A Crawling Approach of Hierarchical GUI Model Generation for Android Applications

Chien-Hung Liu, Ping-Hung Chen

Department of Computer Science and Information Engineering, National Taipei University of Technology, Taiwan
{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 creating 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 for an Android application. 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 coverage as compared to the model created manually. The hierarchical state model can greatly improve the model readability to ease the GUI behavior analysis and validation for Android applications.

Keywords: Android crawler, Android GUI model, Android GUI testing

1 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, such as state or transition coverage. 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

*Corresponding Author: Chien-Hung Liu; E-mail:cliu@ntut.edu.tw
DOI: 10.3966/160792642018091905032

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 the 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 were conducted. The experimental results suggest that the GUI state model generated by the proposed crawler can be much more complete 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 2 briefly reviews the related work. Section 3 presents the hierarchical GUI state model and the crawling algorithm for model generation. Section 4 depicts the design and limitations of the crawler. Section 5 describes and discusses the experimental results. Section 6 summarizes the conclusions and describes possible future work.

2 Related Work

Although crawlers have been studied for many years, most of the existing researches focused on web crawling and its applications [4, 5]. Currently, there is a few studies on the crawlers of Android applications for GUI model generation. The following briefly reviews several researches related to our work.

Wang et al. [6] 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) [7] and identifies the corresponding GUI components. It then simulates user actions using Monkey [8] 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. [9], 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 Android Ripper [10], has been implemented and a case study is presented to illustrate the effectiveness of the method. Moreover, based on the proposed method, [11] presents a toolset to automate the generation of JUnit test cases for testing the GUI of Android applications.

Takala et al. [12] present a model-based testing (MBT) method for testing the GUI of Android applications. They use state machine to model the GUI of applications. In particularly, their 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. [13] 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. [14] 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. [15] present an approach called Cadage (Context-Aware Dynamic Android Gui Explorer) to generate a GUI model automatically for testing Android applications. Similar to [14], 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.

Choi et al. [16] propose a technique, called SwiftHand, that can generate input sequences automatically for testing Android applications. The technique combines model learning with testing in which a state-based model of an Android application is learned as testing is performed. From the learned model, test inputs are generated to explore unvisited GUI states of the application in order to achieve better coverage. Particularly, the proposed technique can guide the learning and testing to avoid restarting the applications so as to save testing time and to merge equivalent GUI states for reducing the search state space. The experimental results show that the technique can achieve branch coverage quickly than random and $\mathcal{L}^*$-based testing.

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 generate a visual GUI state model and improve the readability and completeness of the model. Table 1 shows the comparison of related work with the proposed approach in terms of the generation of GUI model, the crawling algorithm used to explore the GUI states, the selection of events for GUI exploration, and the support of event type and user input data.

3 The GUI Model Generation Approach

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

Table 1. The comparison of the related work

Approach	GUI Model Generation	Crawling Algorithm	Event Selection	Event Type	Input Data
Droid Crawler	GUI Tree	DFS	random	user event	Yes
Android Ripper	GUI Tree	DFS-based	random	user/ system event	Yes
MBT	Labelled state transition system	N/A	random	user event	Yes
ORBIT	FSM	FwdCrawl (DFS-based)	systematic	user event	No
Dynodroid	no support	N/A	random	user/ system event	No
Cadage	Labelled state transition system	BFS	priority, random	user event	No
Swift Hand	FSM	Learning-based	random	user/ system event	Yes
Proposed approach	Hierarchical state model	BFS	systematic	user event	Yes

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 $q\in Q$ of *ATD* with a FSM in *ASDs*.

For illustrating the ATD and ASD, let's consider a trivial Android application shown in Figure 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 ItemList and About activities or between the ItemList and Setting activities by clicking on the corresponding buttons.

The inter-activity GUI behavior of the trivial Android application can be modeled using ATD as shown in Figure 2, where the state of the ATD represents an activity, and the transition of the ATD represents an event between the activities.

