國立臺灣大學電機資訊學院電機工程學研究所

碩士論文

Department of Electrical Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master Thesis

以常識對安卓應用程式測試之技術與工具

Techniques and Tools for Android Application Testing with Common Sense

羅元鴻

Yuan-Hong Lo

指導教授：王凡 博士

Advisor: Farn Wang, Ph.D.

中華民國 104 年 7 月

July, 2015

國立臺灣大學碩士學位論文
口試委員會審定書

以常識對安卓應用程式測試之技術與工具
Techniques and Tools for Android Application Testing with Common Sense

本論文係羅元鴻君（R01921076）在國立臺灣大學電機工程學研究所完成之碩士學位論文，於民國 104 年 07 月 10 承下列考試委員審查通過及口試及格，特此證明

口試委員：

____
（指導教授）

陳明義　　　　陳銘憲

雷欽隆　　　　郭大維

系主任、所長　廖婉君

誌謝

首先誠摯的感謝指導教授王凡博士，老師悉心的教導使我得以一窺軟體測試領域的深奧，不時的討論並指點我正確的方向，使我在這些年中獲益匪淺。老師對學問的嚴謹更是我輩學習的典範。

三年裡的日子，實驗室里共同的生活點滴，學術上的討論、言不及義的閒扯、讓人又愛又怕的宵夜、趕作業的革命情感，感謝眾位學長姐、同學、學弟妹的共同砥礪，各位的陪伴讓三年的研究生活變得絢麗多彩。

感謝實驗室助理段志奇先生不厭其煩的協助我進行實驗，並不時的提供使用者角度上的建議、協助我改進整套實驗工具。

最後，謹以此文獻給我摯愛的家人，感謝他們在我背後默默的支持。

中文摘要

由於行動裝置應用市場的高度競爭，軟體測試逐漸成為一項不可或缺的流程。作為介於程式開發者與使用者間的第三方測試者，在無法拿到程式源碼、正規或半正規的規格、或是開發者所撰寫的基礎測試腳本，要對程式進行有效率的測試是非常困難的。我們希望能提出系統化的方式，將人類對程式的理解應用在程式的測試上。

為此我們創造了一個叫作「常識」的模型以人類對畫面與動作理解的概念來對程式行為進行描述。為了統一並減少混淆，我們對各種類的應用程式定義了各自的統一辭彙來描述程式的畫面與動作。又為了有效率的以人類的概念對程式行為賦予意義，我們開發了一套圖型化使用者介面工具叫作「ＳｐｅｃＥｌｉｃｉｔｏｒ」來輔助測試者建立「常識」模型。並且我們提出了兩個依據「常識」模型的演算法，分別是針對測試案例的產生與測試案例的評估。在章節最後討論了我們對「常識」模型在應用的最大目標與其可能所需要的技術。

關鍵字：常識、安卓、圖型化使用者介面測試、測試案例的產生、測試案例的評估

ABSTRACT

Software testing becoming an inevitable process due to the highly competitive market of the mobile applications. As a third party between an application developer and customers, testing an application without source codes, formal or semi-formal specification, or any testing scripts written by the developer is hard to be efficiency. We want to find a systematic methodology testing an application by the human understanding of the application.

We create a structure named “Common Sense” to model an application behavior with human concepts, and define sets of normalized terms for many kinds of application to descript concepts of screens and actions. To efficiently extract human concepts into a common sense model, we develop a GUI tool called “SpecElicitor”. We also present algorithms generating and evaluating test cases by using common sense models. In the end, we discuss the ultimate objective and potential needs of techniques to achieve our goal.

Keywords: common sense, Android application, GUI testing, test case generation, test case evaluation.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1 Introduction

1.1 Motivation

Due to the highly competitive market of the mobile applications, an application need to be as stable as it could before publishing. Otherwise, the application will rapidly fade away in the market. There are many good applications with innovative ideas disappear because of their poor stability. Therefore, software testing becoming an inevitable process to ensure that an application is steady enough to show up to the public.

In practice, testing applications whether by manually operating mobile devices or writing testing scripts are both labor-intensive. To reduce the cost for personnel expenses, there are many algorithms and methodologies to tests applications automatically. For example, Android provide a random test case generator called “Monkey”[1]. Many studies are focusing on using a state machine to model an application and testing it by covering some criteria like simple path coverage.

However, if a tester cannot acquire source codes, formal specification, semiformal specification, or testing scripts written by the developer, it is not easy to test efficiently. In our supposition, an application is designed to achieve its goals which means it has main functionalities and optional features. The major parts of an application should be tested in a higher priority to ensure a better user experience. However, it is hard to tests the major parts automatically, because it is not easy to let a computer knows which functionalities are more important than others without any human interventions. If we can offer some information about the application like “What screen is it?”, “What action is it?”, or “Is the screen correct?”, then it might help us in testing.

We want to find a way to test an application applying the understanding of an application and use the understanding to automatically generate and evaluate test cases.

Furthermore, we want to build a knowledge database storing human understandings to enhance software testing.

1.2 Purpose

We present a systematic methodology to introduce abstract human concepts into a graphical model. We develop a tool named “SpecElicitor” to help a tester efficiently eliciting his/her knowledge of an application and using those knowledge to construct the common sense model. With our common sense model, we can know what kind of applications in what situations would do and should bring out what consequences. Based on these information, we automatically produce test cases corresponding the distribution in the model to strengthen the intensity of testing and evaluate the result of traces by using our common sense model.

In our work, we focus on Android applications since Android take almost eighty percent of the market share of smartphone operating system[2]. Moreover, there are some basic testing tools can help us developing our techniques quickly.

1.3 Organization

The outline of this paper is narrated in following. Chapter 2 shows related works including automatic test cases generation, GUI testing tools and techniques, Android application testing, etc. Chapter 3 lists preliminaries which are our basic knowledge for constructing our models and developing our tools. Our tools and techniques are depicted in detail at Chapter 4, and applications are showing in Chapter 5. In Chapter 6, we present experiments to show abilities of our works. At last we summarize our contributions, problems we encountered, and future works to improve our tools and techniques.

Chapter 2 Related Work

Most of Android applications are GUI programs and they are event-driven naturally, the flows of programs are determined by user input or events which are triggered by other processes or an operating system.

Atif Memon presents new coverage criteria for GUI testing by defining a structure called “Event-Flow Graph”[3]. An event-flow graph represent a GUI program which consists of events and relations between events. Then they propose a flexible and extensible GUI testing tool named “GUITAR”[4]. Application developers and testers can use GUITAR to customize toolchains, workflows, and measurement tools conducting GUI testing. They also develop a testing tool for testing mobile applications named “MobiGUITAR”[5].

There is a tool also aim to test Android applications called “Dynodroid”[6]. They gather information of user and system events by instrumenting the framework of Android and guide the next event by monitoring the reaction of the previous event.

A testing GUI framework is showing in[7]. They design a working process by linking a tester, a test design library, and a test generation engine with standard commercial capture/replay tools. A tester can use this framework to construct a graphical model of the SUT, navigate the SUT, edit recorder test scripts, and modify test scenarios.

Sikuli[8] is a capture/replay tool based on computer vision. The tool can take a picture from a part of an application as the component and then let user write scripts to trigger the component by image pattern matching. In [9], it shows a GUI testing method by using Sikuli. A tester can design visual test cases using images.

There are some researches about user behavior modeling, most of them are focusing on web services. A study use probabilistic models for behavior modeling for Web users[10]. They introduce a maximum entropy based approach and a first order Markov mixture based approach for learning behavior models and predicting user behaviors. A study works on predicting user behavior over time[11]. They construct models of user activities based on features of current and historical behaviors.

ADAutomation shows an enhanced event-flow graph by using UML activity diagram to model user behavior on mobile applications[12]. They give meaning to the activity diagrams and use the diagrams to test applications.

Model-based testing is an approach to automatically generate and execute test cases based on a formal model that describes the application behavior [13]. By constructing the model, it can generate application’s inputs. A model-based testing toolset called TEMA, which reuses high-level models and tests interactions of applications to reveals robustness issues among different smartphone platforms [14][15].

Chapter 3 Preliminaries

3.1 Graphical User Interface Testing

GUI testing is a process to examine the graphical user interface conforming the specification of the application. GUI testing can be both functional and non-functional testing. (Functional testing is aiming to testing what the application does and non-functional testing is dealing with how well the application does.)

GUI programs are naturally event-driven. It means the program is holding on a state until some conditions has been satisfied. Functional GUI testing is trying to make sure that events are handled usage. One difficulty of GUI testing is that some functionality of the application only accomplished when a sequence of operations occur. It require lots of labors to create test cases manually.

Because of the natural of GUI programs, which is designed with a beautiful appearance and smooth operation to attract users, the layout, the colors, or the response time are also important. Non-functional GUI testing evaluates these design elements to let the application provides a great user experience.

3.2 Specification-based Testing

Specification-based testing is a minor topic of block-box testing. Black-box testing exams the application without knowledge of internal implementations, but only considers input values and output results. Because of lacking information of implementation, black-box testing only concern about whether the application is executed properly or just crashed at some circumstances.

Specification-based testing can be both functional testing and non-functional testing. Specification-based testing is seeking for the application is acting the exactly behavior and doing efficiently as the requirements described. Generally, test cases of specification-based testing are generated according specifications, requirements, and designs provided by the application developer.

3.3 Semantic Network

A semantic network is a graphical structure used for knowledge representation. In computer science, a semantic network often been used in studies of artificial intelligence. A node of a network stands for an object, a concept, or a situation. An edge stands for the relation between nodes. The relation can be “is-(a)”, “has-(a)”, “subset of”, etc. In other words, it presents a concept by using other concepts. Figure 3.1 shows an example of a semantic network.

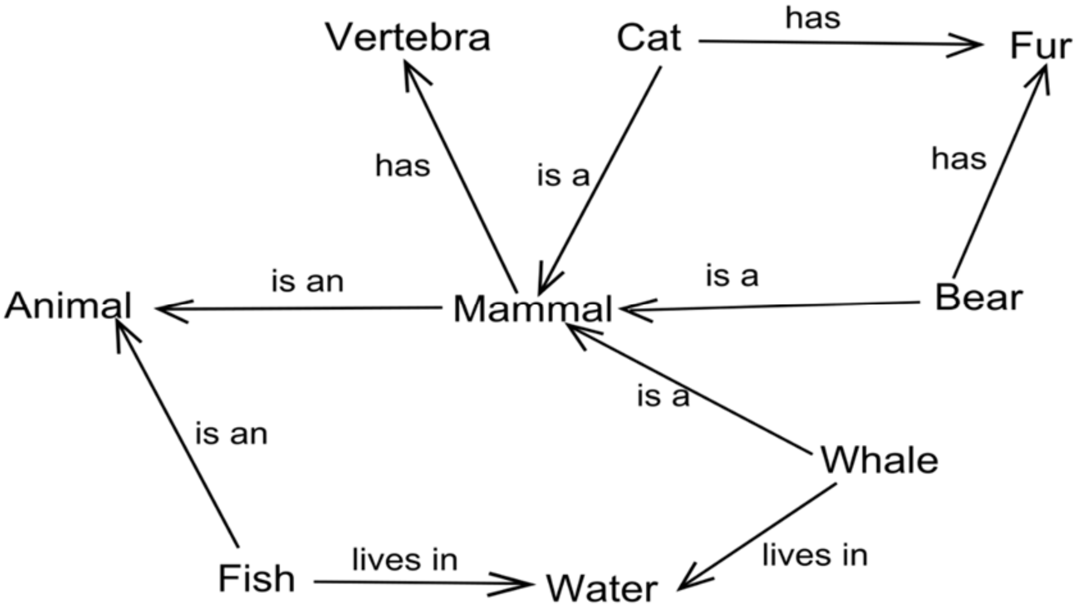


Figure 3.1 An example of a semantic network.

https://en.wikipedia.org/?title=Semantic_network[16].

