by other types of events, such as system events, are unable to cover at the present time.

Table 1 The accuracy of the Android crawler

Apps	**Crawling Time (hh:mm:ss)**	**Accuracy of Transition (%)**	**Accuracy of State (%)**
Notepad [16]	1:09:54	54.0%	100.0%
Taiwan Receipt Lottery [15]	0:07:58	69.2%	80.0%
QR Code Scanner [17]	0:33:28	76.7%	91.7%
Volume Booster Pro [18]	0:23:48	90.9%	100.0%
Magnifier [19]	0:08:47	31.3%	66.7%
Average		64.4%	87.7%

To evaluate the efficiency of the crawler for model generation, the average time required to construct a GUI model manually and to generate a GUI model using the crawler are computed. To minimize the bias, the participants of this experiment are divided into two groups. Each group has five participants. The group 1 will first create the GUI model manually and then generate the model automatically using the crawler. On the other hand, the group 2 will first generate the model automatically using the crawler and then create the GUI model manually. The subject of the experiment is Notepad [16] and both groups have no previous experiences in using the Notepad application.

The average results of the ten participants from the two groups are shown in Table 2. Although the results indicate that the average time spent by human is far less than that required by the crawler, the model created by human is also much more inaccurate than the model generated by the crawler. The average state and transition accuracies of the manually created model are respectively 48.2% and 7.6% which are outperformed by the accuracies of 100.0% and 54.0% obtained from the automatically generated model.

The rationale behind these results may be because most of the participants did not thoroughly analyze the possible interacting events when creating the model. This is not surprising because it requires non-trivial efforts to manually identify the possible user interactions for a medium-sized application like Notepad. As a result, they miss a lot of potential GUI states and transitions. One observation from the GUI models manually created by the participants is that humans often overlook the key events and, hence, ignore many possible GUI states and transitions.

Notice that, in Table 2, the average time required to manually operate the crawler is 2 minutes and 21 seconds (2:21) which is less than the time (3 minutes and 26 seconds) needed to create the GUI model manually. This suggests that the model generation can be completely or largely automated using the proposed crawler although it would take more time to generate the GUI state model.

Table 2 The results of the efficiency case study

Generation of GUI state model	Crawling Time (hh:mm:ss)		Accuracy of Transition (%)	Accuracy of State (%)
Manual model creation	0:03:26		7.6%	48.2%
Automatic model generation with crawler	manual operation	automatic crawling	54.0%	100.0%
	0:02:21	1:10:53		

The results of Tables 1 and 2 suggest that the proposed crawler can extract a fairly large numbers of GUI states and transitions correctly. As compared with the model created manually, the GUI state model generated by the proposed crawler is more accurate. Moreover, the result of Table 2 also indicates that the time efficiency of the crawler would be acceptable if only the manual effort is considered.

VI. Conclusions and Future Work

This paper has presented an approach to automate the generation of GUI state model for Android applications based on the crawler. In particular, a hierarchical state model is proposed to represent the GUI behavior of Android applications for improving the model readability. The proposed model consists of an ATD and a set of ASDs which

can be used to depict the intra- and inter-activity GUI behavior, respectively. A crawling algorithm that can automatically generate the hierarchical GUI state model is described. A tool taking into account the Android GUI characteristics is developed to support the proposed approach. Several case studies were conducted to illustrate the effectiveness of the proposed crawler and the hierarchical GUI state model.

In the future, we plan to improve the efficiency of the proposed crawler. A possible improvement of the crawler efficiency is to minimize the number of restarting the Android applications which is required for backtracking to previous GUI state. Moreover, we plan to extend the crawler to support more types of events. Further, we also plan to enhance the algorithm of the crawler to improve its code coverage and the ability of GUI state abstraction.

Acknowledgment
This work was supported in part by the Ministry of Science and Technology, Taiwan, under the grant No. MOST 104-2221-E-027-009.

References

[1] AppBrain, [Online]. Available: http://www.appbrain.com/stats/number-of-android-apps. [Accessed April, 2016].

[2] Atif Memon, Ishan Banerjee, and Adithya Nagarajan, "GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing," *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, pp. 260-269.

[3] Ali Mesbah, Arie van Deursen, and Stefan Lenselink, "Crawling Ajax-based Web Applications through Dynamic Analysis of User Interface State Changes," *ACM Transactions on the Web*, Vol. 6, No. 1, March 2012, pp. 1-30.

[4] Peng Wang, Bin Liang, Wei You, Jingzhe Li, and Wenchang Shi, "Automatic Android GUI Traversal with High Coverage," *Proceedings of the Fourth International Conference on Communication Systems and Network Technologies (CSNT)*, Bhopal, April 2014, pp.1161-1166.

[5] Android Debug Bridge, [Online]. Available: http://developer.android.com/tools/help/adb.html. [Accessed June, 2015].

[6] Monkey, [Online]. Available: http://developer.android.com/tools/help/monkey.html. [Accessed June, 2015].

[7] Domenico Amlfitano, Anna Rita Fasolino, and Prfirio Tramontana, "A GUI Crawling-based technique for Android Mobile Application Testing," *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, Bhopal, March 2011, pp.252-261.

[8] Domenico Amlfitano, Anna Rita Fasolino, Prfirio Tramontana, Salvatore De Carmine, and Atif M. Memon, "Using GUI Ripping for Automated Testing of Android Applications," *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Essen, September 2012, pp.258-261.

[9] Domenico Amlfitano, Anna Rita Fasolino, Prfirio Tramontana, Salvatore De Carmine, and Gennaro Imparato, "A Toolset for GUI Testing of Android Applications," *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, September 2012, pp.650-653.

[10] Tommi Takala, Mika Katara, and Julian Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," *Proceedings of the Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST)*, Berlin, March 2011, pp.377-386.

[11] Wei Yang, Mukul R. Prasad, and Tao 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*, 2013, pp. 250-265.

[12] Aravind Machiry, Rohan Tahilizni, and Mayur Naik, "Dynodroid: An Input Generation System for Android Apps," *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp.224-234.

[13] Haowen Zhu, Xiaojun Ye, Xiaojun Zhang, and Ke Shen, "A Context-Aware Approach for Dynamic GUI Testing of Android Applications," *Proceedings of the IEEE 39th Annual Computer Software and Applications Conference (COMPSAC)*, Taichung, 2015, pp. 248-253.

[14] UI Automator, [Online]. Available: http://developer.android.com/tools/testing-support-library/index.html#UIAutomator. [Accessed June, 2015].

[15] Taiwan Receipt Lottery, [Online]. Available: https://play.google.com/store/apps/details?id=com.twecs&hl=zh-TW. [Accessed June, 2015].

[16] Notepad, [Online]. Available: https://play.google.com/store/apps/details?id=notepad.sisa.mobi&hl=zh-TW. [Accessed June, 2015].

[17] QR Code Reader, [Online]. Available: https://play.google.com/store/apps/details?id=tw.mobileapp.qrcode.banner&hl=zh-TW. [Accessed June, 2015].

[18] Volume Booster Pro, [Online]. Available: https://play.google.com/store/apps/details?id=com.volume.sound.manager&hl=zh-TW. [Accessed June, 2015].

[19] Magnify, [Online]. Available: https://play.google.com/store/apps/details?id=com.appdlab.magnify&hl=zh_TW. [Accessed June, 2015].