Component-based Modeling for Android Application Testing

by

Yatian Gao

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (MSc)

in

Computer Science

University of Ontario Institute of Technology

Oshawa, Ontario, Canada

April 2017

Abstract

The rising popularity of Android and the component-based structure of its apps have motivated the need for automated model-based testing techniques on Android platform. Prior researches have primarily focused on the GUI-based model of Android apps. GUI-based model only includes Activity targeting graphical user interfaces. It neglects other components such as Service and Broadcast Receiver in the Android Development Framework. Although the GUI-based model testing has achieved a good testing result targeting the graphical user interface, its effectiveness has been decreasing as Android applications become more complex in both functional behaviors and component-based structure. This phenomenon challenges the feasibility of currently existing model-based testing on Android platform.

To address the challenges mentioned above, we propose a component-based approach of automated model generation for model-based testing on Android platform in this thesis. First, we extend the state definition in the model. Activity, Service and Broadcast Receiver are abstracted into the component-based model as states. Newly introduced states can depict the behaviors of a given app in a larger scope for better descriptive modeling and input generation. Second, we extend transition definition, and also propose a static mapping transition builder for transition construction across different kinds of components. Then the event sequence generator & cluster is proposed to generate proper test sequences for testing. The event cluster assists the input generation of the component-based model testing.

Also, we present CamDroid, a tool implementing the proposed approach for Android apps testing. Lastly, our experiments have corroborated CamDroid's ability to build a model connecting components including Service, Activity and Broadcast Receiver. It can overcome the new challenges of Android apps in model-based testing. As a result, component-based model can achieve better performance in real model-based testing in terms of code coverage, comparing to the traditional GUI-based model testing.

Acknowledgments

I have been very fortunate to finish my Master's degree at the University of Ontario Institute of Technology (UOIT). Over last two years, people around me provide both academic intelligence and adequate learning resources to help me improve in academic skills.

I would like to express my deep and sincere gratitude to my supervisor Prof. Xiaodong Lin for his kindness, encouragement, continuing guidance, and support during my research. His enthusiastic attitude towards life and academic research always encourages me to be a better student.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background & Motivation

With a market share of 86.2% in smartphone OS and a device shipment over 1.4 billion [1] [2], Android has become one of the dominant operating systems in the world. Market domination is reflected in both hardware and software aspects.

As for hardware, Android is designed to run on various kinds of devices such as vehicle navigation devices, video players, smart TVs, Android Wear[3] or smart surveillance cameras. Aside from the dominant market share in the smart phone and tablet industries, Android also shows its superiority in the niche market [4][5].

As for software, Google Play has more than 2.2 million apps available for download which ranks first among all application stores such as AppleStore, Microsoft App Store. This number will increase if third party app stores are included.

But Google cannot guarantee the quality and benignity because of the large amount of the applications in GooglePlay. So it is critical to ensure that an Android application is in high quality and functional benignity.

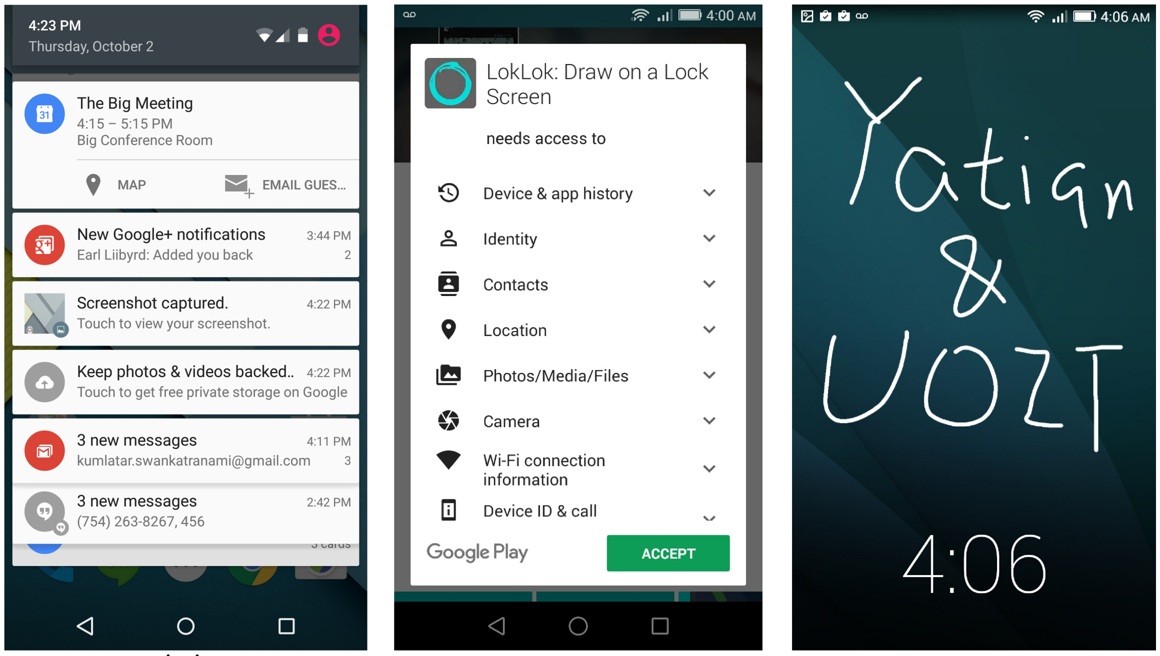
Although the Android apps are diverse in functions. All of the Android applications are relying on four components [6] structurally in the Android Development Framework. Meanwhile all functions are implemented in the components indirectly. The component consisted structure of Android apps enable to perform the model-based testing to Android applications. To retrospect prior researches, almost all works are using the GUI-based model while concentrating on graphical user interfaces and imitating user inputs for model-based testing.

The GUI-based model has proven to be feasible and efficient in Android apps testing in the past few years. This is because the Android apps structure is relative simple years before. However, the GUI-based model testing's effectiveness keeps decreasing as the graphical interface and functional structure become more complex as the development of Android system advances. As a summary, Android's developments in recent years has challenged the model-based testing in two aspects. The first way, Android apps' designs are more responsive and user-oriented. For example, the notification bar and lock screen are widely used nowadays. The second way, many new events are introduced into Android systems while more sensors and external devices are embedded into Android devices. It enriches the interactive patterns in Android applications.

As for newly embedded hardware, dual cameras, finger print scanners and baroceptor are becoming standard configurations in smart phones. The Changhong H2 smart phone even has a molecular spectrum sensor for future smart home markets. It will be a trend to embed precise sensors into smart phone in the near future.

As for the software design, the app's functional structure also has expanded rapidly in recent years. Three typical phenomena can reflect the problem of structure complexity in graphical user interfaces.

As shown in Figure 1.1 (a), the first phenomenon is the control race of the notification bar. Most apps push notification to users for user loyalty, while some others


4:23 PM
Thursday, October 2

The Big Meeting
4:15 – 5:15 PM
Big Conference Room
MAP EMAIL GUES...

New Google+ notifications 3:44 PM
Earl Liibyrd: Added you back 2

Screenshot captured. 4:22 PM
Touch to view your screenshot.

Keep photos & videos backed.. 4:22 PM
Touch to get free private storage on Google

3 new messages 4:11 PM
kumlatar.swankatranami@gmail.com 3

3 new messages 2:42 PM
(754) 263-8267, 456

4:00 AM

LokLok: Draw on a Lock Screen

needs access to

Device & app history
Identity
Contacts
Location
Photos/Media/Files
Camera
Wi-Fi connection information
Device ID & call

Google Play ACCEPT

Figure 1.1: Challenges in Android application's structure

use it as a visual controller bar for background tasks. A notification bar can be fully occupied within 5 minutes after powering on the smart phone, as based on our repeated experiments. However, this phenomenon cannot be described in GUI-based model testing.

Figure 1.1 (b) shows the phenomenon of permission abuse. As developers want the users to pay more attention to their app, they are likely to require more permissions and monitor more system statuses for invocation anytime. These system event related interactions are escaped from GUI-based model investigation.

Figure 1.1 (c) shows a lock screen occupied by Loklok [7]. The lock screen is the most important interactive user interface nowadays and all apps want to occupy this precious interactive entry. The user experience will become a disaster when the lock screen is occupied by multiple apps. All these challenges are occurred in recent years. Traditional GUI-based model testing cannot operate effectively when facing these challenges. GUI-based model becomes insufficient in model-based testing. It motivates the need to increase effectiveness and efficiency of model-based testing.

Aside from the graphical user interface itself, some new challenges are introduced into the apps. Many new interactive patterns are shown in today's Android applications.

The GUI-based model looks insufficient to deal with the new born challenges in the past few years. As a result, the GUI-based model need to be ameliorated for more challenging model-based testing tasks.

Although many efforts have been made to improve the performance of the GUI-based model. The effectiveness of GUI-based model still cannot increase dramatically. Based on our research, four challenges cannot be overcome by the GUI-based model. These challenges may be the main factors which affects the GUI-based model's effectiveness.

They are summarized as follows:

- Firstly, the GUI-based model only includes data from graphical user interface layouts and Activities, neglecting behaviors defined in Services and Broadcast Receivers. It may cause defects of the model in describing behaviors in system under test [9].

- Secondly, GUI-based model abstracts apps as single entry and graphical interface driven finite state machine. Actually Android application can have more than one program entry or exit. Although launcher Activity is always defined as the only program entry in the GUI-based model. Normally some system or custom event can also behave as program entries & exits. This single entry abstraction sacrifices code coverage to seek the model's simplicity in the early Android versions, although it works well for early versions of Android.

- Thirdly, Activity can be started in Service or Broadcast Receiver. As a result, some transitions defined in the GUI-based model cannot be fully covered in the traditional ways. It may cause transition deficiency in the GUI-based model. Furthermore, almost all previous researches are defining graphical user interface transformation as transition in the GUI-based model. As a consequence, some vital transitions may be neglected by this definition. This problem causes a serious consequence that the constructed model cannot reflect the app's behaviors authentically.

- Fourthly, not all graphical user interface layouts are controlled by Activity. Some graphical user interface layouts are controlled by Service or triggered by Broadcast Receiver. Neglecting Service and Broadcast Receiver can cause an increasing rate of invalid test inputs. It also leads to the decrease of code coverage consequently.

These four challenges motivate us to increase the model-based testing effectiveness as well as model quality through enhancing existing GUI-based model testing techniques.