3.4 Event-Flow Graph

An event-flow graph is a graphical structure to describe execution relationship between GUI events. The model is proposed by Atif Memon in 2001[3]. Nodes represent events, and edges represent the execution orders of events. An example of an event-flow model is shown in Figure 3.2.

To fully describe a GUI application, they not only use an event-flow model, but also define a structure called an integration tree. A node of integration tree represents a component of the application, and an edge represents the relation of a component contains a restricted-focus event that invokes another component. A restricted-focus event means opening a modal window. An example is shown in Figure 3.3.

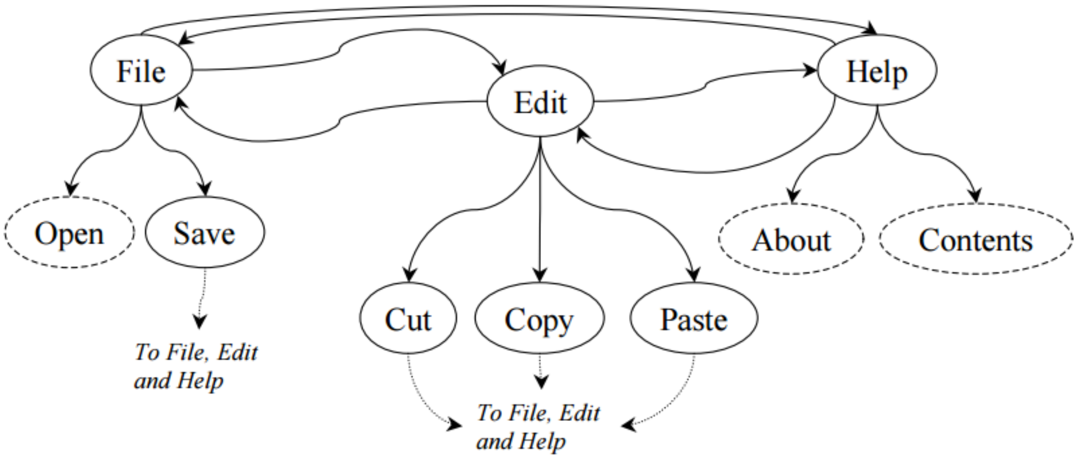


Figure 3.2 An event-flow graph for a part of MS WordPad.

Atif Memon, Mary Soffa, Martha Pollack, (2001) Coverage Criteria for GUI Testing, Page 4[3].

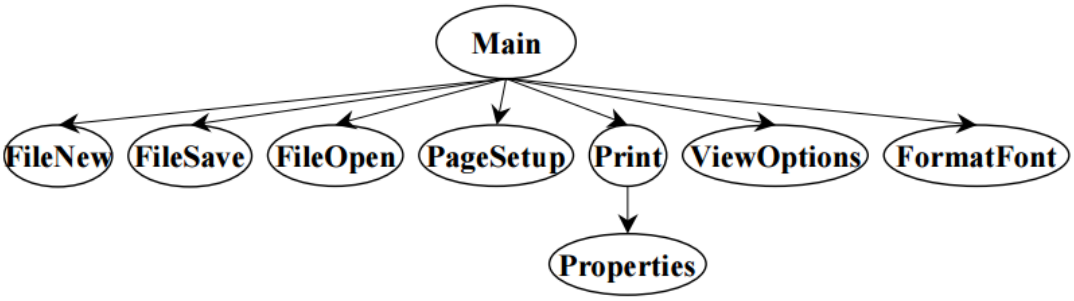


Figure 3.3 An integration tree for a part of MS WordPad.

Atif Memon, Mary Soffa, Martha Pollack, (2001) Coverage Criteria for GUI Testing, Page 4[3].

3.5 Android Automated Testing Framework

To establish an efficient environment for developing testing tools and techniques for Android applications, our laboratory create a powerful Android automated testing Framework. Developers can easily program their algorithms for test case generation, test case evaluation, or others by using the framework. The framework provides a dynamic mechanism to generate test cases and perform the test cases on the fly. The decision of the generation could refer the feedback of the previous test case. At the same time, the framework build the automata of traces which is produced by performed test cases.

The framework interacts with Android devices through Android Debug Bridge [17] and UIAutomator [18]. Those tools are developed by Android and have been widely used in testing Android applications and devices.

The framework supports various functions like "Failure Detection" and "Abstraction". "Failure Detection" can exam the application under test is crashed or not. "Abstraction" is used for filtering out redundant information of a screen. The reason why a developer needs "Abstraction" is when there exist two screens with very similar layouts and the developer wants to treat them as a same screen. If the two screens are "unabstracted", they would be distinguished by string matching. However, after remove unnecessary attributes of screens, those two screens would be seen as a same screen if the abstraction is set-up properly.

Most of our tools and techniques are developed based on this framework. We use the framework to create a GUI program for construct common sense models. Also, we use the framework to design our test case generator and experiment on our works.

Chapter 4 Common Sense Model and SpecElicitor

In this chapter, we introduce our common sense model and SpecElicitor in detail. People expect same kind of applications will perform similar behaviors and get results alike. We propose a systematic methodology to generalize those similar behaviors, which is called "Common Sense", by using normalized terms and develop a tool named SpecElicitor helping testers construct a model represent the common sense of the application under test without its specification.

A normalized term represent a concept of a screen or an action for an actionable object in the screen of the application. A common sense model is a graphical model which is integrated by normalized terms and relations between each other. SpecElicitor is a tool extracting knowledge of the application by interacting with tester and using normalized terms to endow the application human meanings.

In the beginning, a tester would learn miscellaneous information about the application (e.g. what is this screen, what is this icon means). Later, the tester has gotten the knowledge about the application, he/she has to list multiple normalized terms. Then the tester tags the application through SpecElicitor. After the tagging process, tester builds a common sense model integrated by normalized terms.

4.1 Common Sense Model

To bring human concepts to an application, we want to use a graphical structure to present the meanings of screens and user actions. There are some structures serve for representing concepts and relations between them. A semantic network is often used in artificial intelligence. Traditional semantic networks emphasize the "is-a" relation between concepts, in other words, to present a concept by other concepts. However, we do not need the “is-a” relation. We only focus on the meanings of the transition from one screen to another screen by triggering an event. An example is shown in Figure 3.1.

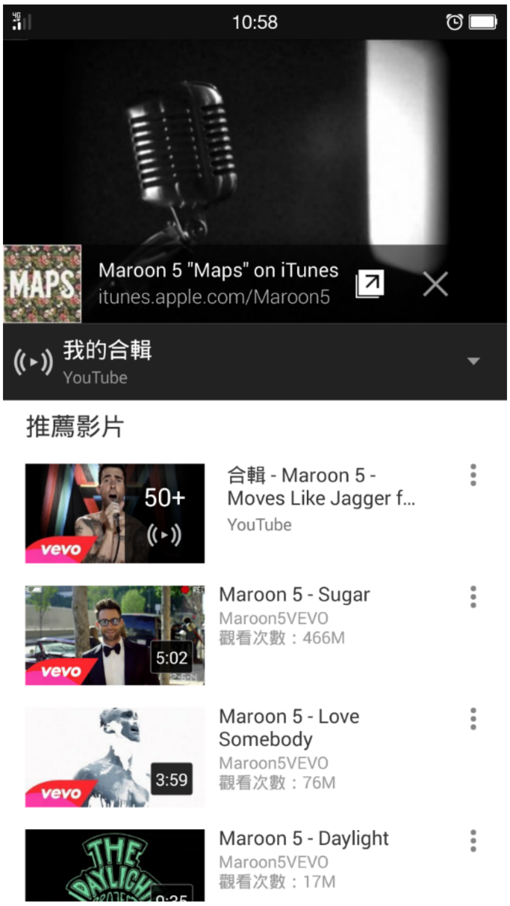
Constructing an event-flow model is an option, too. Nevertheless, to present a transition, we have to not only build an event-flow model to picture the execution flow of events, but also construct an integration tree to form components of a GUI application. Figure 3.2 and Figure 3.3 show an event-flow graph and an integration tree. There should be a table to record which event in the event-flow graph can be triggered in which component in the integration tree. The structure is a little bit complicated.

We define a graphical model to present meanings of GUI components and user events of GUI applications. On a common sense model, each node stands for a concept of screens and each edge stands for a concept of actions. The word “action” means a user event. A concept for screens or actions can use for labeling with more than one screens and actions. Likewise, a screen or an action can be tagged with multiple concepts.

An example is shown on Figure 4.1. This is YouTube on Android system. We can see there is a video playing on the top of the screen and a playlist is below the video. In this case, the tester may want to define a class named “Media Player” and labels this screen with “Display Screen” and “Playlist”.

The information of an action includes action types (e.g. click, long-click, text, etc.), action arguments (e.g. a coordinate for clicking and long-clicking, a string for texting, etc.), and the actionable object which takes the action. Human comments are recording whether this transition is pass or fail according to the opinion of the tester and reasons why the transition is fail.

For every application under test, a tester tags it with concepts and builds a common sense model. For multiple applications with same class, we can easily combine their models into one big common sense model for the class. Because nodes and edges in common sense model is represented by concepts, combining models are simply adding nodes and edges in each model to the new model.



10:58

Maroon 5 "Maps" on iTunes
itunes.apple.com/Maroon5

我的合輯
YouTube

推薦影片

合輯 - Maroon 5 - Moves Like Jagger f...
YouTube

Maroon 5 - Sugar
Maroon5VEVO
觀看次數：466M

Maroon 5 - Love Somebody
Maroon5VEVO
觀看次數：76M

Maroon 5 - Daylight
Maroon5VEVO
觀看次數：17M

Figure 4.1 YouTube on Android

Furthermore, we want to not only present the meanings in a common sense model, but also show which screen or action is important than others. We use a variable called the reference count of a screen or an action. The reference count of a node means for every screen which been tagged with this concept has appeared. The reference count of edges means there is a transition from one node to another node by doing the same action on the same actionable object. If a node has a higher reference count, the concept of the node would have been referred more times than other concepts.