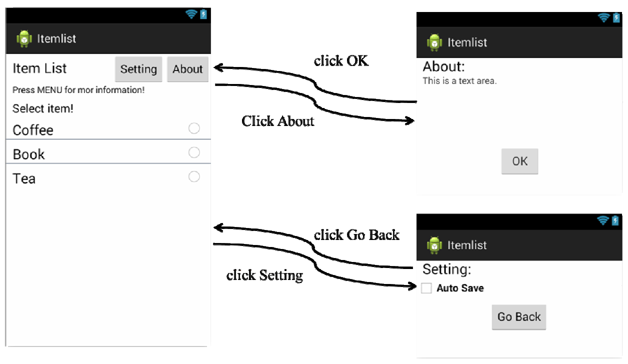


Figure 1. The activity snapshots of a trivial Android app

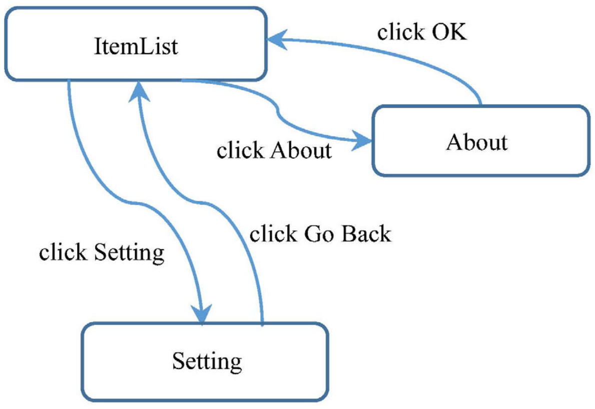


Figure 2. The ATD of the trivial Android app

Notice that in the ItemList activity users can click the corresponding radio buttons to select an item, such as coffee, book, or tea. The selection of item can change the GUI state of the trivial application as shown in Figure 3. Although clicking on different radio button can change the GUI state, it won't cause the activity to change. Thus, the trivial application remains at the ItemList activity. Such intra-activity GUI behavior is abstracted in the ASD shown in Figure 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.

3.2 The Crawling Algorithm of Model Generation

To generate the proposed hierarchical GUI state model automatically, a GUI crawling approach is employed. The approach will explore the possible GUI states of an Android application and generate the corresponding ATD and ASDs dynamically. Figure 5