1.2 Study Necessity & Practicability Survey

Before introducing the proposed component-based model, we need to answer one important question to prove the necessity of our study :

- Is it necessary to introduce Service and Broadcast Receiver related states into the model?

To prove the necessity of extending the GUI-based model as well as answer the question above, we designed two surveys to investigate the necessity for introducing new components into the testing model. First survey is concentrating on the software aspect. We downloaded 100 most popular applications from GooglePlay. Our surveys are concentrating on the usage percentage of Activity, Service and Broadcast Receiver in Android applications. These are the focus because different applications register different number of various components. So we define ratios between number of Activity and Service or Broadcast Receiver.

$$Ratio_{Service} = \frac{SumofServicesinApp}{NumberofActivityinApp} \tag{1.1}$$

$$Ratio_{BroadcastReceiver} = \frac{SumofBroadcastreceiversinApp}{NumberofActivityinApp} \tag{1.2}$$

$$f(A, S, B, n) = \frac{1}{n}\sum_{i=1}^{n} Ratio_{Service} + Ratio_{BroadcastReceiver} \tag{1.3}$$

From equation 1.1, we can see the ratio definition of Service while the equation 1.2 shows the ratio definition of Broadcast Receiver. If the ratio equals to zero, there is no Service or Broadcast Receiver in the application. If the ratio equals 1, there are the same numbers of Activity and Service or Broadcast Receiver inside an application. Otherwise Service or Broadcast Receiver has a larger number than Activity in apps. In simple terms, the larger the ratio, the more importance of other components in an application. The equation 1.3 shows the average expectation of Service and Broadcast Receiver's importance, where A stands for the Activity number, S stands for the Service number, B stands for the Broadcast Receiver number and n stands for the total number of the components registered in the Application. Figure 1.2 shows the coverage expectation of Service & Broadcast Receiver's importance based on our survey.

As we can see in figure 1.2, all applications register the Service component while

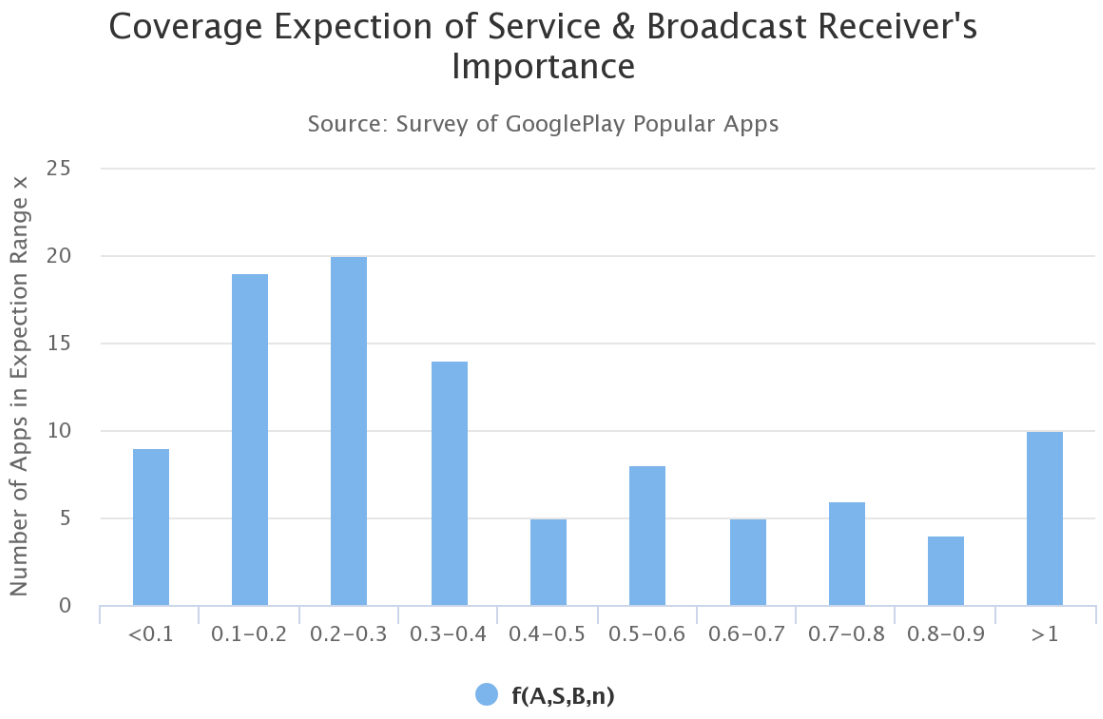


Figure 1.2: Coverage expectation of Service & Broadcast Receiver's importance.

most of the applications register Broadcast Receiver. Meanwhile, around 10% of applications have more Service and Broadcast Receiver than Activity in number. Most of them are popular and widely installed on Android devices worldwide. They are the Opera browser[10], Google Message[11], Google Maps[12], Chrome browser[13], UC browser[14], and clash of clans[15]. It can be evident by the survey that Service and Broadcast Receiver are widely used in Android applications.

We can conclude the necessity of introducing Service and Broadcast Receiver into the testing model based on the first survey. The survey provides an overall component usage investigation of Android software. Then we need to investigate the interactive patterns in Android hardware.

The second survey is concentrating on the hardware aspect. More interactive modes aside from the graphical user interface interactions are flourishing nowadays.

As comparison of the concentration on the GUI-based model, Android devices had reached 1.4 billion shipments in last year while Android has been installed in more than 4000 different kinds of devices including smart phones, vehicle navigation devices, video players and smart surveillance cameras. Inputs other than graphical interaction are playing an important role in Android system. Meanwhile, these inputs are out of description of the traditional GUI-based model. For instance, Pokemon Go [55] needs geographic data as continuous user input. Rinna [56], developed by Microsoft can provide natural language interactive interface, both in oral and written format. All these functions are deeply Service and Broadcast Receiver related.

Based on discussion of different aspects in the Android environment, more and more challenges are affecting the effectiveness of the GUI-based model. So we can say that now is a proper time to extend the conventional GUI-based model by introducing Service and Broadcast Receiver into a new proposed component-based model to describe the model's behaviors more preciously and authentically. In doing so, model-based testing of Android systems can be more effective and efficient.

1.3 Objectives & Methodology

1.3.1 Objectives

Our study has several objectives for the model-based testing optimization on Android platforms.

The first objective of this thesis is to do a complete survey of new challenges in GUI-based model testing techniques. So we can have a clearer view of challenges occurring in Android systems in recent years.

The second objective of this thesis is to propose a component-based model testing approach. It can replace the GUI-based model testing and become a new standard of model-based testing on Android platform. This approach can overcome the challenging problems from the surveys above and achieve a better coverage in model construction and testing.

The third objective of this thesis is to evaluate the proposed approach through an implemented tool, CamDroid. Evaluation will show whether it can achieve a better code coverage and succeed to trigger actions in components of Service and Broadcast Receiver precisely with less input tests relatively. CamDroid also reduces the test sequence number and avoids invalid test samples in the sequences. The coverage expansion is important, as it not only helps to trigger more behaviors, but also improves the test quality. The reduction strategy of input test generation enables both to execute and traverse efficiently. Especially, we evaluate the component-based model's ability to overcome the new challenges in the currently existing model-based testing.

1.3.2 Methodology

Thus all the model-based testing researches targeting the Android platform are abstracting apps as finite state machine [8] for modeling, therefore the method of improving the model-based testing effectiveness is laying under the finite state machine. In our study, our methodology inherits the finite state machine organized model.

Figure 1.3 shows a finite state demo of a door. We can see the key factors of a finite state machine are state, initial state, transition and conditions for each state. The first two factors can be classified as state definition while the last two factors can be classified as transition definition. State definition refers to the state abstraction strategy. Transition definition refers to the state definition in the model of SUT [9]. As a summary, we change the state and transition definition to upgrade the model's state abstraction and structure.

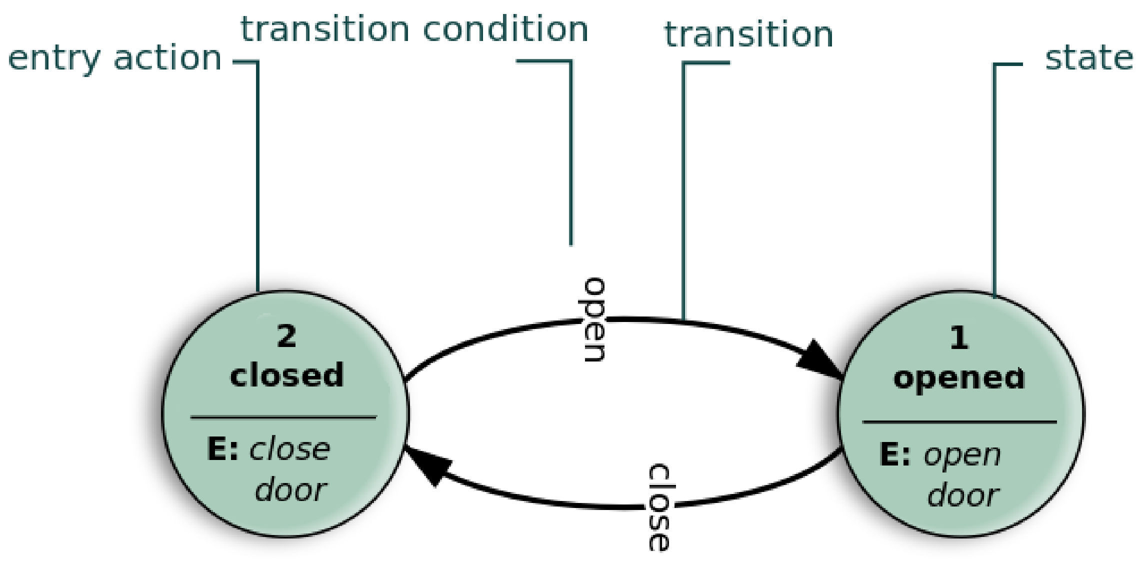


Figure 1.3: Finite state machine demo

The component-based model is an abstraction of Android applications used to assist input generation which includes Activity, Service, Broadcast Receiver and graphical user interface.Unlike GUI-based model, mainly focused on Activity that is associated with the user interface(UI). Component-based model is an implementation of finite state machine which includes UI-state, BR-state, SER-state and UI-SER-state as states. Transition is the conditional trigger to transit between different states defined in component-based model.