There are two factors affecting a reference count. To cover an application, we have to explore the application as possible as we can. If a screen has many functional buttons, the screen would appears many time.

It is possible that there will be more than one edge constructed for a single action. For instance, Figure 4.1 shows that the application is playing a video and listing other videos. This screen will be tagged with multiple concepts. If tester chooses the “Back” button, this action will affect whole screen, both “Display Screen” and “Playlist”. Then there will be two edges been constructed. One is from node represent “Display Screen” and the other is from node represent “Playlist”. If tester chooses the stop button, there will only construct one edge which is from node “Display Screen”. An action can be tagged with multiple concepts, too. Because each edge only represent one concept, there will be multiple edges been created.

Figure 4.2 shows an example of a common sense model. The equal sign shows the reference count of a node or an edge. The darker a node is, the more referenced times it has. The topmost node is represent the initial node of the graphical structure and labeled with “<dummy>”. Because there might have multiple initial screens for one for more applications, a dummy node helps us traverse the model slightly easily. The example is combined with nine applications. Four of them are classified to “Notepad”, four of others are classified to “File Manager”, and the last one is classified to both classes.

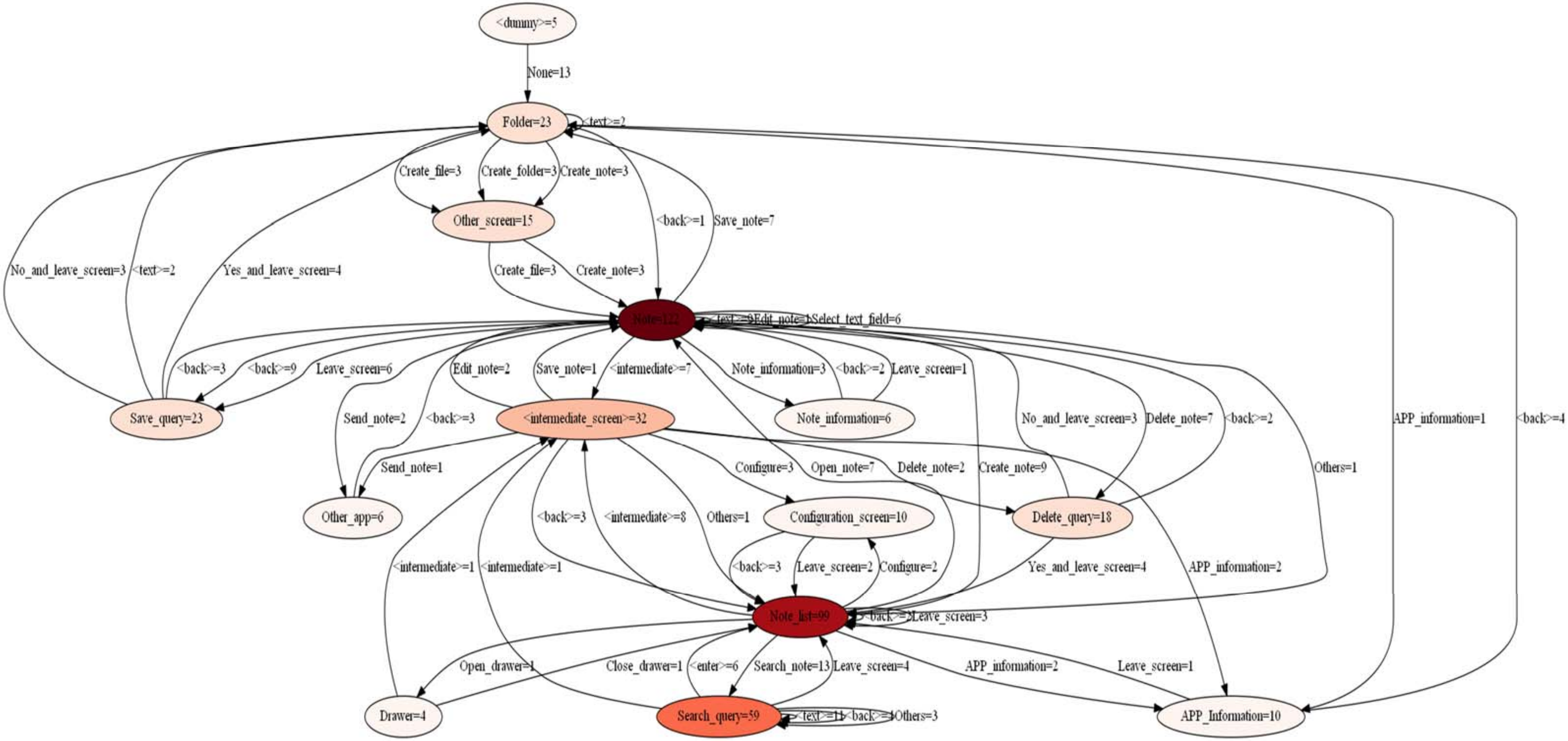


Figure 4.2 An example of a common sense model for nine applications

4.2 Normalized Term

We use a word or a term to express a concept for a screen or an action concretely. The simplest way is using the string shown on the screen or the button. However, a screen may not have a title or buttons in different applications have same functionalities but using different words. For instance, buttons with “Compress”, “Create Zip”, and “Archive” are used for create a compressed file. To reduce superfluous words or terms and avoid ambiguous concepts, we must define a normalized term for each concept.

We define sets of classes and classify applications according its design purposes, functionalities, or features into the classes. For each class we define normalized terms for screens and actions in the screen.

For instance, we define a class called “Notepad”. In this class, there are normalized terms for screens, e.g. “Note List” for the screen listing notes, “Note” for the screen displaying the note, and “Delete Query” for the screen which is popping up a dialog to verify user’s operation. And for actions, we defined “Create Note” for a button use to create a note and let user editing the note, “Search” for searching notes, “Configure” for entering the configuration page, etc.

There will be some normalized terms are generally for all kinds of application. For example, “Configure” for actions and “Configuration” for screens would appear in many application. Actually, Android even provide an API named PreferenceActivity that developers can quickly setup a configuration page by using it. In this case, we defined a general class of normalized terms. Terms in this class are describing concepts which is for all Android applications.

Therefore, when the tester need to classify an application, he/she not only chooses a class by application’s functionality but also chooses the general class. Furthermore, if an application is designed for many purpose which has lots of functionalities, the tester can choose multiple classes in this situation.

There are some special terms for particular reasons. "<intermedia_screen>" is for the screen which is for dedicated some menu screens, showing in Figure 4.3. Android provide a delicate user interface layout API called Action Bar. When the screen of the device is wider, buttons on the Action Bar will be displayed directly on the bar, showing in left side of Figure 4.3. However, buttons might be putted in the menu which is on leftmost of the Action Bar in a narrow screen, showing in right side of Figure 4.3. It depends on number of buttons on the Action Bar, width of the resolution of the device, and the developer's design. For portability, we use "<intermedia_screen>" to describe the menu in the Action Bar. Buttons in the menu will be treated as in the screen which contains the Action Bar.

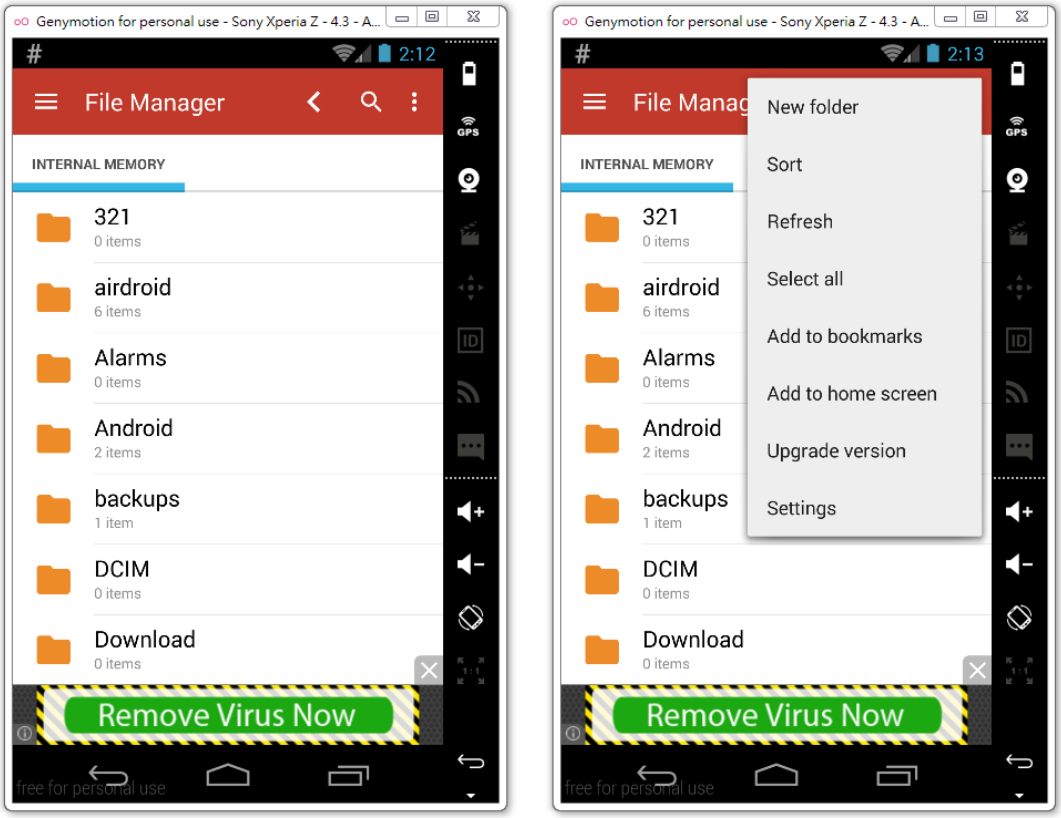


Figure 4.3 An application with Action Bar

“Other” and “Other Screen” are using for describing some actions and screens that cannot be described by other terms. If an application has some nontrivial feature, “Other” and “Other Screen” may come in handy.

To present every concept in each type of class, we need to choose a sufficient number of normalized terms. However, it will be inefficient and against our goal, finding same pattern that across multiple applications, if we define too many normalized terms. Deciding the accuracy of terms is a problem that can be trade-off.

4.3 SpecElicitor

We develop a tool called SpecElicitor to help testers to rapidly tagging test traces and use tagged traces building a common sense model. The basic layout is shown in **Figure 4.4**. SpecElicitor is a “middleman” between an Android device and a tester.

At first, a tester chooses application to test and starts SpecElicitor through our testing framework developed by my colleague. SpecElicitor get a XML file for the layout and an image file for the screenshot of the application then display on the GUI. Tester chooses an action (e.g. click, long-click, swipe, etc.) for an actionable object and tags it with a normalized term. SpecElicitor return the action to Our Android Testing framework and wait for Our Android Testing framework performing the returned action on the Android device. After the device finishing the action, Our Android Testing framework take a snapshot of the screen, and pass a XML file and an image file to SpecElicitor. When the tester want to stop tagging traces and end SpecElicitor, he/she presses the “Stop labeling” button on SpecElicitor then SpecElicitor will tell our Android Testing framework to terminate and release the Android device. SpecElicitor will loop many turns until all screens has been tagged and the tester thinks it is enough.