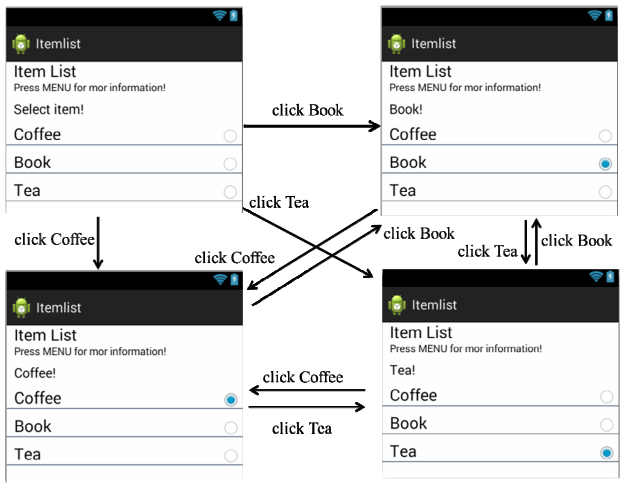


Figure 3. The snapshots of interacting radio buttons

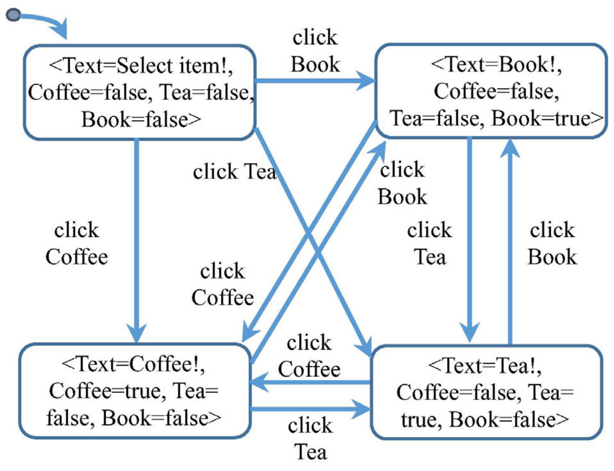

<Text=Select item!, Coffee=false, Tea=false, Book=false>
click Book
<Text=Book!, Coffee=false, Tea=false, Book=true>
click Tea
click Book
click Coffee
click Coffee
click Tea
click Book
<Text=Coffee!, Coffee=true, Tea= false, Book=false>
click Coffee
click Tea
<Text=Tea!, Coffee=false, Tea= true, Book=false>

Figure 4. The ASD corresponding to ItemList activity

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.

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 GUI state is identified and a new activity with an exploring task for the identified GUI state is created. This new activity is then added into the list of activities (i.e., activityList) and will be explored later. Moreover, if the event execution changes the activity, this means that there is an inter-activity GUI state change and the ATD of the application will be updated accordingly.

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

Figure 5. The crawling algorithm of model generation

Note that if the event execution results in a GUI state change, the proposed crawler will backtrack to previous GUI state in order to continue the crawling. To backtrack to previous GUI state properly, the crawler will restart the application and then forward traverse to the previous GUI state using the forwardCrawling(a_i, s) function in Figure 5. This function will change the GUI state of the application from its initial GUI state to the GUI state s of activity a_i (i.e., the previous or target GUI state) by firing a sequence of events after restarting the application.

If the event execution does not change the activity, but it leads to a new GUI state of the same activity, then an exploring 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 and the ASD of the current activity is updated accordingly. The crawler then restarts the application again and backtracks to its previous GUI state. However, if the event execution does not introduce a new GUI state, then the crawler simply updates the ASD based on the execution result.

If all the events of a task have been fired, the task will be removed and the crawler will restart the application and traverse to the GUI state of the next task to be explored. 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 or events explored.

4 Design and Implementation of the Crawler

To support the proposed approach, a tool called Android Crawler is developed. This section describes the architecture design and implementation of the tool and its current limitations.

4.1 The System Design and Implementation

Figure 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.

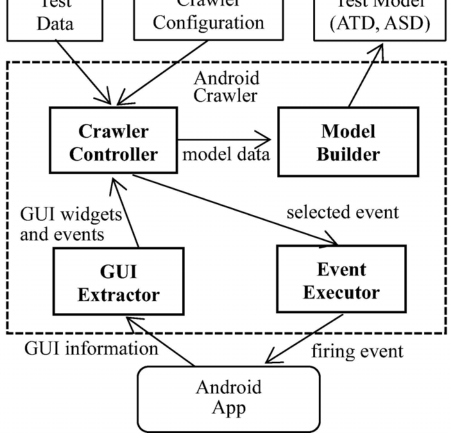


Figure 6. The system architecture of Android Crawler

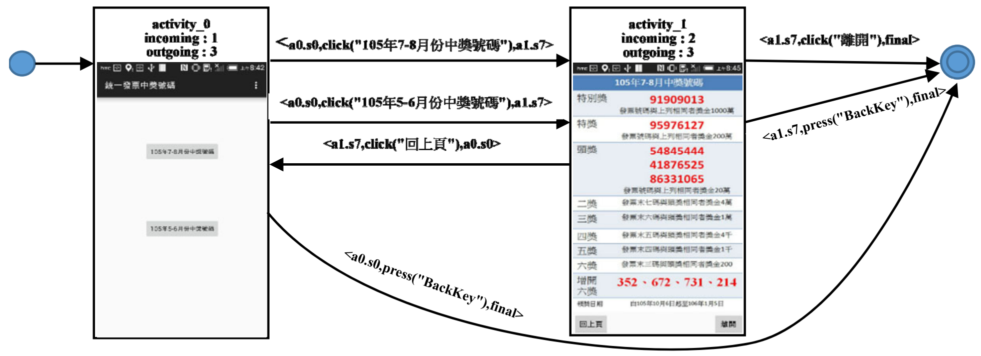
Notice that to obtain a list of fireable events for a GUI state, the crawler will dump the GUI information using Uiautomator [17] and analyze the widgets of the GUI. The fireable events corresponding to each widget are identified, including click, double-click, long-click, swipe, scroll, edit-text, menu 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.