In this thesis, we present CamDroid (**C**omponent-based **A**utomated **M**odel Testing for An**DROID**) , a fully automated tool for component-based model generation. It is also an implementation of our proposed approach. Given an Android APK file, CamDroid can first generate a GUI-related states automatically using methods from [16][17]. While it can generate Service and Broadcast Receiver related states, it then relates the component-based model using static-mapping transition builder. Finally, generated Service or Broadcast Receiver related inputs can be combined with GUI related inputs by event sequences generator & cluster. By using component-based modeling strategy, CamDroid can achieve better code coverage with reasonable computational resources. Comparing to a GUI-based model, the component-based model can describe more behaviors in applications at a more authentic level.

1.4 Contributions

The contributions of this thesis can be summarized as following:

- Firstly, we proposed a new component-based model which integrates GUI layout, Service, Broadcast Receiver as well as Activity into the model. It extends model's state definition in model-based testing. Newly introduced BR-state, SER-state, UI-SER-state can describe behaviors in Service and Broadcast Receivers. This is achieved by static code analysis using both soot analysis and smali code audit. The proposed state definition increase model's description ability comparing to traditional GUI-based model.

- Secondly, we proposed a static-mapping transition generation method which enables to construct transitions across different states in component-based model. The method extends transition definition based on GUI-based model and can guarantee the integrity of model's structure. It improves the authentication of model's structure in model-based testing.

- Thirdly, we proposed an event sequence generator & cluster method to generate test sample sequences suitable for automatic testing. Event cluster is an optimized test sequence generation method. Unlike traditional sequence generation method, we generate GUI-related sequences and Service or Broadcast Receiver related sequences respectively. Then combine them together through model's exploration. It can decrease the sequence number and increase execution efficiency comparing to traditional sequence generation.

- Fourthly, the proposed approach can overcome the new challenges in the Android system. Many intractable problems in GUI-based model such as Isolated State, Pseudo Graphical User Interface Interaction are solved by proposed approach. Its model has a better performance in authentic description of app's behaviors.

- Finally, the proposed approach achieves a better code coverage than traditional GUI-based testing techniques in the real model-based testing. It has been proven by our experiments. Based on our evaluation, it achieves a code coverage of 68.9% comparing to 31.4% of random testing and 44.8% of GUI-based model testing.

1.5 Thesis Organization

This thesis consists of six chapters. Chapter 1 presents an introduction to this work and the other chapters are organized as follows:

- Chapter 2: In this chapter, a number of testing techniques on Android systems are surveyed. We also outline the features, advantages and disadvantages of different techniques. The components defined in Android development framework are also surveyed in this chapter.

- Chapter 3: This chapter describes an detailed investigation of challenges in GUI-based model testing techniques which can decrease the GUI-based model's effectiveness. Then we introduce an illustrative application released in Google-

Play as a motivating example to reiterate the limitation of GUI-based model and demonstrate the superiority of the proposed component-based model.

- Chapter 4: This chapter depicts the component-based model generation approach in details.
- Chapter 5: This chapter evaluates the proposed approach in effectiveness and performance.
- Chapter 6: This chapter briefly summarizes the key outcomes of our approach, and offers some suggestions for future directions of this work.

Chapter 2

Components & Testing Techniques on Android Platform

In this chapter, we discuss two pieces of background knowledge needed in our study. For the first part, we introduce the commonly used Android application's components and their life cycles. They are Activity, Service & Broadcast Receiver. For the second part, we introduce the concept of testing techniques & the existing model-based testing tools.

Then we provide a brief overview of existing works on test input generation techniques using random, model-based and systematic strategies. The tools can be classified into two categories. Our proposed approach and implementation have been built on these dependent projects and tools. The related projects provide the fundamental functions to our implementing tool. The related projects provide beneficial hints to our approach.

2.1 Android Development Framework & Components

In this section, we will focus on the life cycle and interaction pattern of components in Android development framework. We concentrate on Activity, Service, Broadcast Receiver in this section.

First of all, we need to have an overall view of the Android system architecture firstly. Components are defined in the Java API framework as shown in figure 2.1. Four components are key factors in the Android application development.

Application components are the essential building blocks of an Android app. Each component is an entry point through which the system or a user can interact with an application. As shown in figure 2.2, four components are defined into the Android framework; They are Activity, Service, Broadcast Receiver and Content Provider. Each type serves a distinct purpose and has a distinct life cycle that defines how the component is created and destroyed.

2.1.1 Broadcast Receiver

Broadcast Receiver is a component in the Android system which can respond to broadcasts from various sources. Broadcast Receiver is not well studied in the traditional model-based testing techniques, because traditional model-based testing concentrates on the GUI-based model. Although Broadcast Receiver is not a direct controller of the graphic user interface, it still can control the application's behaviors and state transitions indirectly.

From figure 2.3, we can see the life cycle of the Broadcast Receiver. Broadcast Receiver should be registered in the Android system before use. After registration, the Broadcast Receiver is managed by the Android system. If the trigger condition

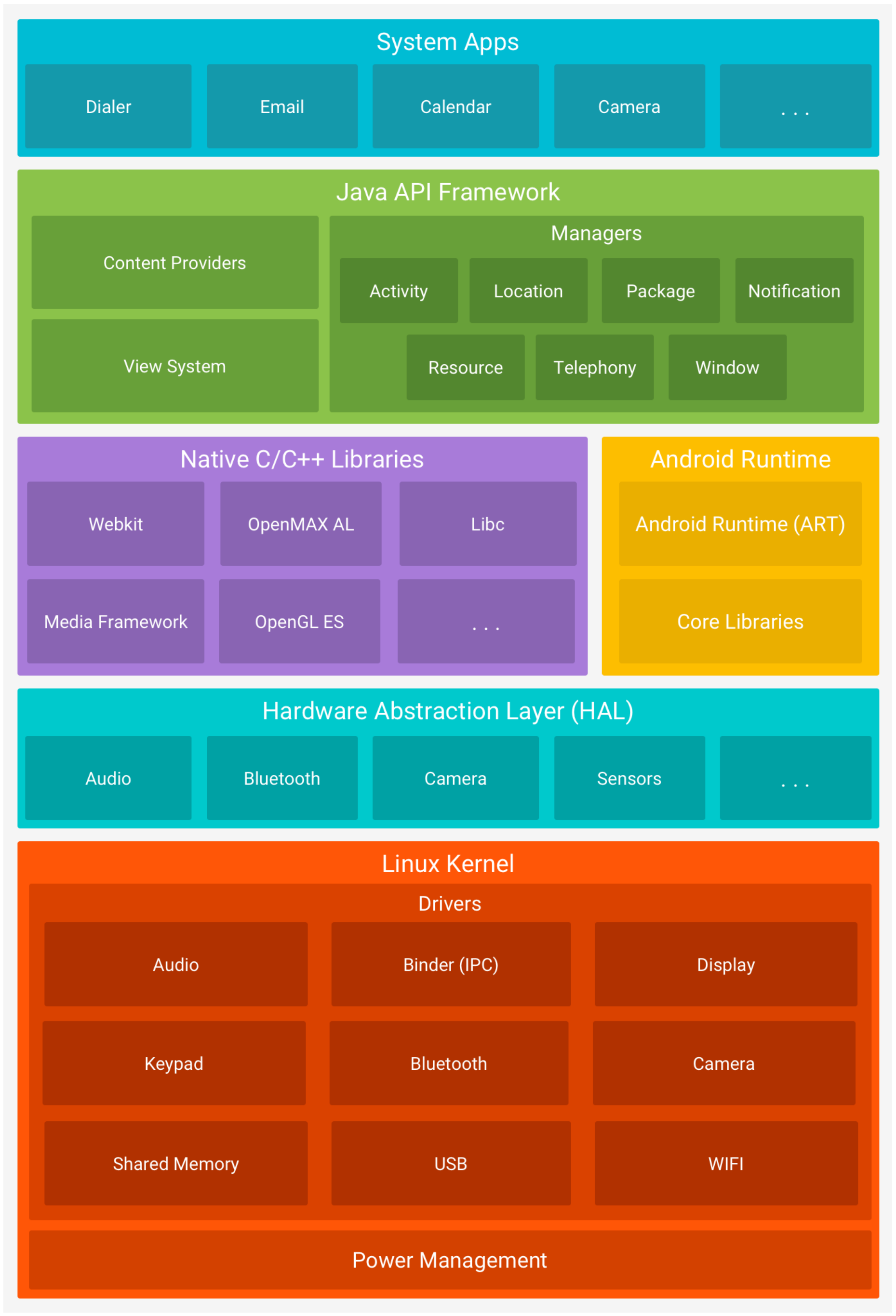


Figure 2.1: Android system architecture

of Broadcast Receiver is met at a certain time point, the function onReceive() will be invoked and executed. After finishing execution the Broadcast Receiver will stay

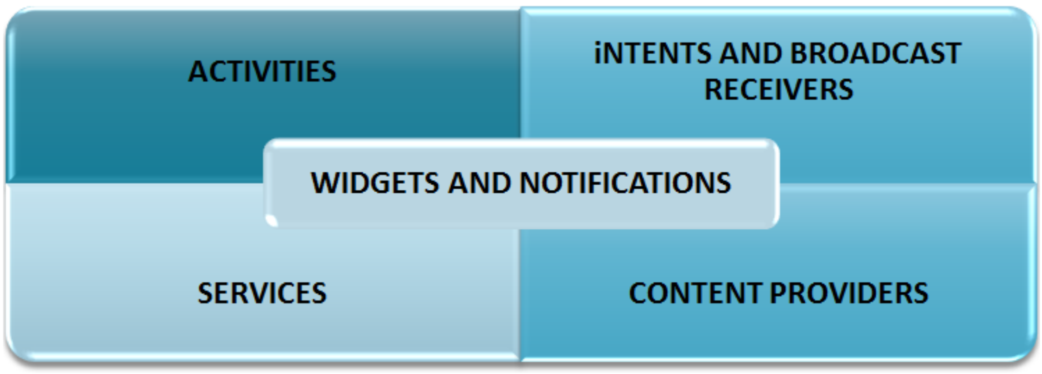


Figure 2.2: Components in Android framework

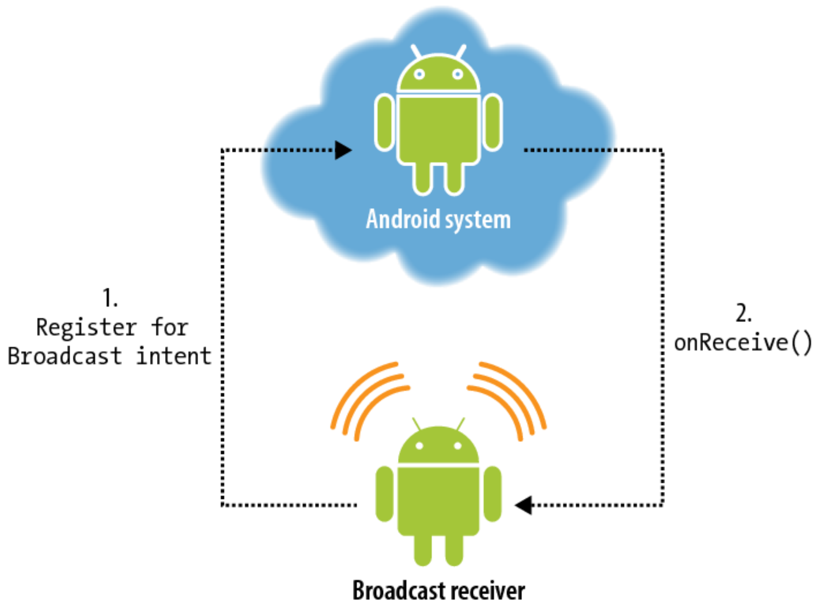


Figure 2.3: Life cycle of Broadcast Receiver

in the state of standby. The method will be invoked again if the trigger condition is being met.