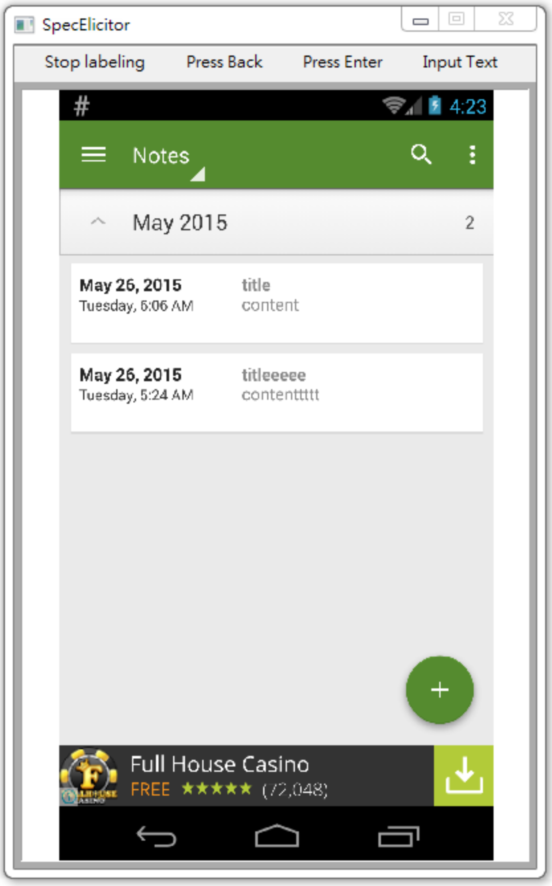
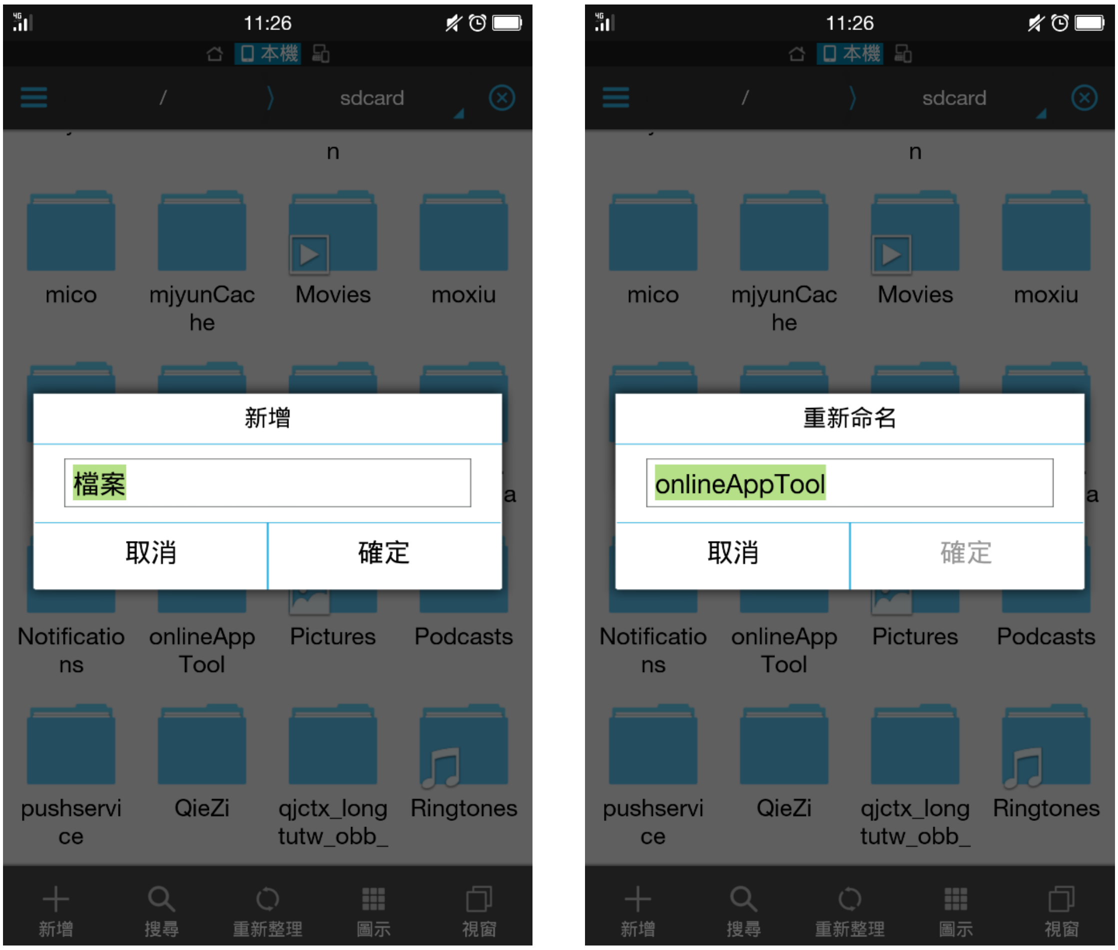


Figure 4.4 SpecElicitor

4.3.1 Screen and Action Abstraction

When applying a common sense model to generate or evaluate test cases, we expect when we encountering a screen and doing an action, we would know the meaning of the screen and the action. That means, we have to map the common sense model back to the screens and actions. There will be some screens have similar layout and functionalities that can be treated as same screen. So does actions. Even though we tagged a screen before, as long as the layout XML file of a new screen is not exactly same as the tagged one, the new screen will be treated as a different screen without tags. We using screen abstraction which is a function supported by our Android Testing framework to solve this problem.

Screen abstraction has to be as precise as possible. If the abstraction is too abstract, there will be some screens with different functions and design purposes been treated as a same screen. An example is shown in Figure 4.5. The screen on the left is creating a file and query for a file name. The screen on the right is renaming a file and query for a new file name. People can clearly know that they are two different screens cause their functionalities. Nevertheless, if the abstraction is configured too fuzzy, these two screen will be treated as a same screen. The tester is unable to choose from normalized terms from “Rename Query” and “Create Query” for these screens, because the screen on the left side is “Create Query”, not “Rename Query”, vise versa.



新增

檔案

取消 確定

重新命名

onlineAppTool

取消 確定

Figure 4.5 Two different screens with a similar layout.

Screen abstraction is hard to be generally configured. Not only since different applications would have to use different configuration, but also there are some cases is hard to distinguish whether screens are equal or not. For instance, Figure 4.5 shows two different screen as mentioned before. However, if we do not take functionality for granted, these screen can be treated as same screen.

Actionable objects would have same problems. Because objects in layout XML file does not have a unique identifier for whole application, we can only exam attributes of actionable objects and action type to guess the meaning of the action. For now we use “resource-id”, “class”, “package”, “content-desc”, and “text” when class is “android.widget.EditText” to identify actionable objects because they are more reliable than the others. All attributes and their information are listed in Table 4.1.

Table 4.1 Attributes of actionable objects

Attribute	*Type*	*Description*
index	Integer	Index for XML entry
text	String	Text showing on the object
resource-id	Integer	Java class type of the object
package	String	Java package name of the object
content-desc	String	Description of the object
checkable	Boolean	Flag showing whether the object is checkable
checked	Boolean	Flag showing whether the object is checked
clickable	Boolean	Flag showing whether the object is clickable
enabled	Boolean	Flag showing whether the object is enabled
focusable	Boolean	Flag showing whether the object is focusable
focused	Boolean	Flag showing whether the object is focused
scrollable	Boolean	Flag showing whether the object is scrollable
long-clickable	Boolean	Flag showing whether the object is long-clickable
password	Boolean	Flag showing whether the object is a password
selected	Boolean	Flag showing whether the object is selected
bounds	String	Boundary of the object

4.3.2 Iteration

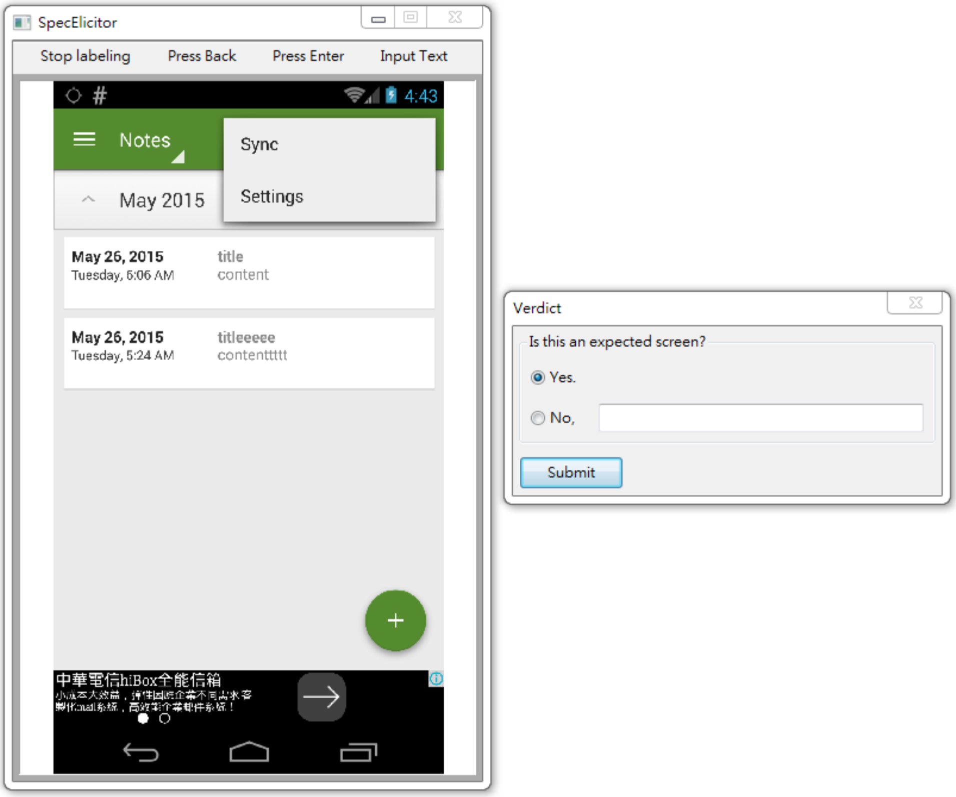
As mentioned previously, a tester using SpecElicitor to generate traces and tag traces on the fly. An iteration start with giving verdict of the last action, then determine and tag current screen with one or more normalized terms. After tagging the screen, the tester chooses an actionable object and chooses an action on the object to tag. All these operations are performed through SpecElicitor.

At first, Our Android Testing framework will take a snapshot of the screen from Android device. The snapshot contain many information, including the layout XML file, the screenshot image file, memory dump, etc. For now, SpecElicitor only require the layout XML file and the screenshot image file for letting the tester knowing the screen of the Android device interacting with it.