Moreover, similar to web applications, an Android application may require users to provide proper input data in order to navigate to some GUI states. The generation of such input data, such as user name and password, may need human intelligence. To support this, the tool allows to provide a list of name-value pairs to guide the exploration of crawler.

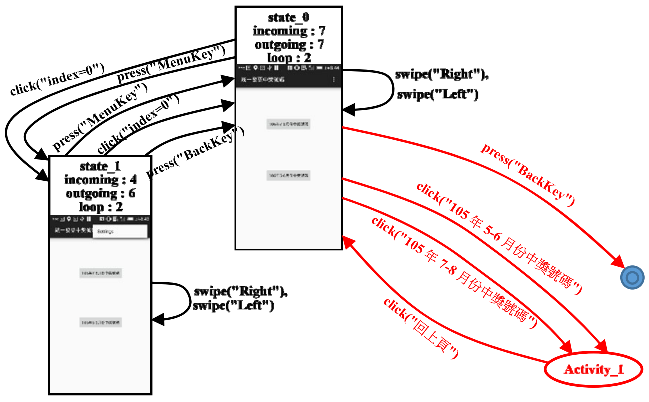
Figure 7 shows the generated hierarchical GUI state model for a small Android application, Taiwan Receipt Lottery [18]. Particularly, Figure 7(a) is the ATD of the application. It has two GUI states, activity-0 and activity-1, denoting two activities. The transitions represent the possible interactions between the activities and are labeled by <source state, event, target state>. Notice that the GUI of the activity-0 has two buttons. When users clicking on different buttons, the activity-1 will show different images of the lottery numbers. However, the crawler cannot recognize the difference of the images. Thus, these two images are considered as the same GUI state.

Figure 7(b) shows the ASD of activity-0. It illustrates that users can toggle between two GUI states of activity-0 by clicking on the setting button ("index=0") at the top-right corner of the GUI screen or by pressing the physical "menu" key. Similarly, Figure 7(c) shows the ASD of activity-1. Although the ASD has only one GUI state that is the same as its corresponding ATD state, it depicts several transitions within activity-1 which are shown in the ATD.

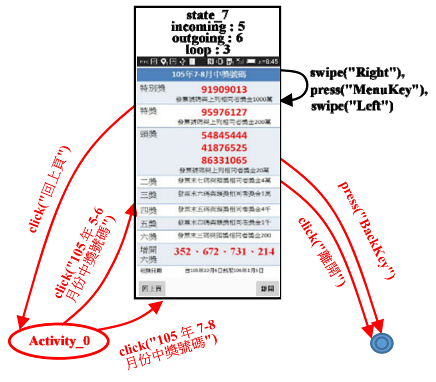
Note that, in the ASD, an event may cause a GUI change from an ASD state of one activity to an ASD state of another activity. To model such state change, a dummy state is used in the ASD to represent the target state of a transition that causes an ASD state change across different activity. As shown in Figures 7(b) and 7(c), there is a dummy state "Activity_1" in the ASD of activity-0 and a dummy state "Activity_0" in the ASD of activity-1, respectively. Each dummy state is represented by using an oval in the ASD. Further, to distinguish a regular ASD transition that does not cross different activity from a transition that leads to a dummy or terminal state, in Figures 7(b) and 7(c) the regular ASD transitions are shown in black color while the ASD transitions crossing different activity or leading to a terminal state are shown in red color.