Although the life cycle of Broadcast Receiver is relatively simple. There are still various of methods in registration and execution. There are two type of registration methods in Broadcast Receiver: Static & dynamic registration. Static registration registers Broadcast Receiver in the AndroidManifest file and invokes itself when the execution condition has been met. Because of the mechanism of static registration, static registered Broadcast Receiver can not be included into the call-graph of a single entry GUI-based model. Dynamic registration can register certain Broadcast Receiver everywhere in the application.

Aside from the registration methods, Broadcast Receiver can be divided into system Broadcast Receiver, custom Broadcast Receiver and local Broadcast Receiver. That means Broadcast Receiver not only behaves as program entry, it can also execute certain behaviors. More importantly, the Broadcast Receiver can be used as transition trigger in model-based testing. Due to the flexibility of the Broadcast Receiver's usages, Broadcast Receiver is versatile and plays roles in many apps' design demands. In this sense, Broadcast Receiver plays a critical role in the Android application model.

2.1.2 Service

Service is a general-purpose entry point for keeping an app running in the background for all kinds of reasons. It is a component that runs in the background to perform long-running operations or to perform work for remote processes, however , Service can interact with users graphically. First, Service can partake some time consuming tasks such as Internet connection through binding with a Activity. Second, Service can control graphic user interface directly as well.

Figure 2.4 shows the life cycle of Service. From the figure we can see that Service can be categorized as unbounded service and bounded service. There are three critical methods in the life cycle. They are onCreate(), onStartCommand() and onDestory(). When a service is executed the first time, the onCreate() will be invoked. Then every time the Service is invoked, the onStartCommand() will be loaded and executed. When all the tasks have been executed, the onDestroy() function will be executed to

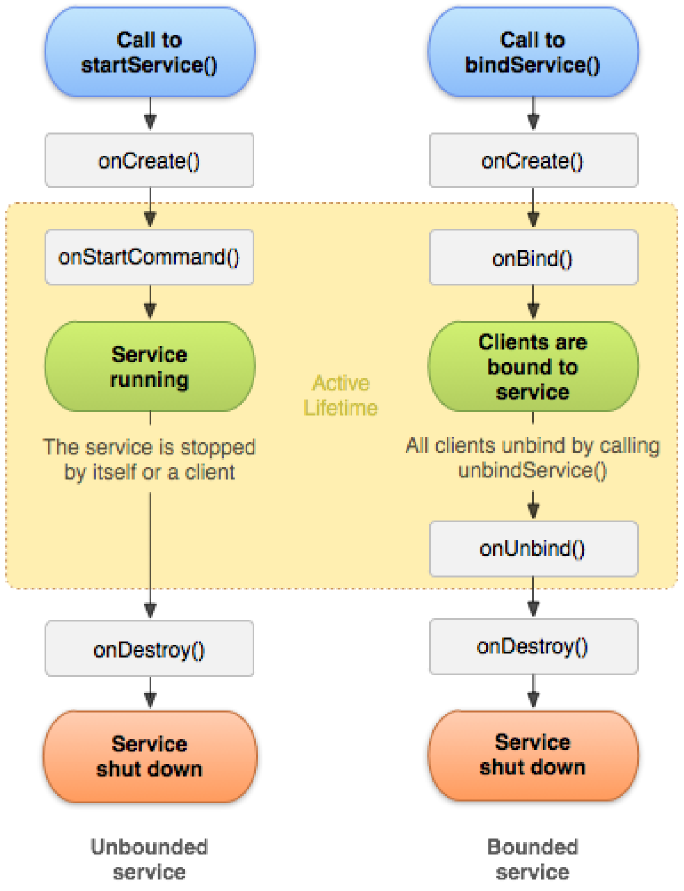


Figure 2.4: Life cycle of Service.

destroy the Service to end the life cycle.

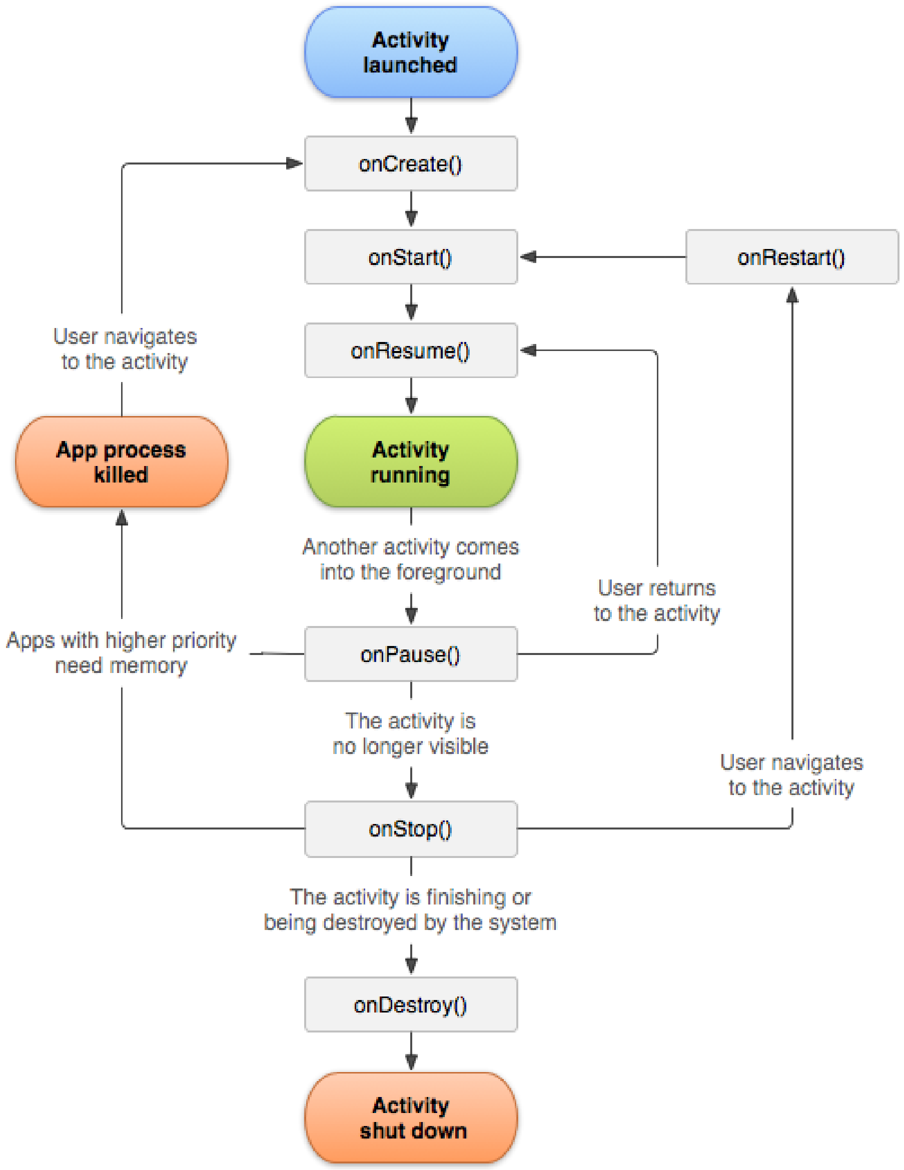
Service's life cycle is more complex than Broadcast Receiver's. Meanwhile Service's life cycle is similar with the life cycle of Activity. Compared to the functions defined in Broadcast Receiver, more functions can be accomplished by Service.

First, Service can control graphical user interfaces through the notification bar. Alternatively, Service can control graphical user interface through message interaction.

Second, Service can partake tasks of the graphical user interface through the Service connection mechanism. Normally, Service is registered statically in the application.

2.1.3 Activity

Activity is a component which deals with the behaviors in the graphical user interface. It is usually abstracted as a UI-state in the GUI-based model.



Activity launched
onCreate()
onStart()
onRestart()
User navigates to the activity
onResume()
App process killed
Activity running
Another activity comes into the foreground
User returns to the activity
Apps with higher priority need memory
onPause()
The activity is no longer visible
User navigates to the activity
onStop()
The activity is finishing or being destroyed by the system
onDestroy()
Activity shut down

Figure 2.5: Life cycle of Activity.

From the figure 2.5 we can see the different states in the life cycle of Activity. They are onCreate(), onStart(), onResume(), onPause(), onStop(), onDestroy and onRestart(). OnCreate() is the entry of the activity's life cycle. This method will be invoked when the Activity is initialized. Some initialization tasks will be executed such as layout load and Service binding. The function onStart() will be invoked when the Activity becomes visible in the application. The Activity will pause if the onPause() function has been triggered, while onResume() is triggered when the activity goes back to work. The onStop() function can be triggered when the Activity becomes invisible. The Activity can restart again from the stop state. Finally, the Activity can be eliminated after onDestroy() is triggered.