In the beginning of an iteration, SpecElicitor pop-up a dialog asking tester to give verdict value which is pass or fail. If the verdict is fail, tester can input the reason why it fail to the text field. Figure 4.6 shows the GUI.

After the tester giving the verdict, SpecElicitor pop-up a dialog for asking what normalized terms should be used to tag this screen. As showing in Figure 4.7, the tester can select multiple normalized terms to describe the screen.

Then SpecElicitor wait for the tester to choose an object in the screen. The tester choose an action on an object, is there is more than one action and one object. SpecElicitor will pop-up a dialog for asking normalized terms to describe this action. The GUI is shown in Figure 4.8.



Verdict

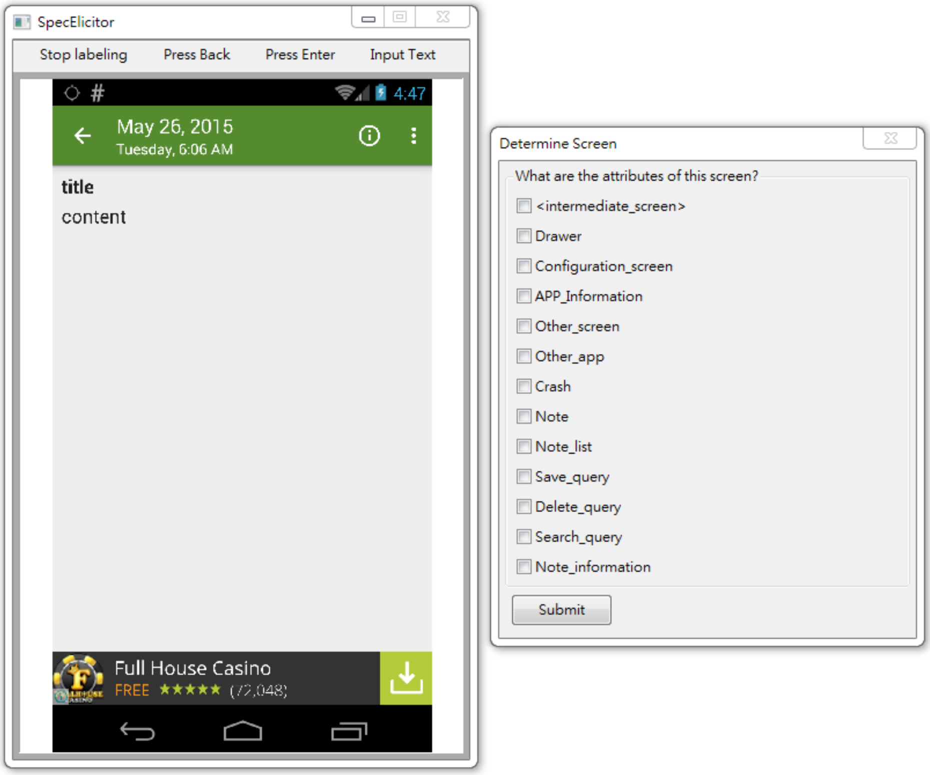
Is this an expected screen?

● Yes.

○ No,

Submit

Figure 4.6 SpecElicitor ask for verdict



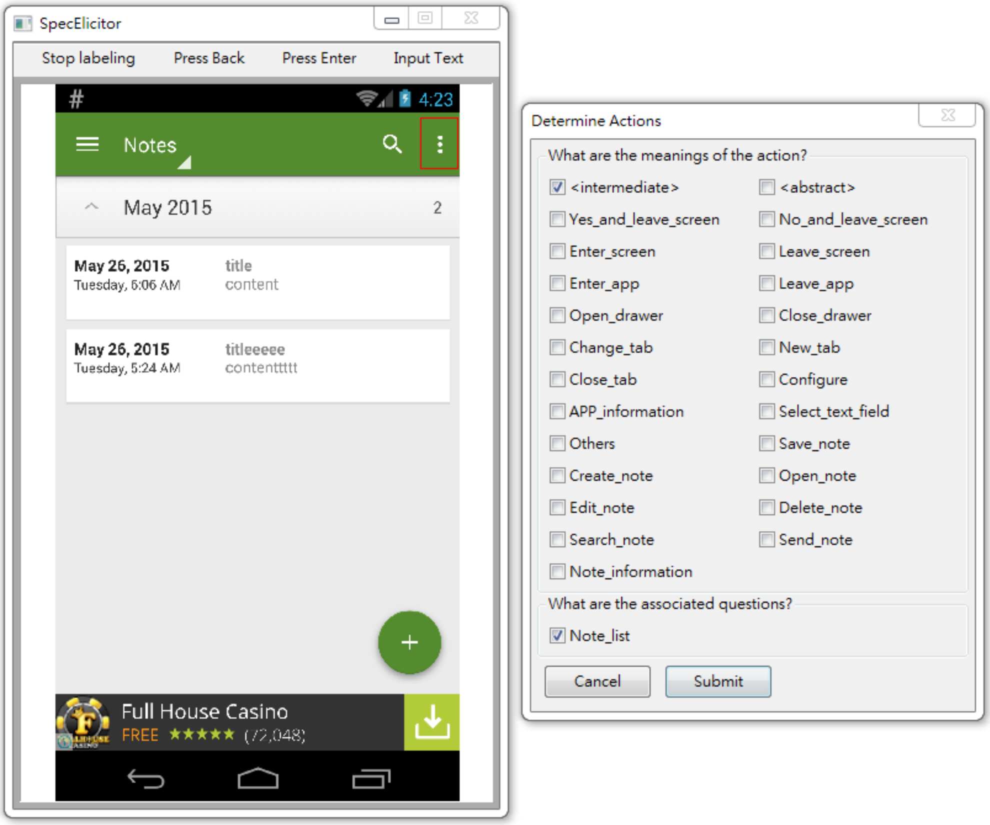
Determine Screen

What are the attributes of this screen?

□ <intermediate_screen>
□ Drawer
□ Configuration_screen
□ APP_Information
□ Other_screen
□ Other_app
□ Crash
□ Note
□ Note_list
□ Save_query
□ Delete_query
□ Search_query
□ Note_information

Submit

Figure 4.7 SpecElicitor ask for normalized terms of the screen



Determine Actions

What are the meanings of the action?

☑ <intermediate> ☐ <abstract>
☐ Yes_and_leave_screen ☐ No_and_leave_screen
☐ Enter_screen ☐ Leave_screen
☐ Enter_app ☐ Leave_app
☐ Open_drawer ☐ Close_drawer
☐ Change_tab ☐ New_tab
☐ Close_tab ☐ Configure
☐ APP_information ☐ Select_text_field
☐ Others ☐ Save_note
☐ Create_note ☐ Open_note
☐ Edit_note ☐ Delete_note
☐ Search_note ☐ Send_note
☐ Note_information

What are the associated questions?

☑ Note_list

Figure 4.8 SpecElicitor ask for normalized terms of the action

Chapter 5 Applications

5.1 Test Case Generation

As we mentioned in the motivation, we want to figure out which parts of the application are important than others.

The weight of a screen or an action is the frequency that it appears in the process of using SpecElicitor. When a tester uses SpecElicitor to give meanings for an application, if a screen has many actionable objects in it, a tester should visit the screen many times. And if an action would lead to a screen with many actionable objects, the action would be also performed many times. Thus the weight in a single common sense model only represent the frequency of appearances.

However, if lots of applications has a same action in a same screen, people might consider the action is an important feature. So, we assume that by overlapping common sense models, the weight of an action means the quantized value of the importance for the action.

To testing an application with lacks of time and resources, the main functionalities or important features should be tested more. We design an algorithm to increase the strength of testing for relatively important parts of an application.

5.1.1 Algorithm

The method is randomly selecting next step like Monkey[1], but selecting from weighted actions. The inputs of test case generation is the demanded trace length, the common sense models which have same type with application under test, and the common sense model of the AUT.

We overlapping common sense models including the model of AUT and initial program control variables in the beginning. To generate each step, we capture a snapshot of the application at first. Then we identify the screen whether it has been graven a meaning. If we cannot find the meaning of the screen, we randomly choose an action like Monkey. Otherwise, we find actions in the screen and identify whether the screen has a meaning and its weight of the meaning in the overlapped common sense model. We collect these actions and their weight into a list, and randomly select an action from the weighted list. The pseudocode is shown in below.

Algorithm 1: Test Case Generation

```
Input: model, modelList, traceLength
overlappedModel = Overlap(modelList);
traceStep = 0;
while traceStep less or equal than traceLength do:
    screen = CaptureScreen();
    state = AbstractScreen(screen);
    stateLabel = model.FindLabel(state);
    actionList = GetActions(state);
    if stateLabel is not null then:
        weightedActionList = [];
        foreach action in actionList do:
            actionLabel = model.FindLabel(action);
            actionWeight = model.GetWeight(action);
            weightedActionList.append((action, actionWeight));
        end
        action = WeightedRandomChoose(weightedActionList);
    end
    else:
        action = RandomChoose(actionList);
    end
    PerformAction(action);
    traceStep += 1;
end
```

5.2 Test Case Evaluation

We use a common sense model to form the behavior of an application. So we can check whether the trace complies with the model to verify whether every transitions meet the behavior of the application. If a transition does not meet the model, it might be caused by a bug or a defect of the model. Also we record a verdict in a common sense model for every transitions when a tester using SpecElicitor. A trace contains a transition complying with the model but the verdict of the transition is “Fail” would be evaluated a fail trace.

If a transition contain an unidentifiable screen or an unidentifiable action, it will be reported to the tester.

5.2.1 Algorithm

The input of test case evaluation is the model of the application and a trace generated by any test case generation algorithm. We initialize the label of the previous action and screen to null and the trace step to zero.

To verify each transition, we find the screen of the current step of the trace, and identify the screen to get the meanings of the screen. If the labels of the screen cannot be figure out, it will be reported to the tester. It will be reported when the labels of the current screen is “Crash”. Then we identify the previous screen and action, too. If the labels of the previous screen and action is null, it will be reported.

After we get the labels of the previous screen, the previous action, and the current screen, we check whether the transition exists in the model. If the transition does not exist or the verdict of the transition is “Fail”, it will be shown on the report. Followings are the pseudocode of our test case evaluation.

Algorithm 2: Test Case Evaluation

Input: model, trace

```
traceStep = 0;
preActionLabel = null;
preScreenLabel = null;
while traceStep less or equal Length(trace) do:
    screen = GetScreen(trace, traceStep);
    state = AbstractScreen(screen);
    curScreenLabel = model.FindLabel(state);
    if curScreenLabel is null then:
        ReportError(trace, traceStep, “Unidentifiable Screen”);
        traceStep += 1;
    end
    if curScreenLabel is “Crash” then:
        ReportError(trace, traceStep, “Crash”);
        traceStep += 1;
    end
    if preActionLabel is null and traceStep not equal zero then:
        ReportError(trace, traceStep, “Unidentifiable Action”);
        traceStep += 1;
    end
    else:
        if preScreenLabel is not null then:
            verdict = model.GetVerdict(preScreenLabel,
                                       preActionLabel,
                                       curScreenLabel);
            if verdict is null then:
                ReportError(trace, traceStep, “No Verdict”);
                traceStep += 1;
            end
            if verdict is “Fail” then:
                ReportError(trace, traceStep, “Fail”);
                traceStep += 1;
            end
        end
    end
    action = GetAction(trace, traceStep);
    preActionLabel = model.FindLabel(action);
    preScreenLabel = curScreenLabel;
    traceStep += 1;
end
```

Chapter 6 Experiments

To show the ability of our common sense model and the algorithm for test case generation, we choose three applications and define three classes of normalized terms.