(a) The ATD of the Taiwan Receipt Lottery application



(b) The ASD of activity-0



(c) The ASD of activity-1

Figure 7. An example of the generated hierarchical GUI state model

4.2 The Limitations of the Crawler

Currently, the crawler still has several limitations. One limitation is that the proposed approach defines a GUI state with reference to the properties of UI components on the GUI screen. Thus, the crawler takes into account only the property value changes of UI components, but not the content changes of images. As a result, a change of an image is currently not considered as a change of GUI state.

Besides, our crawler supports only UI and key events for simulating user interactions with UI components and physical keys. There are still other types of events, such as system, multi-touch, and sensor events, that are not supported in our crawler yet.

Moreover, currently the crawler compares all the properties of UI components on the GUI screen to determine whether two GUI states are equivalent. This limits its ability for handling the state explosion problem. Instead of using the stopping criteria to avoid possible state explosion, one way to deal with this problem in our crawler is to abstract similar GUI states into an approximated state so as to reduce the size of state search space.

5 Experiments

To evaluate the effectiveness of the proposed crawler and the hierarchical GUI state model, the following research questions are proposed.

RQ1. Is the GUI state model generated by the crawler more complete as compared with the model created manually?

RQ2. What is the time efficiency of the crawler as compared with human?

RQ3. Does the hierarchical state model generated by the crawler have a better readability than the traditional flat state model?

The following experiments were conducted to address each of the research questions. For the experiments, five small to medium-sized Android applications are chosen from different categories of Google Play. The computer used for running the tool has an Intel Core i5-4210U CPU (1.7GHz) and 4GB of memory. The device used for running the Android applications is Samsung Galaxy S5. The Android OS of the device is v4.4.2 and the device has a 2.5GHz CPU (Qualcomm Snapdragon 801) and 2GB of memory.

5.1 Evaluation of the Crawler's Coverage

To evaluate 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 graduate students 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 2 lists several attributes of the selected applications, such as the download number, version, user rating, and the number of events and states.

Table 2. The selected applications in the experiments

Apps	Download number	APK file size	Version	User Rating	Number of Events	Number of States
Notepad [19]	100,000 - 500,000	3.9 MB	1.4	3.9	261	15
Taiwan Receipt Lottery [18]	100,000 - 500,000	4.3 MB	4.3	4.2	28	4
QR Code Reader [20]	10,000,000 - 50,000,000	2.0 MB	1.7.4	4.4	42	8
Volume Booster Pro [21]	1,000,000 - 5,000,000	1.9 MB	2.6.4	4.3	55	9
Magnifier [22]	1,000,000 - 5,000,000	202.4 KB	2.2.9	3.9	61	6

Table 3 shows the results of the coverage of the GUI models generated by the crawler as compared with the referenced models. The results indicate that the average coverages 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 causes of such experimental results can be varied. One possible reason could be that the events currently supported by the crawler are limited to some UI and key events. The transitions introduced by unsupported events, such as multi-touch and system events, are unable to cover at the present time. For example, the Magnifier application [22] allows users to zoom in/out an image using multi-touch events which are unsupported by the crawler yet.

Table 3. The coverage results of the Android crawler

Apps	Crawling Time (hh:mm:ss)	Transition Coverage (%)	State Coverage (%)
Notepad	1:08:56	54.0%	100.0%
Taiwan Receipt Lottery	0:07:58	69.2%	80.0%
QR Code Reader	0:33:28	76.7%	91.7%
Volume Booster Pro	0:23:48	90.9%	100.0%
Magnifier	0:08:47	31.3%	66.7%
Average		64.4%	87.7%

Moreover, both Notepad [19] and QR Code Reader [20] allow user to add and delete file records. However, the deletion event cannot be enabled unless at least a file record has existed before the deletion. Since the crawler has no knowledge about the event semantic, the crawler may not always create and add a file record into storage before deleting it. If the crawler visits the GUI of file deletion before exploring the GUI of file creation, some transitions associated to file deletion events might not be covered. This could affect the transition or state coverage. One way to avoid this situation is to allow the applications to have several records initially before starting the crawler.