Most GUI-based model's status are abstracting from the Activity, but still there are more works to be done to ameliorate the traditional GUI-based model. All the components introduced above should also be included into the component-based model using the model-based testing techniques, but it is currently missing from existing model-based testing for Android applications.

After introducing the commonly used components in Android development framework, we introduce the testing techniques on the Android platform in next section.

2.2 Testing Techniques on Android Platform

Testing is an indispensable process for all application development. However, testing is labor-intensive and costs a lot of resources. It is meaningful if some software testing tools can reduce resource cost and improve testing effectiveness. This effort in the Android platform has started since the release of the Android system. Testing practices on the Android platform focus on three categories. They are fuzz testing, systematic testing and model-based testing. First we introduce the fuzz testing and systematic testing briefly, then we will concentrate on the model-based testing's development on Android platform.

2.2.1 Fuzz Testing & Systematic Testing

Fuzz testing is a software testing technique which obeys random input generation strategy. Fuzz testing is often automated or semi-automated. Implemented tools of fuzz testing usually provide invalid, unexpected and random input to the computer program. The program is then monitored by the system to find bugs such as crashes, potential memory leaks and logic bombs[22][24]. Many projects are also applying the fuzz test to the Android platform. The fuzz testing has an obvious advantage, the technique does not need to consider the interface structure of the program. The fuzz testing is highly efficient in input generation. At the same time fuzz testing is not efficient enough to generate precise test inputs to achieve a satisfactory code coverage. Fuzz testing is now widely used in application testing. More and more tasks should be handled by automated tools in a systematic way.

Systematic testing is a testing technique which applies the systematic method such as call path resolver to test all possible methods in the application. The goal is to test the software against a variety inputs in order to find as many defects as possible. The advantage of the systematic testing is reducing redundancy in input generation. It can also reduce the numbers of the test samples compared to the random strategy of testing. The problem of systematic testing is the path explosion when the control flow in the application becomes too complex to explore.

2.2.2 Model-based Testing

Model-based testing (MBT) is a light-weight formal method which uses models of software systems for derivation of test suites [18]. Models can be used to represent the desired behavior of a System Under Test (SUT), or to represent testing strategies and a test environment. Model-based input generation is an approach in which a model of the app under test is used as a reference to generate test cases. The model is abstracted from the behaviors of the app and possible sequences of transitions between states inside the SUT [9]. As for the structure and representation of the model, the model-based input generation can be classified into three categories. They are the axiomatic approach, the Finite State Machine approach and the labeled transition system approach.

The axiomatic approach of model-based testing are based on logic calculus. Labeled transition systems (LTS) are common for describing the operational semantics of process algebra [18]. These two approaches do not apply to the testing of the Android system. The finite state machine approach is commonly used in model-based testing of the Android platform. As a result, the finite state machine represented model-based approach is pivotal in the Android system.

The finite state machine represented model in model-based testing is used in different kinds of software before the release of Android system. [38] applies model-based testing to construct a graphical user interface model towards desktop software. [37] [39] studies the model-based testing on web applications. Arilo C [41] and Mark [44] presents a detailed survey on model-based testing before the release of Android system in 2007.

After the release of iOS and Android system in November 2007, researchers paid more and more attention to model-based testing on mobile platforms. The iCrawler [40] is a reverse-engineering tool for iOS mobile applications which uses a state-machine model. A.Jaaskelainen [43] also studies graphical user interface modeling in the Symbian system.

The model-based approach is also widely studied on the Android system. There are four development stages of model-based testing on Android platforms.

At the beginning, model-based testing techniques are transplanted into the Android system. Takala [42] studies the model-based testing application on the Android system and summarizes their experiences in model-based graphical user interface testing of Android applications. At this stage, model-based testing is manually performed.

At the second stage, researchers focus on the automatic GUI-based model testing. The most convenient method is to transplant the existing tools into Android systems. Amalfitano [45] ameliorates the GUI Ripping tool and presents Android Ripper as an automatic model-based testing tool. MobiGUITAR [23] is another model-based testing tool on the Android system.

At the third stage, Wei Yang [16] proposes a grey-box approach for automated GUI-model generation of mobile applications. It is a fully automated approach of model-based testing on the Android platform. The proposed tool achieves a better performance than traditional model-based GUI testing.

Figure 2.6 shows an illustrative example and its GUI-based model in [16]. From figure 2.6, we can see different graphical user interfaces are abstracted as states in the GUI-based model. The elements of user interface and registered event listeners are included into states.

The GUI-based model testing has achieved a satisfactory performance in the Android system in the early years, but it still has some problems in dealing with complex graphical user interfaces. In the fourth stage, researchers combine the model-based testing and combinatorial testing to improve the test effectiveness. M[agi]C [46] is

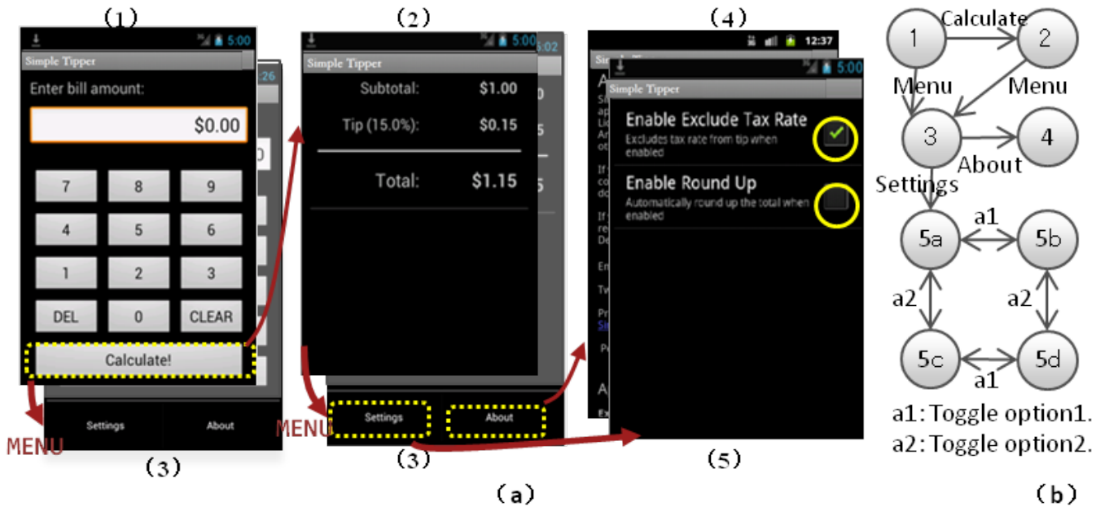


Figure 2.6: Overview of SimpleTipper(a) and its state graph(b).

a tool combines the model-based testing and combinatorial testing to get a better result. TrimDroid is another tool which combines the model-based testing and combinatorial testing[57]. Moreover, TrimDroid is effective in complex graphical user interface processing.

The GUI-based model also plays an important role in dynamic model exploration. A^{3}E-Depth-first[47] is an implementing tool which searches the dynamic model in depth. SwiftHand [26] is an automated Android GUI testing tool. The tool uses machine learning to learn a model of the target app during testing, then uses the learned model to generate user inputs that visit unexplored states of the app, and uses the execution of the app on the generated inputs to refine the model. PUMA [27] is a programmable UI-Automation Framework for dynamic app model analysis.

From the survey of model-based testing, the model-based testing is developing to meet the newly introduced testing requirements. It is still insufficient to perform satisfactory model-based testing. New method of model-based testing should be proposed for better testing effectiveness.

2.3 Related Work

After knowing the basic concepts of components and testing techniques on Android platform. We need to know some related projects in input generation techniques for Android application testing. Related work can be separated into two categories: dependent projects and related input generation projects following different strategies.

2.3.1 Dependent Projects

CamDroid is developed on a basis of other projects. They are ClassyShark[19], Soot[20] and TrimDroid[17].

ClassyShark is an Android executable browser. It can extract meta data and perform navigation and search functions. In CamDroid, we use ClassyShark to extract meta data for model's blue print construction.

Soot is a tool used to analyze, instrument and visualize Java code. In CamDroid, Soot is used to perform the static analysis and instrument functions.

TrimDroid is a tool for automated input generation using GUI-based model testing technique.

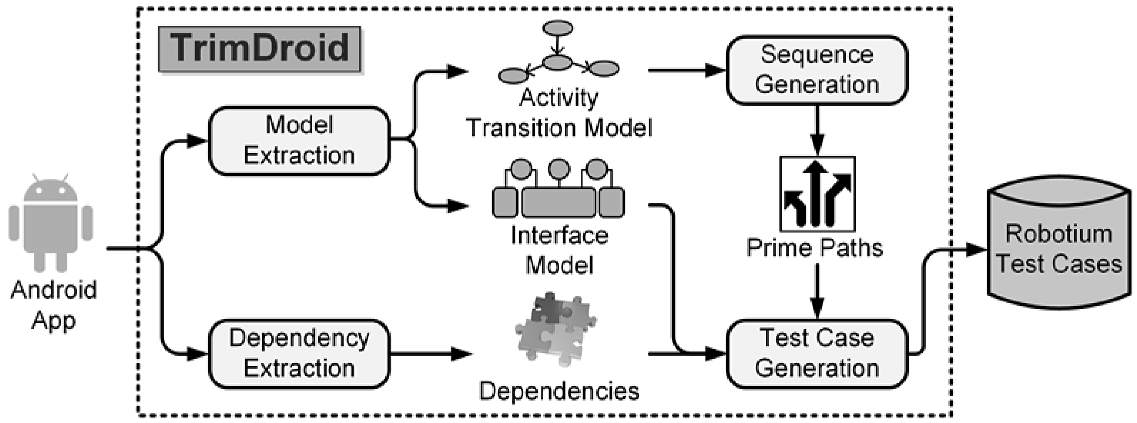


Figure 2.7: Overview of TrimDroid.