We investigate three issues:

1. The results of evaluating traces generated by our test case generator.
2. The proportions of actions in traces produced by our algorithm are in accordance with the distribution of the weights of the actions.
3. The appearance times of unlabeled screens.

We generate test cases by using our algorithm mentioned in Section 5.1, and evaluate these traces as we described in Section 5.2. For each application, we run a hundred traces for each application and twenty steps for each trace.

6.1 Application under Test

We download eight applications from Google Play. Four of them are classified to "Notepad", and others are classified to "File Manager". We choose two applications in each class type, and the others are used for constructing our common sense database.

We also choose an application named "Omnidroid"[19], which is an open source project. Omnidroid is an automated event manager make users to automate system functions.

All of applications are listed in Table 6.1. The names of applications are similar, so we use package name to identify each other. The italics in Table 4.1 are the applications using in the experiments.

Table 6.1 Applications under Test

#	Application	Class
1	MiniNote	Notepad
2	*Natural Notes*	*Notepad*
3	Ultimate Notepad	Notepad
4	My Notes – Notepad	Notepad
5	ES 檔案瀏覽器	File Manager
6	*Fo File Manager*	*File Manager*
7	File Manager	File Manager
8	Root Explorer	File Manager
9	*Omnidroid*	**Omnidroid*

6.2 Normalized Term

We define four sets of normalized terms for general applications, “Notepad”, “File Manager”, and a special set for Omnidroid. For applications classified with “Notepad”, we use normalized terms of general applications and “Notepad” to present their concepts. For applications classified with “File Manager”, we apply normalized terms of general applications and “File Manager”. For the application classified with both “Notepad” and “File Manager”, we employ all four sets of normalized terms. The four sets of normalized terms are listed in Table 6.2, Table 6.3, , and Table 6.5.

Table 6.2 Normalized terms for all Android applications

#	*For screens*	*For actions*
1	<intermediate_screen>	<intermediate>
2	Drawer	Yes_and_leave_screen
3	Configuration_screen	No_and_leave_screen
4	APP_information_screen	Enter_screen
5	Other_screen	Leave_screen
6	Other_app	Enter_app
7	Crash	Leave_app
8		Open_drawer
9		Close_drawer
10		Change_tab
11		New_tab
12		Close_tab
13		Configure
14		APP_information
15		Select_text_field
16		Others

Table 6.3 Normalized terms for “File Manager”

#	For screens	For actions
1	Folder	Change_folder
2	Create_query	Open_file
3	Search_query	Create_file
4	Compress_query	Create_folder
5	Delete_query	Search_file_or_folder
6	Rename_query	Select_all
7	Copy_query	Select_file_or_folder
8	Cut_query	Deselect
9	Paste_query	Compress_file_or_folder
10	Move_query	Delete_file_or_folder
11	Bookmark_screen	Rename_file_or_folder
12	Item_information	Copy_file_or_folder
13		Cut_file_or_folder
14		Paste_file_or_folder
15		Move_file_or_folder
16		Refresh
17		Bookmark
18		Item_information

Table 6.4 Normalized terms for “Notepad”

#	For screens	For actions
1	Note	Save_note
2	Note_list	Create_note
3	Save_query	Open_note
4	Delete_query	Edit_note
5	Search_query	Delete_note
6	Note_information	Search_note
7		Send_note
8		Note_information

Table 6.5 Normalized Terms for Omnidroid

#	For screens	For actions
1	Main	Create_rule
2	Event_list	Select_event
3	Event_screen	Add_filter
4	Option_list	Add_action
5	Filter_setting	Select_option
6	Rule_list	View_rules
7	Log_screen	View_log

6.3 Implementation

Our tools and techniques are developed based on our Android testing framework which is introduced in Section 3.5. We program our tools and algorithms with Python

3.4.3 and use wxPython Phoenix for constructing graphical user interfaces.

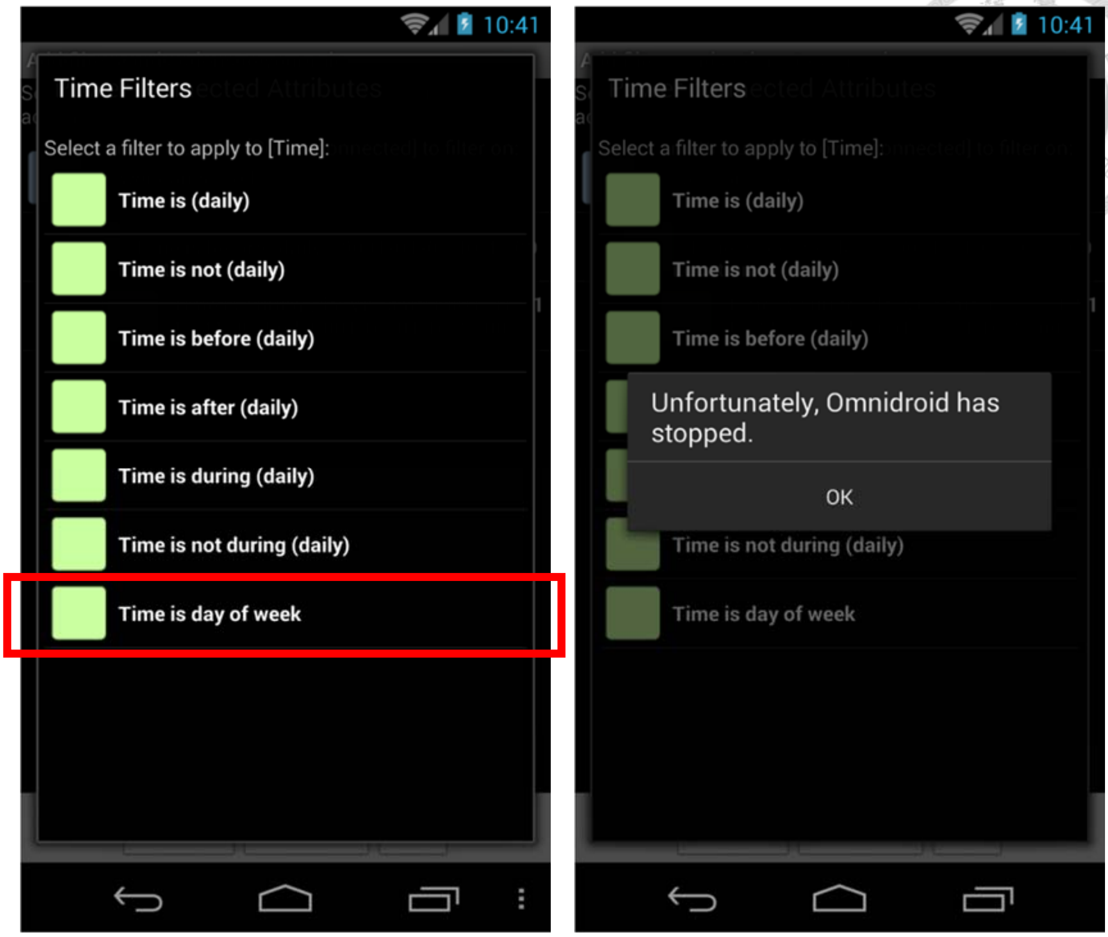
Our programs can be executed on every operating systems supported Python and wxPython. We executed SpecElicitor and created common sense models on Windows 7 and Ubuntu 14.04. Then we generate test cases on Windows 7. We use a virtual device[20] and HTC Desire 820[21] through our experiments.

6.4 Result

We did not find any bugs neither in Natural Notes nor Fo File Manager, but we found twelve crashes in Omnidroid. Results are listed in Table 6.6. The reason why Omnidroid has twelve crashes is triggered by clicking a button showing in Figure 6.1.

Table 6.6 Result of Test Case Evaluation

AUT	*Total Traces*	*Crash*	*Failed Transition*	*Invalid Transition*	*Total Bugs*
Natural Notes	*100*	*0*	*0*	*0*	*0*
Fo File Manager	*100*	*0*	*0*	*0*	*0*
Omnidroid	*100*	*12*	*0*	*0*	*12*



Unfortunately, Omnidroid has stopped.

Figure 6.1 A bug in Omnidroid

We show the porportions of actions in Figure 6.2 and 錯誤! 找不到參照來源。. The ordinate lists actions concating with the screen which the action is triggered at, and the abscissa shows the percentage of an action. For example, at the screen with label “Search_query” in Natural Notes, we can do “leave_screen”, “text”, “back”, or “enter”. The test case generator randomly choose an action from these four actions by their weight. So the summation of the percentage for the four actions is one hundred percent. We can see that most of proportions of actions will be similar with the proportions in the model. However, in Fo File Manager, we can see that there is an action “Change_folder” in screen “Folder” has been triggered more times then other actions. The reason is there are many differet screens are labeled with “Folder”, but some screens may not contain actions like

“Create_Folder” or “Back”. Therefore the test case generator can only perform other actions. Figure 6.4 shows an example of two screens both are labeled with “Folder”. The left screen is does not have a button to create folder.