In addition, both Taiwan Receipt Lottery and Magnifier have a relatively low state or transition coverage as compared with others. One possible reason for this result could be that the proposed crawler is unable to identify a GUI state involved an image. In the Taiwan Receipt Lottery application, some GUIs are represented using the images of winning lottery numbers. Although human can easily identify different GUI states from those images, the proposed crawler is unable to recognize the differences of the images at the present time as mentioned in section 4.2. Hence, the crawler considers different lottery images as the same GUI state and won't explore them further. Thus, the coverage of the crawler can be affected.

Likewise, the Magnifier application allows users to see, magnify, and freeze the images captured using camera. Thus, the GUI states of Magnifier also involve images and the crawler is unable to identify the differences of such GUI states. Besides, the widgets (i.e., buttons) of Magnifier are contained in a HorizontalScrollView layout. This means that some widgets of Magnifier won't appear on the device screen unless users scroll the widget container. However, the proposed crawler currently has no knowledge about the coordinates of a Horizontal Scroll View. Thus, the crawler is unable to scroll the horizontal widget container and find more Magnifier widgets to explore. Consequently, Magnifier has relatively poor state and transition coverages than others.

Further, the proposed crawler can also be used as a stress testing tool to test if an Android application will crash while the crawler is building the GUI model of the application. To evaluate the effectiveness of the crawler, we compare the coverage of the crawler with that of Monkey, a popular random stress testing tool developed by Google for Android. The subject of the experiment is Notepad and the experiment was conducted 10 times. For the sake of fairness, the execution time of Monkey is set to be about the same as the crawling time required by the crawler. Two event options of Monkey are used in the experiment. One is the default option in which the percentage for each type of event generated by Monkey is random. The other option is that the percentage of generated events for each event type is set to be equal.

Table 4 shows the average transition and state coverage results for the crawler and Monkey. Both transition and state coverages of the crawler are about 10 percent more than those of Monkey with different event options. The results indicate that the crawler can achieve a better coverage than Monkey. Such results might be due to the fact that the crawler can use the extracted GUI information and the knowledge of the generated GUI state model for selecting events and providing input data to explore the possible GUI states of an Android application. On the other hand, Monkey simply provides random events and has no knowledge about the application. Thus, for a nontrivial application, the crawler can achieve a better coverage than Monkey.

Table 4. The results of coverage comparison for Notepad

Tool	Average Execution Time (hh:mm:ss)	Average Transition Coverage (%)	Average State Coverage (%)
Monkey (default option)	1:08:53	40.2%	90.9%
Monkey (percentage of event type is equal)	1:05:58	42.5%	90.9%
Android crawler	1:08:56	54.0%	100.0%

5.2 Evaluation of the Crawler's Efficiency

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 and both groups have no previous experience in using the Notepad application.

The average time results to create the model for the ten participants and the crawler are respectively shown in Table 5. 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 incomplete than the model generated by the crawler. The average state and transition coverages of the manually created model are 48.2% and 7.6% which are correspondingly outperformed by the coverages of 100.0% and 54.0% obtained from the model generated by the crawler.

Table 5. The results of efficiency comparison for Notepad

Generation of GUI state model	Crawling Time (hh:mm:ss)		Transition Coverage (%)	State Coverage (%)
Manual model creation	0:03:26		7.6%	48.2%
Automatic model generation with crawler	manual operation	automatic crawling	54.0%	100.0%
	0:02:51	1:10:53		

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, may fail to capture many possible GUI states and transitions.