Table 2.1: Overview of existing test input generation tools for Android

App Name	Ava	Instrument		Events		Exploration	Source	Testing
-	-	Plat	App	GUI	Sys	Strategy	Code	Strategy
Monkey[35]	✓	✗	✗	✓	✗	Random	✗	Black-box
Dynodroid[33]	✓	✓	✗	✓	✓	Random	✗	Black-box
DroidFuzzer[48]	✓	✗	✗	✗	✗	Random	✗	Black-box
IntentFuzzer[49]	✓	✗	✗	✗	✗	Random	✗	White-box
Null-IF[50]	✓	✗	✗	✗	✗	Random	✗	Black-box
GUIRipper[38]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
Android Ripper[45]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
ORBIT[16]	✗	✗	✗	✓	✗	Model-based	✓	Grey-box
MobiGUITAR[23]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
M[agi]C[46]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
TrimDroid[17]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
A^3E-Depth[51]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
SwiftHand[26]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
PUMA[27]	✓	✗	✓	✓	✗	Model-based	✗	Black-box
A^3E-Targeted[51]	✗	✗	✓	✓	✗	Systematic	✗	Grey-box
EvoDroid[52]	✗	✗	✓	✓	✗	Systematic	✗	White-box
ACTEve[53]	✓	✓	✓	✓	✓	Systematic	✓	White-box
JPF-Android[54]	✓	✗	✗	✓	✗	Systematic	✓	White-box

From figure 2.7, we can see that TrimDroid is using model-based strategy for input generation. TrimDroid extracts both interface model and action model at the same time, then TrimDroid combines them together to find the prime paths in the application. Next, it traverses the prime paths to generate test case combinations. TrimDroid was released in June 2016.

From figure 2.7, we can see that TrimDroid is using a model-based strategy for input generation. TrimDroid extracts both the interface model and the action model at the same time. TrimDroid then combines them together to find the prime path in application. Next, it traverses the possible combination to generate test case combinations. TrimDroid was released in June 2016, and it is one of most updated tools in test input generation research. TrimDroid is used to generate GUI related test samples so that CamDroid can concentrate on the model extension and amelioration. In this thesis, we assume that events generated by TrimDroid are complete and non redundant.

In summary, the dependent projects are used to preform some basic functions, to enable CamDroid to concentrate on the model extension, amelioration, as well as input sequence generation.

2.3.2 Related Project

Followed by the dependent projects, we discuss related research of testing on the mobile platform to get a beneficial hint to CamDroid.

Table 2.1 shows the existing test generation tools on Android platform. We can see the model-based testing is widely applied in the Android test sample generation tools. However, most tools in the table are concentrating on the GUI-based model and neglecting to the system events.

Chapter 3

Challenges of Model-based Testing in Android System

Chapter 1 summarizes four challenging problems to be solved when introducing the component-based testing approach. The four challenges are summarized below:

- Prior model-based testing techniques do not include Service and Broadcast Receiver.
- All related works abstract application's model as solo entry finite state machine. Therefore it cannot describe the apps' behaviors properly.
- GUI-based model neglects transitions defined in Service and Broadcast Receiver. This phenomenon decreases the authentication level of generated model.
- Broadcast Receiver and Service controlled graphical user interface are neglected in the GUI-based model, this decreases the code coverage theoretically.

In chapter 3, we mainly investigate the challenges of model-based testing in the Android platform and summarize how many situations we need to deal with for better testing in the component-based model. After challenges are investigated in the Android system, we also present an illustrative application for the proposed methodology demonstration.

3.1 Challenges Classification

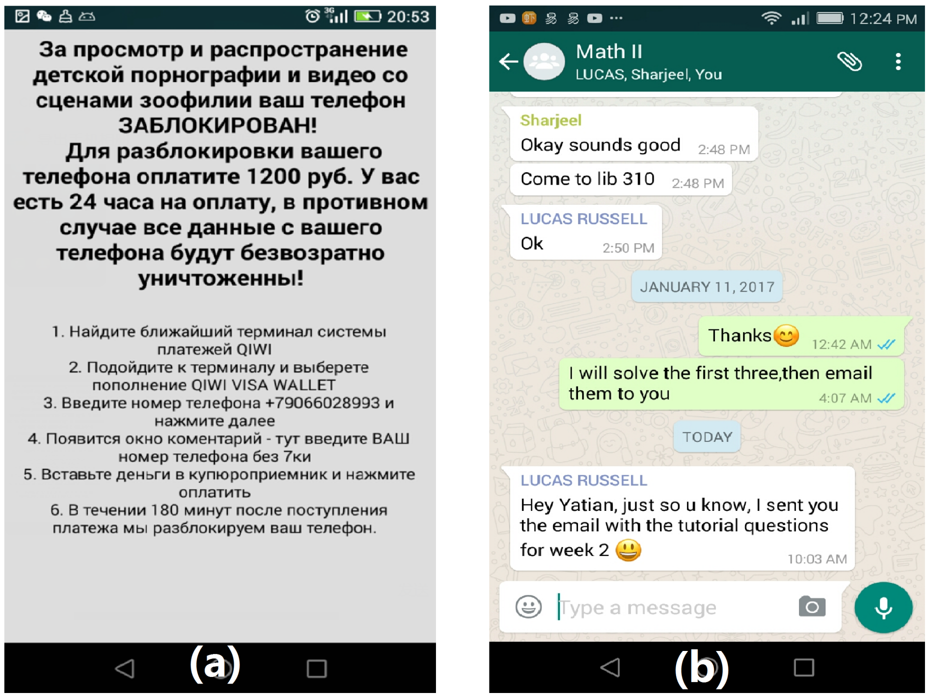
In this section, the challenges of the GUI-based model in the Android system should be clarified. These challenges decrease the effectiveness of the GUI-based model testing. It is also a source of design objectives for CamDroid.

To upgrade the GUI-based model as a component-based model, we classify the challenges into four categories. They are summarized as four specific dilemmas in model-based testing on updated Android system nowadays.

3.1.1 Program Entry & Exit Definition

Program entry & exit definition is caused by the second problem proposed in the introduction. Normally, the GUI-based model takes apps as graphical interface driven finite state machine in which the launcher activity is the only program entry. Meanwhile, the applications are assumed to exit by user's command of double back button click in GUI-based model.

In reality, the Android apps have multiple entries and exits. For a better user experience, most apps register Broadcast Receivers to monitor different kinds of events received in the system. When certain events are occurring, the app will invoke itself and show proper graphical user interface to users. That means apps may have multiple program entries other than a solo entry. As shown in figure 3.1 (b) , the conversation activity will be triggered automatically when an incoming message is received even if the user does not open Whatsup[28]. At the same time, most modern apps have the



За просмотр и распространение детской порнографии и видео со сценами зоофилии ваш телефон ЗАБЛОКИРОВАН!
Для разблокировки вашего телефона оплатите 1200 руб. У вас есть 24 часа на оплату, в противном случае все данные с вашего телефона будут безвозвратно уничтоженны!

1. Найдите ближайший терминал системы платежей QIWI
2. Подойдите к терминалу и выберете пополнение QIWI VISA WALLET
3. Введите номер телефона +79066028993 и нажмите далее
4. Появится окно коментарий - тут введите ВАШ номер телефона без 7ки
5. Вставьте деньги в купюроприемник и нажмите оплатить
6. В течении 180 минут после поступления платежа мы разблокируем ваш телефон.

Math II
LUCAS, Sharjeel, You

Sharjeel
Okay sounds good 2:48 PM
Come to lib 310 2:48 PM

LUCAS RUSSELL
Ok 2:50 PM

JANUARY 11, 2017

Thanks😊 12:42 AM
I will solve the first three,then email them to you 4:07 AM

TODAY

LUCAS RUSSELL
Hey Yatian, just so u know, I sent you the email with the tutorial questions for week 2 😃 10:03 AM

Type a message

Figure 3.1: Demonstration of auto-trigger.

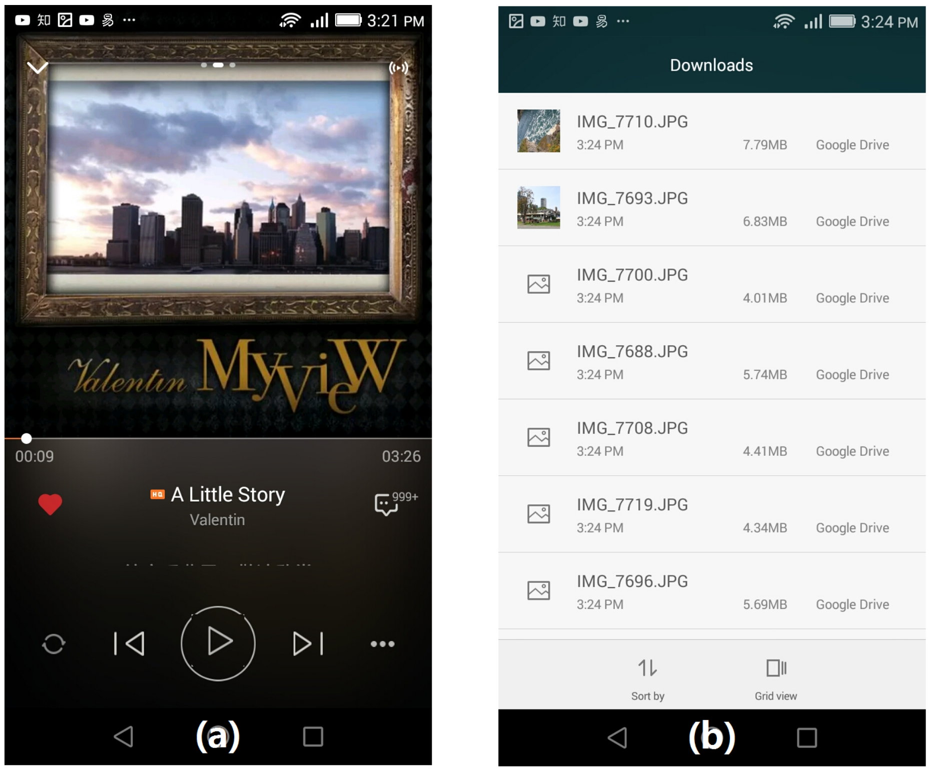
mechanism of auto-exit when the battery is low or Internet is not available. Owing to these designs, users are able to have multiple ways to interact with applications.

Alternatively, some malicious behaviors may also hide in these components' code. So we need to cover this situation in our model. Ransomwares often uses this mechanism to avoid detection and to occupy resources. As shown in figure 3.1 (a), Simple Locker monitors boot event to occupy the screen to perform blackmail to the users.

At the same time, most program exit in the GUI-based model is triggered by clicking the back button, but the apps can exit by monitoring and receiving events from outside. For example, many applications exit automatically when the battery is low or the internet connectivity is unavailable.

As shown in the demonstrations above, the app's program entry and exit is out of the description of a conventional GUI-based model. This challenge needs to be solved for a better testing effectiveness.