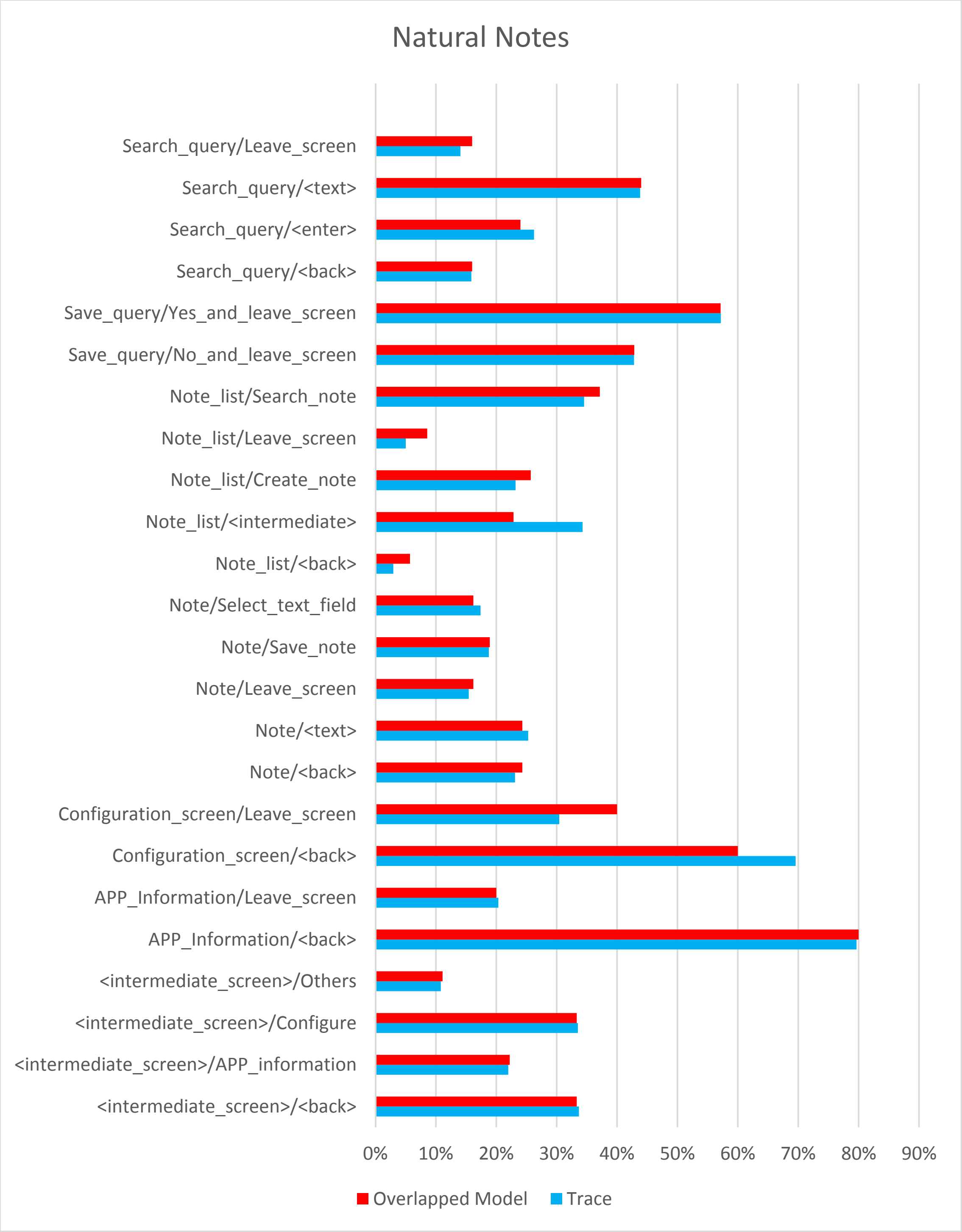


Figure 6.2 Action proportions of Natural Notes

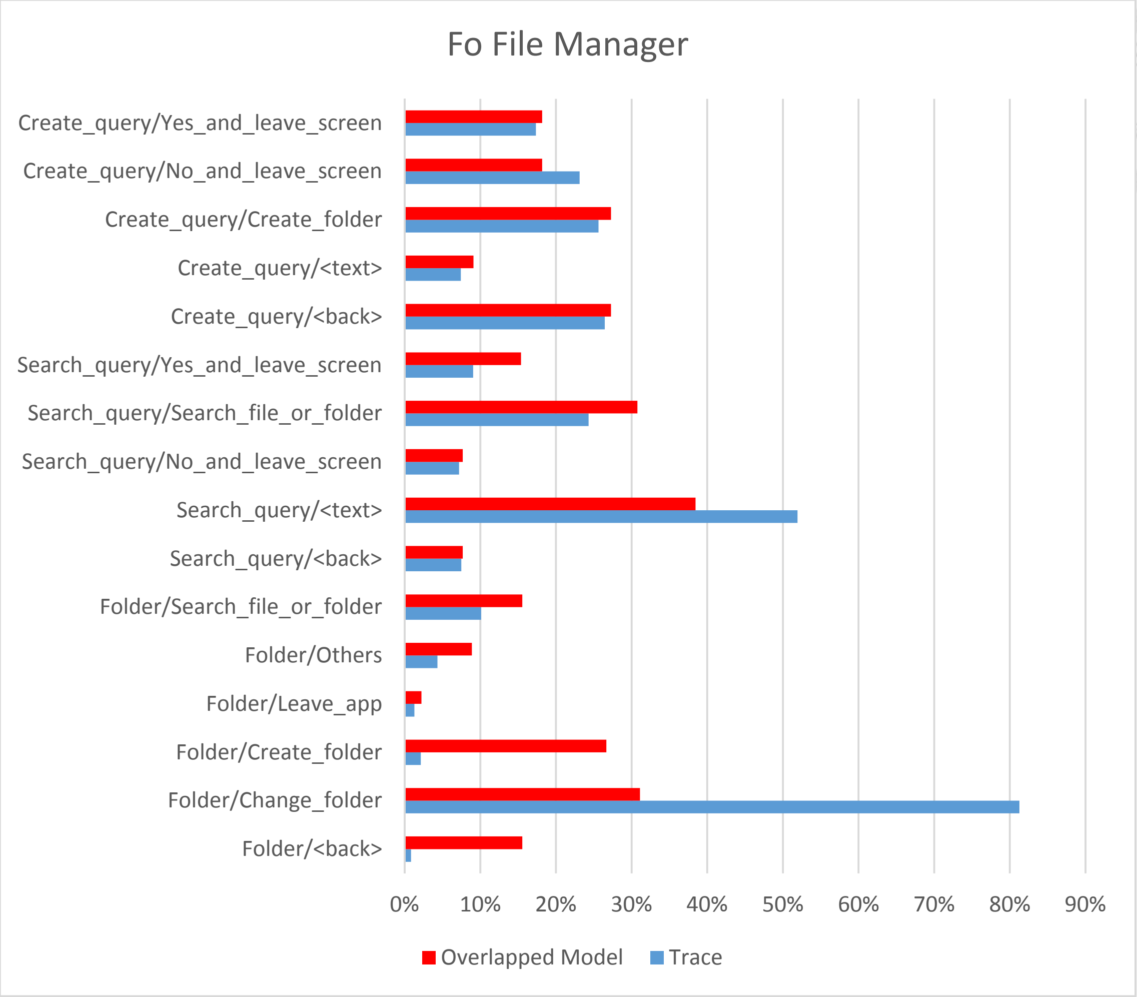


Figure 6.3 Action proportions of Fo File Manager

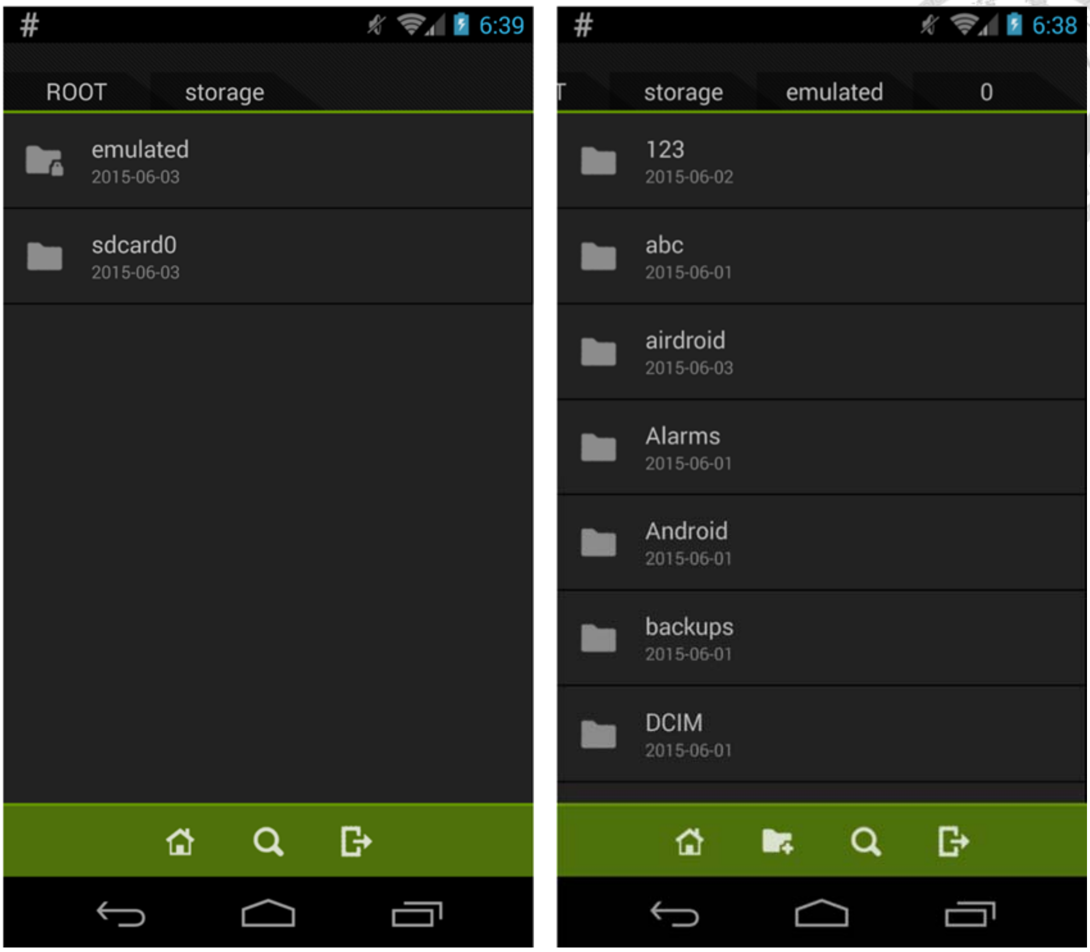


Figure 6.4 A screen comparison of Fo File Manager

At last, as we mentioned in introducing SpecElicitor, test case generation, and test case evaluation, identifying a screen or an action is a basic but important function in our works. Currently, we use abstraction to filter out redundant information of a screen and actionable objects, then compare the rests to identify a screen or an action. Nevertheless, if an application frequently changes its layout, abstraction might not be able to abstract every similar screens into one abstracted screen. For instance, Fo File Manager has various layouts, so there are almost fifty percent of steps are chosen randomly, since a screen cannot be identified. Layouts of Natural Notes are more simple than Fo File Manager, there are only five percent of actions are chosen randomly. Figure 6.5and Figure 6.6 show the results of Natural Notes and Fo File Manager. The ordinate is a number of cumulated steps or actions choosed randomly. The abscissa is a number of traces.