5.3 Evaluation of the GUI Model's Readability

To evaluate if the hierarchical GUI model has a better readability than the corresponding model constructed using a flat finite state machine, the average time required to search a GUI state from both models are evaluated. The subject of the experiment is Notepad again and four GUI states of Notepad are randomly selected. Participants of the experiments are required to find and locate these four GUI states from both the hierarchical and flat state models of Notepad.

To minimize the bias, the participants of this experiment are divided into two groups again. Each group has five participants. The group 1 will first use the flat model and then use the hierarchical model to search the GUI states. The group 2, on the contrary, will first use the hierarchical model and then use the flat model to search the GUI states. Both groups have no previous experience in reading the hierarchical and flat GUI models of Notepad. The time spent to search each of the four states is recorded for each participant.

Table 6 shows the average time results for searching the states. The results indicate that the hierarchical state model has less search time than the flat state model except only for the state C. The average search time for the flat state and hierarchical models is respectively 37.1 and 19.4 seconds. This may suggest that the proposed hierarchical state model has a better readability than the traditional flat state model.

Table 6. The average search time of the state models

State	Search time for the flat state model (sec)	Search time for the hierarchical state model (sec)
A	40.0	22.8
B	20.4	15.0
C	20.8	32.8
D	67.3	7.0
Average	37.1	19.4

5.4 Discussions of the Empirical Evaluations

From the experimental results of sections 5.1 and 5.2, we can observe that the average state and transition coverages of the generated GUI model are 87.7% and 64.4%, respectively. Besides, as compared with Monkey, the crawler also can achieve better transition and state coverages. This suggests that the proposed crawler can extract a fairly large numbers of GUI states and transitions correctly. The coverage of the generated model can be largely dependent of the types of events supported by the crawler and the event types used by the Android applications. Moreover, the average state and transition coverages of the manually created model are 48.2% and 7.6%, respectively. This shows that the manually created GUI model can be error-prone, especially for nontrivial applications.

Thus, from the results of sections 5.1 and 5.2, the answer to *RQ1* is that "the GUI state model generated by the proposed crawler is more complete as compared with the model created manually."

Moreover, from the experimental results of section 5.2, we can observe that the time spent by humans to create the GUI model is far less than the time used by the crawler. This result is not surprising because human can directly derive a GUI model intuitively when interacting with the application. In contrast, the crawler needs to explore all possible events before generating the model. Besides, the crawler needs to restart the application frequently in order to backtrack to previous state. A restart of an Android application may take about 50 to 60 seconds, which can significantly affect the efficiency of the crawler.

However, the model generation can be fully or largely automated by using the crawler. As shown in Table 5, the average time required to manually operate the crawler is 2 minutes and 51 seconds (2:51) which is less than the time (3 minutes and 26 seconds) needed to create the GUI model manually. Thus, from the results of section 5.2, the answer to *RQ2* is that "the time efficiency of the crawler would be acceptable if only the manual effort is considered."

Further, from the experimental results of section 5.3, we can see that the average search time of the hierarchical state model (19.4 seconds) is much less than that of the traditional flat state machine (37.1 seconds). The rationale behind this result could be that the GUI behaviors of intra- and inter-activity are represented in different levels of the hierarchy using the ATD and ASDs. This can greatly reduce the complexity of the GUI state model and allow users to find and understand some parts of the GUI model more easily and, hence, can facilitate the model validation.

Overall, from the results of section 5.3, the answer to *RQ3* is that "the hierarchical state model generated by the crawler can have a better readability than the traditional flat state model."

6 Conclusions and Future Work

This paper has presented a crawling approach to automate the generation of GUI state model for Android applications. 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 GUI characteristics of Android is developed to support the proposed approach. Several case studies were conducted to evaluate the proposed crawler and the hierarchical GUI state model.