3.1.2 Silent Code Block & Transition Neglect


3:21 PM
Valentin MyviW
00:09
03:26
A Little Story
Valentin
999+
(a)
3:24 PM
Downloads
IMG_7710.JPG 3:24 PM 7.79MB Google Drive
IMG_7693.JPG 3:24 PM 6.83MB Google Drive
IMG_7700.JPG 3:24 PM 4.01MB Google Drive
IMG_7688.JPG 3:24 PM 5.74MB Google Drive
IMG_7708.JPG 3:24 PM 4.41MB Google Drive
IMG_7719.JPG 3:24 PM 4.34MB Google Drive
IMG_7696.JPG 3:24 PM 5.69MB Google Drive
Sort by
Grid view
(b)
Figure 3.2: Demonstration of transition trigger.

The Android development framework provides two mechanisms of component registration. They are static registration and dynamic registration. Some behaviors defined in the dynamic registered components keep silent in most situations, but it will be registered and executed when needed.

This phenomenon is usually important functionally and enables triggering transitions in GUI-based model. Figure 3.2(a) shows that the music player pauses the music automatically when the headphone is unplugged. As for figure 3.2 (b), Google Drive [29] will stop downloading if the network connectivity is unavailable. These two examples demonstrate that Broadcast Receiver can intervene in the Activity's behaviors as well as the graphical user interface display.

This mechanism increases the flexibility of Android apps functionally. Meanwhile the transitions and code blocks in Service and Broadcast Receiver are neglected in the conventional GUI-based model. This defect may cause the invalidity of test input samples and isolated state. More importantly, some hidden behaviors in silent code blocks will not be included GUI-based model and stay unnoticed in the testing stage.

In this section, we can see the traditional transition definition cannot be fully covered in the model. To overcome this problem, more programming interfaces need to be included into the model as transitions.

3.1.3 Isolated State

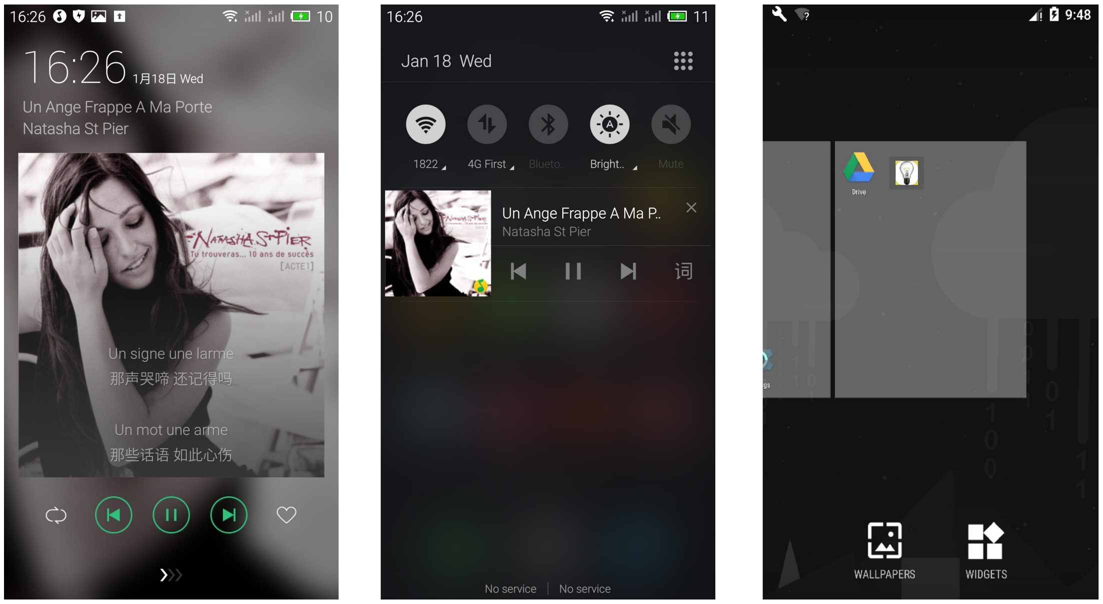


Figure 3.3: Demonstration of isolated state.

”Isolated states” is a state defined in the GUI-based model , but can not be tested by the generated test samples from the GUI-based model because transitions connected to the state are neglected.

As shown in figure 3.3 (a), lock screen is usually a typical example of an isolated state. The lock screen is an Activity which can be abstracted as state in the GUI-based model. But the state is invoked by Broadcast Receiver and can define its transition into the broadcast. Thus the state can not be tested even if test samples are generated for them.

As shown in figure 3.3 (b) the graphical user interface may be controlled by Activity. Because of its location, it also cannot be triggered even it is in GUI-based model.

Figure 3.3 (c) shows another example of isolated state. Sometimes apps register a Broadcast Receiver for a widget update. The widget is normally a graphical user interface, but it cannot be reached by the GUI-based model.

From this phenomenon, the traditional model strategy cannot even construct a fully reachable GUI-based model definitely when facing the new challenges in application's structure. This challenge will be overcome and addressed in our study.

3.1.4 Pseudo Graphical User Interface Interaction

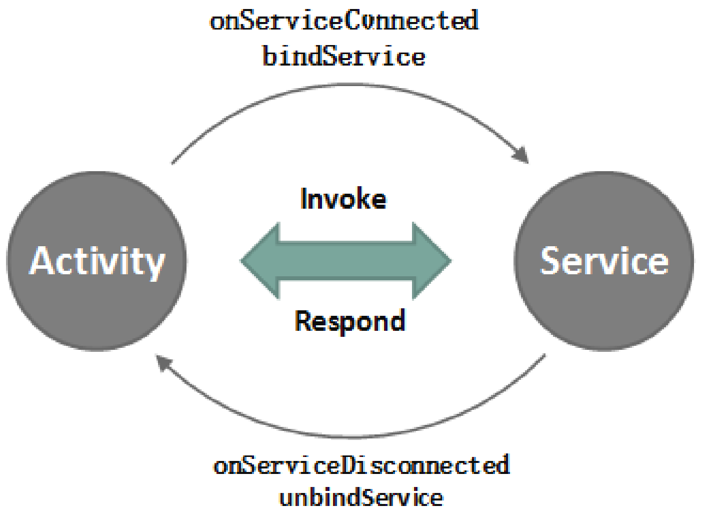


Figure 3.4: Demonstration of pseudo graphical user interface.

In most apps, developers use Service to deal with time-consuming tasks, but users always think that they are interacting with Activity. However, users cannot distinguish which component is interacted with when they interact with the graphical user interface. From GUI-based perspective, the Service handled tasks and Activity handled tasks can be seen as equivalent, but the Service controlled graphical user interface can change the graphical state, but the transition is not in GUI-based model. The Service or Broadcast Receiver controlled graphical user interfaces are defined as a pseudo graphical user interface in our study.

Figure 3.4 demonstrates the situation of pseudo graphical user interface interaction. This situation is designed to avoid ANR [36]. Some tasks need some time to be done, such as visiting a remote server or graphics processing. The apps will be terminated if the apps are slow in response. To avoid this kind of phenomenon, some time consuming tasks will be handled by a service. This design releases the burden of Activity in execution while increasing the difficulty in testing. Manual testing and the GUI-based model cannot distinguish whether the response is from Service or the graphical user interface itself.

This situation is common in Android application testing and also out of the GUI-based model's scope.

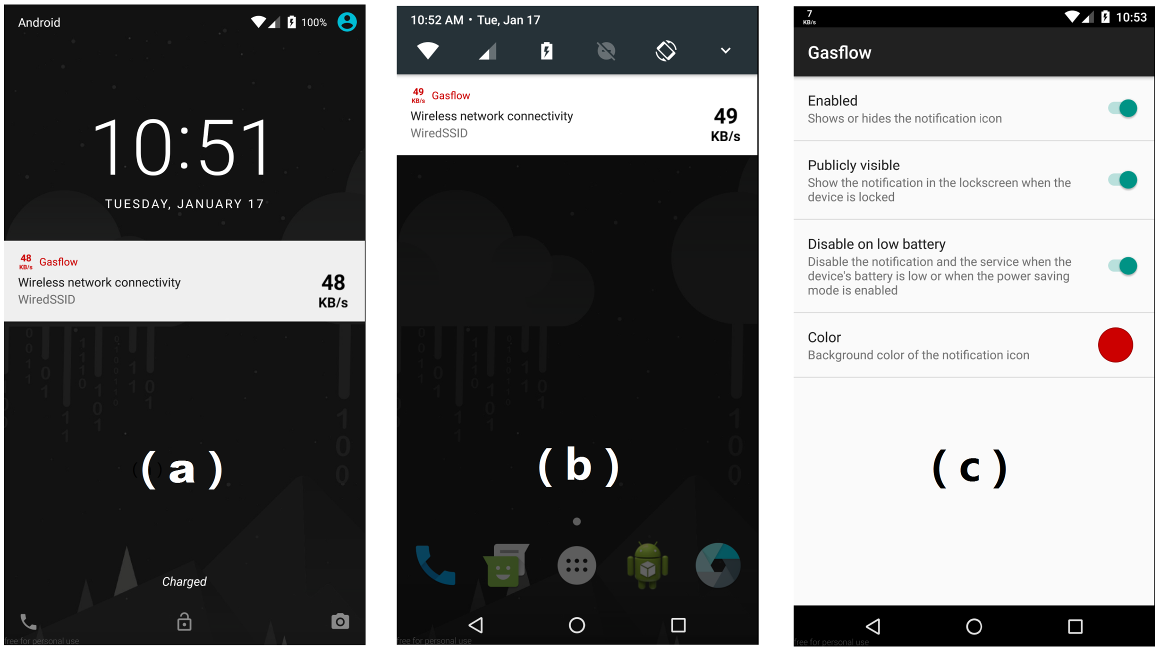
The four challenges in Android apps decrease effectiveness of the GUI-based model testing largely. Overcoming these challenges is the first step of successful model-based testing on the Android platform.

3.2 Illustrative Application

As a demonstration of the proposed method, we use an illustrative application for showing the methodology.

3.2.1 Overview of Illustrative Application

Gasflow [30] is a practical tool released in GooglePlay, used to monitor current network connectivity and speed. The network speed is shown in status bar as well as in the notification bar and lock screen. Gasflow will cancel network speed display if network connectivity is disabled. We first analyze the functionality in the app towards different components. Then, we analyze the GUI-based model of Gasflow. Finally we compare the component-based model with the GUI-based model for evaluation.