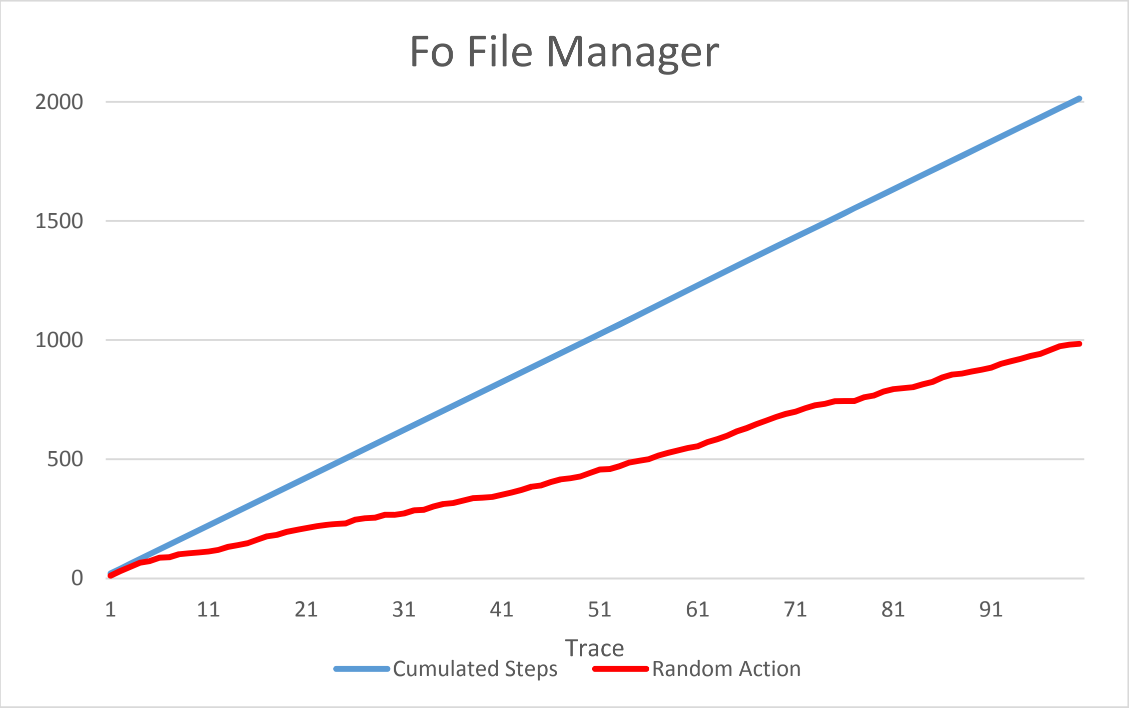


Figure 6.5 Unidentifiable screens of Fo File Manager

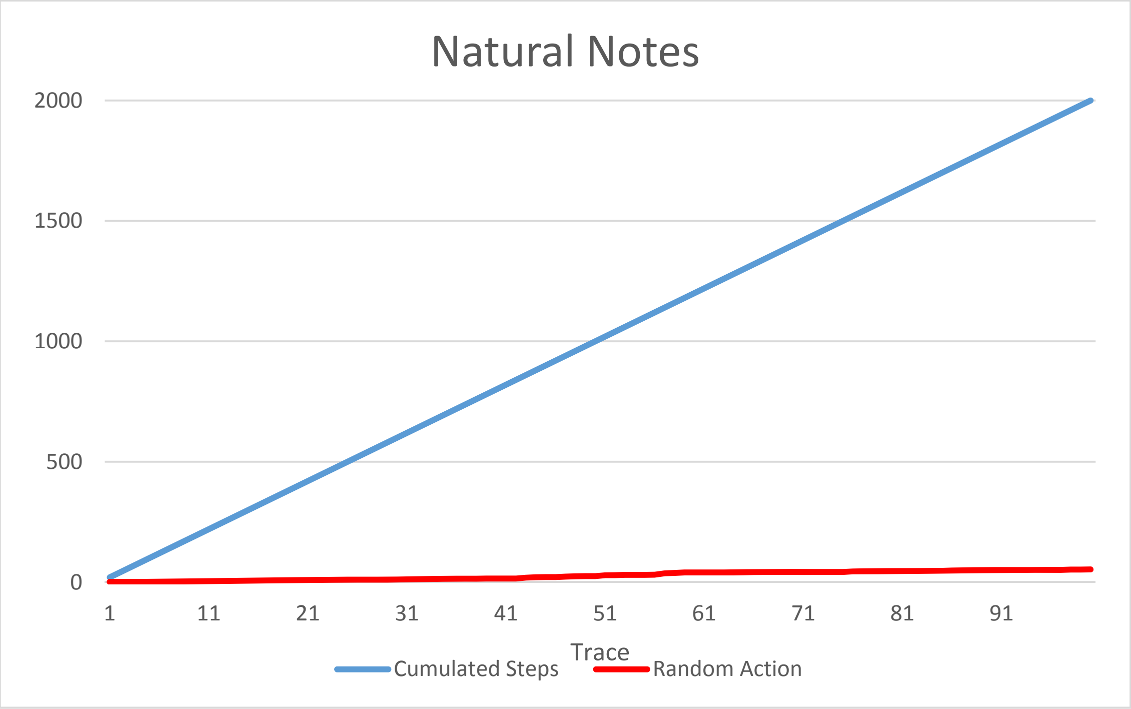


Figure 6.6 Unidentifiable screens of Natural Notes

Chapter 7 Conclusion

7.1 Summary

We define a new graphical structure to model an application. By using sets of normalized terms, we give descriptions for each screen and action in an application then use the transition from a screen to another screen to form the program behavior. Based on human understanding of the normalized terms and the application, we call the new graphical structure “Common Sense” model.

To elicit the human common sense of an application, we develop a GUI tool. The tool is a middle man between a tester and an application. By interacting with the tester, SpecElicitor construct the common sense model step by step for each operation. SpecElicitor produces a common sense model to model the behaviors that the tester performed in the end.

According to our assumption, we find important features of a type of application by overlapping their common sense models. To test these important feature in higher intensity, we design an algorithm by randomly selecting weighted actions. We also design a test case generator output test cases following the algorithm and a test case executor runs these test cases and produce a test report. The test report contain not only the result of running test cases, but also a simple evaluation according to whether the traces are passed, failed, or indicating that our common sense model is inadequately to model the application.

7.2 Limitation

The meaning of a word might be various for different persons. So either defining normalized terms or using them could be ambiguous. If the common sense models of a kind of applications are built by different testers, the overlapped model of these models will be meaningless in worst. Moreover, if a tester chooses a wrong term to describe a screen or an action, it will cause the same circumstance. So labor training and fault tolerance would becoming a problem.

Another limitation is identification of screens and actions. We use abstraction to filter out redundant information and comparing remain for now. Nevertheless, if a screen has a very slightly difference and cannot be gotten rid of by abstraction, the screen would be treat as a new, different screen. This will acquire more labor works to give meanings to the screen and actions.

7.3 Future Work

One of future works is trying to make SpecElicitor more powerful and efficiency. For now there are some technical problems could not be solved. For instance, scrolling the Android device is one of the problems. We can perform a user input through ADB[17] as touching down at one position, moving to another position in a predefined number of seconds, then lifting the finger. However, the scrolling distance in the application is considering the speed of moving. It is not easy to set a suitable number for the time of moving. Therefore we do not support the scrolling action in SpecElicitor. It would let SpecElicitor skip some functionalities and build an incomplete common sense model.

Let the user of SpecElicitor can alter the normalized terms of a labeled screen or action can reduce wastes of labors. It is nearly impossible to ensure that two snapshots between sequences of user input have equal status. Because internal changes may not be revealed in the screen. That makes re-doing an action being more complicated. If a user chooses normalized terms mistakenly and SpecElicitor has no mechanism for modification, the common sense model will contain incorrect concepts. A modification mechanism can help users prevent producing a wrong common sense model and re-labeling the application intermediately.

Another improvable component is usages of our common sense database. Because we already get the meanings for each screen and action, we might be able to make a sequence of transition meaningful, too. Then use skills of data mine to find the valuable sequences of transition which might be able to correspond user behavior in real-life. For example, a process for “Login” consist of many user inputs and different screens. To successfully login an account, it has to input specific sequence of actions. The sequence would be a critical function in applications containing “Login” feature.

Our ultimate goal of using common sense models is automatically or semi-automatically identification applications. For example, we know the type “Notepad” has “Note”, “Note List” for screens. If there is a new application, we can simply set the type of the application as “Notepad”. There will be an algorithm automatically identify which screen is “Note” and which screen is “Note List”. And we can test the application based on common sense without any intervention from labors. It may require knowledge of machine learning, natural language processing, and computer vision.

REFERENCE

[1] "UI/Application Exerciser Monkey." [Online]. Available: http://developer.android.com/tools/help/monkey.html.

[2] D. Olenick, "Apple iOS And Google Android Smartphone Market share Flattening: IDC," 2015. [Online]. Available: http://www.forbes.com/sites/dougolenick/2015/05/27/apple-ios-and-google-android-smartphone-market-share-flattening-idc/.

[3] A. M. Memon, M. Lou Soffa, and M. E. Pollack, "Coverage criteria for GUI testing," *ACM SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, p. 256, 2001.

[4] B. N. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Autom. Softw. Eng.*, vol. 21, no. 1, pp. 65–105, 2014.

[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. Ta, and A. Memon, "MobiGUITAR -- A Tool for Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, pp. 1–1, 2014.

[6] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," *Proc. 2013 9th Jt. Meet. Found. Softw. Eng. - ESEC/FSE 2013*, p. 224, 2013.

[7] T. Ostrand, A. Anodide, H. Foster, and T. Goradia, "A visual test development environment for GUI systems," *ACM SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 82–92, 1998.

[8] "Sikuli." [Online]. Available: http://www.sikuli.org/.

[9] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," *Proc. 28th Int. Conf. Hum. factors Comput. Syst.*, no. Figure 1, pp. 1535–1544, 2010.

[10] E. Manavoglu, T. Building, and C. L. Giles, “Probabilistic User Behavior Models,” 2003.

[11] K. Radinsky, K. Svore, S. Dumais, J. Teevan, A. Bocharov, and E. Horvitz, “Modeling and predicting behavioral dynamics on the web,” *Proc. 21st Int. Conf. World Wide Web - WWW ’12*, p. 599, 2012.

[12] A. Li, Z. Qin, M. Chen, and J. Liu, “ADAutomation : An Activity Diagram Based Automated GUI Testing Framework for Smartphone Applications,” *IEEE Int. Conf. Softw. Secur. Reliab.*, pp. 68–77, 2014.

[13] M. Utting, A. Pretschner, and B. Legeard, “A Taxonomy of Model-Based Testing,” no. April, pp. 1–18, 2006.

[14] “TEMA.” [Online]. Available: http://tema.cs.tut.fi/index.html.

[15] T. Takala, M. Katara, and J. Harty, “Experiences of system-level model-based GUI testing of an android application,” *Proc. - 4th IEEE Int. Conf. Softw. Testing, Verif. Validation, ICST 2011*, pp. 377–386, 2011.

[16] “Semantic Network - Wikipedia.” .

[17] “Android Debug Bridge.” [Online]. Available: http://developer.android.com/tools/help/adb.html.

[18] “UIAutomator.” [Online]. Available: https://developer.android.com/tools/testing-support-library/index.html#UIAutomator.

[19] “Omnidroid.” [Online]. Available: https://code.google.com/p/omnidroid/.

[20] “Genymotion.” [Online]. Available: https://www.genymotion.com/.

[21] “HTC Desire 820.” [Online]. Available: http://www.htc.com/tw/smartphones/htc-desire-820-dual-sim/.