The experimental results show that the proposed approach can be valuable. The GUI model generated by the crawler has a promising coverage as compared with the model created manually. Although the developed crawler may require more time to generate a GUI model than human does, the model generation can be completely or largely automated and, hence, the manual efforts required to generate the GUI state model using the crawler can be ignored. Besides, the evaluation results also indicate that the proposed hierarchical state model can greatly improve the readability of model and, hence, can facilitate the model validation.

In the future, we plan to improve the efficiency of the proposed crawler. A possible efficiency improvement for the crawler is to minimize the number of times to restart the Android applications when the crawler requires to backtrack 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 to abstract GUI state information.

Acknowledgments

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, http://www.appbrain.com/ stats/number-of-android-apps.

[2] A. Memon, I. Banerjee, A. Nagarajan, GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing, *Proceedings of the 10th Working Conference on Reverse Engineering*, Victoria, Canada, 2003, pp. 260-269.

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

[4] F. Zaidi, Fuzzy Clustering and Visualization of Information for Web Search Results" *Journal of Internet Technology*, Vol. 13, No. 6, pp. 939-952, November, 2012.

[5] S.-A. Lo, C.-C. Hsu, S.-M. Hsieh, W.-M. Chen, RankCloud: A Cloud-Based Webometrics Ranking System, *Journal of Internet Technology*, Vol. 14, No. 1, pp. 45-55, January, 2013.

[6] P. Wang, B. Liang, W. You, J. Li, W. Shi, Automatic Android GUI Traversal with High Coverage, *Proceedings of the Fourth International Conference on Communication Systems and Network Technologies (CSNT)*, Bhopal, India, 2014, pp. 1161-1166.

[7] Android Debug Bridge, http://developer.android.com/tools/help/adb.html.

[8] Monkey, http://developer.android.com/tools/help/monkey.html.

[9] D. Amalfitano, A. R. Fasolino, P. 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)*, Berlin, Germany, 2011, pp. 252-261.

[10] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, A. 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, Germany, 2012, pp. 258-261.

[11] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, G. Imparato, A Toolset for GUI Testing of Android Applications, *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM)*, Trento, Italy, 2012, pp. 650-653.

[12] T. Takala, M. Katara, J. 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, Germany, 2011, pp. 377-386.

[13] W. Yang, M. R. Prasad, 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*, Rome, Italy, 2013, pp. 250-265.

[14] A. Machiry, R. Tahilizni, M. Naik, Dynodroid: An Input Generation System for Android Apps, *Proceedings of the 9th Joint Meeting on Foundations of Software Engineering*, Saint Petersburg, Russian, 2013, pp. 224-234.

[15] H. Zhu, X. Ye, X. Zhang, K. Shen, A Context-Aware Approach for Dynamic GUI Testing of Android Applications, *Proceedings of the 39th IEEE Annual Computer Software and Applications Conference (COMPSAC)*, Taichung, Taiwan, 2015, pp. 248-253.

[16] W. Choi, G. Necula, 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, Indiana, 2013, pp. 623-640.

[17] UI Automator, http://developer.android. com/tools/testing-support-library/index.html#UIAutomator.

[18] Taiwan Receipt Lottery, https://play.google.com/store/apps/details?id=com.twecs&hl=zh-TW.

[19] Notepad, https://play.google.com/store/apps/details?id=notepad.sisa.mobi&hl=zh-TW.

[20] QR Code Reader, https://play.google.com/store/apps/details?

id=tw.mobileapp.qrcode.banner&hl=zh-TW.
[21] Volume Booster Pro, https://play.google.com/store/apps/details?id=com.volume.sound.manager&hl=zh-TW.
[22] Magnify, https://play.google.com/store/apps/details?id=com.appdlab.magnify&hl=zh_TW.

Biographies

Chien-Hung Liu is an assistant professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. He received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. His research interests include software engineering, software testing, and cloud computing.

Ping-Hung Chen is a Ph.D. candidate of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. He received his M.S. degree from Graduate Institute of Computer, Communication and Control Engineering at National Taipei University of Technology in 2002. His research interests include software development and testing.