48 KB/s Gasflow
Wireless network connectivity
WiredSSID
48 KB/s

49 KB/s Gasflow
Wireless network connectivity
WiredSSID
49 KB/s

Gasflow

Enabled
Shows or hides the notification icon

Publicly visible
Show the notification in the lockscreen when the device is locked

Disable on low battery
Disable the notification and the service when the device's battery is low or when the power saving mode is enabled

Color
Background color of the notification icon

Figure 3.5: User interfaces of Gasflow. (a)Lock screen graphical user interface (b) Notification graphical user interface (c) UI-State setting activity.

Gasflow has six components in total. There are four Broadcast Receivers , one Activity and one Service. Broadcast Receivers can be divided into two groups. RecScreen is a dynamic registered broadcast receiver which monitors the actions of screen on, screen off, network connectivity status change and airplane mode change. This Broadcast Receiver will control the notification display through monitoring target actions. Battery receiver is a static registered Broadcast Receiver which monitors the battery status. When the battery is low , it terminates background Service automatically. It will start the background service when the battery status is normal. Boot receiver is responsible for monitoring device boot action. When the boot completes, Gasflow will invoke the background service automatically. Power Receiver monitors the energy plan of the device. When the device is in battery saver mode, it terminates the background Service and invokes the Service when the device quits the battery saver mode.

Traffic Service is a Service used to construct the notification and it also takes the functions as an Activity extension. Setting Activity is a user interface used to configure the functions in the app.

Figure 3.5 shows the graphical user interface of Gasflow. There are three graphical user interfaces in Gasflow while only one Activity is registered in the app.

3.2.2 GUI-based Model of Illustrative Application

Based on the description of the application, there is only one state in the GUI-based model.

Setting
Activity

Figure 3.6: GUI-based model of Gasflow.

In the GUI-based model, Setting Activity will be abstracted as a UI-state which can receive various kinds of inputs for testing. The GUI based model will abstract Gasflow as a solo graphical user interface app under test. Actually, Gasflow has more functions beyond the description of GUI-based model.

First, the factors which can affect the application's behaviors such as lock screen, network connectivity and battery status will not be described in the GUI-based model. Moreover, actions triggered by dynamic registered Broadcast Receiver and corresponding state change will not be included in the GUI-based model.

Second, traffic Service is both invoked in notification controller and Setting Activity's extender which helps to deal with tasks defined in Activity. This collaborative relationship also can not be described in the GUI-based model.

As summarized above, the GUI-based model has disadvantages in application's behaviors description. This if the application is highly relying on Services and Broadcast Receivers functionally. Specifically speaking, disadvantages in Gasflow can be stated as four problems in GUI-based model. These problems are isolated state, entry neglect, transition deficiency and state abstraction inaccuracy. Isolated state and transition & states deficiency are similar to sides of a coin. The deficiency of transition causes the isolated state. Unlike most GUI-based model, we define apps as programs with multiple entries to optimize the problem of program entry neglect.

3.2.3 Component-based Model of Illustrative Application

To overcome the problems in the GUI-based model, we introduce a novel component-based model for Android system. Next, we will use the proposed method to depict the component-based model of Gasflow. In figure 3.7, there are 5 sequences & 8 states in the component-based model which covers six different components. Comparing the GUI-based model, six components have been included into the component-based model.

From figure 3.7, we can see six components are abstracted as eight states. Battery receiver, Power receiver, Boot receiver and Screen receiver are abstracted as four BR-states. Traffic service and Setting activity are abstracted as SER-state and UI-state

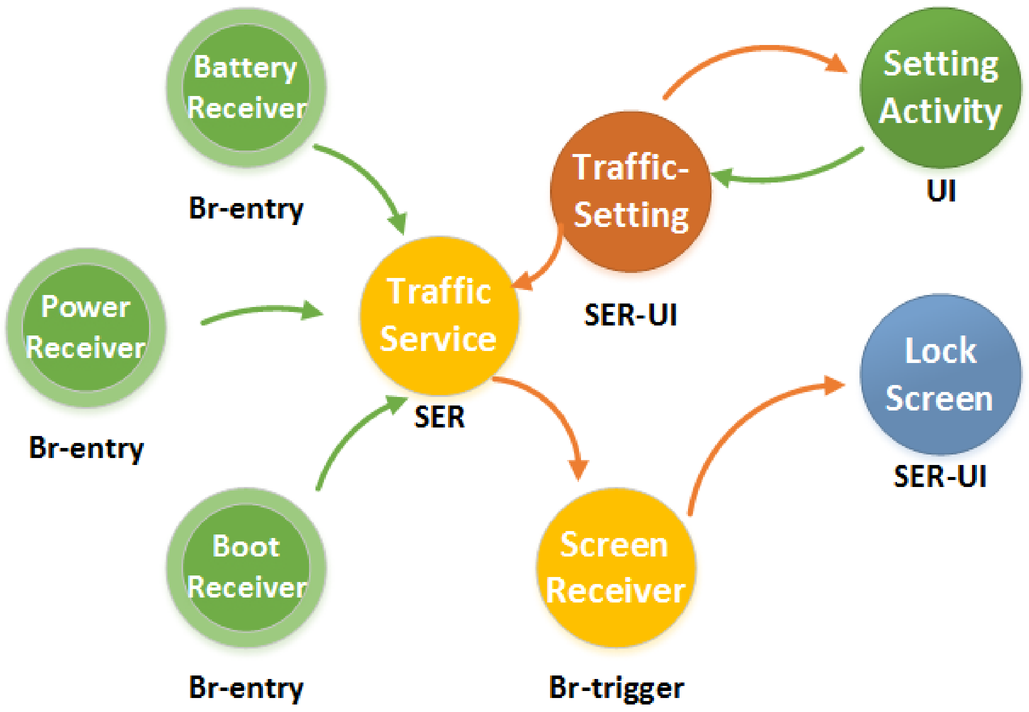


Figure 3.7: Component-based model of illustrative Android application example.

respectively. Furthermore, traffic service and setting activity are combined as SER-UI-state.

In regards to completeness, the component-based model can cover six components of the illustrative example while GUI-based model can cover only one of them. The proposed method achieves a 500% increase in component code coverage. At the same time, three more program entries have been detected by our proposed method which is an 300% increasing in number of program entry detection. Then we can conclude that the proposed model has a better coverage in testing theoretically.

As for the efficiency of the proposed solution, the component-based model increases the complexity at a reasonable cost of computational resources. Although the structure of the model expands based on the GUI-based model, It still remains feasible in model-based analysis.

Based on the description of the illustrative example, some advantage of the proposed approach has been revealed. For instance, the component-based model can describe Service and Broadcast Receiver related behaviors in apps. Furthermore, the proposed approach enables to build transitions across different components. The component-based approach can also combine UI-state,BR-state, SER-state and UI-SER-state together as component-based model for assistance of input generation to solve the problem of isolated state, transition deficiency and program entry neglect.

The illustrative example is also used to test and demonstrate the functionality of the proposed approach.

Chapter 4

Component-based Model Generation Approach

In this chapter, we will explicate the component-based model generation approach in details. First we give an overview of the proposed component-based approach & provide a clear view of our approach blueprint. Then we introduce functional modules in this approach individually.

They are model constructor, static-mapping transition builder and event sequence generator & cluster. Model constructor explains how we select the states & abstract the states. State-mapping transition builder enables to build transitions across different kinds of life cycle & components. Event sequence generator translates the gathered model data to event sequences for input sequences generation. Event cluster is designed to eliminate the redundancy in the generated sequences.

Finally the input sequences will be translated into script code to control the automated model-based testing. We will explicate our approach step by step.

4.1 Overview of Approach

Algorithm 1 depicts a high level overview of the proposed approach, which is composed of three processing modules: model constructor, static-mapping transition builder, event sequences generator & Cluster.

Algorithm 4.1 Approach Overview Algorithm

```
Require:
    The set of System Under Test, $S_i$;
Ensure:
    The set of Generated Test Input , $T_m$;
1: for $i = 0$ to $n$ do
2:     BR-state $b$ = BR-state Model Constructor($S_i$);
3:     SER-state $s$ = SER-state model Constructor($S_i$);
4:     UI-state $u$,UI-SER-state $us$ = GUI-based model Constructor($S_i$);
5:     Transition $t$ = Transition Constructor($S_i$);
6:     CBM $cbm$ = StaticMappingTransitionBuilder($b$,$s$,$u$,$us$,$t$);
7:     $T_m$ = EventSeqenceGeneratorCluster($cbm$);
8:     return $T_m$;
9: end for
```

There are two questions the approach needs to answer before extending the GUI-based model to component-based model. Normally the extension of the model will increase the complexity of states in the finite state machine presented component-based model. Can we include Service & Broadcast Receivers into the model with less increase in state's category number with better code and function coverage? Furthermore, the increased states could cause the explosive expansion of the model structurally. Can we amalgamate the Service and Broadcast Receiver related states with GUI related states with minor change in model structure? How to answer these two questions determines whether this approach can be executed functionally correctly.

As components are close related and may be executed simultaneously. If 4 components are introduced in the proposed model, $C^4{}_4+C^3{}_4+C^2{}_4+C^1{}_4=15$ states will be defined as presentation of these 4 components theoretically. From the survey towards the components, Content Provider is feeble in usage frequency and functional importance compared to Activity, Service and Broadcast Receiver. Hence the proposed component-based model are concentrating on the components except Content Provider. There still has $C^3{}_3+C^2{}_3+C^1{}_3=7$ states need to be included into the model. Owing to the execution brevity, all actions in Broadcast Receiver are accomplished instantaneously. We assume that Broadcast Receiver is abstracted as independent state and cannot overlap with other component's execution.

As a result, most behaviors can be described in apps with 4 states in the component-based model which can cover Activity, Service, Broadcast Receiver in Android Development Framework. The definition of the 4 states are listed as below

- **UI-State** is abstracted from graphical user interface layout and Activity in Android applications. This is also the main receiver for user's gesture input.

- **SER-State** is abstracted from Service. This state can be GUI controller, transition trigger and activity extension partaking to time consuming tasks. In most cases, Service related tasks are controlled by graphical user interface then abstracted as UI-SER-state below.

- **UI-SER-State** is a combination of UI-State and SER-State. This is used to represent the situations when an application is both active in foreground graphically and background computationally.

- **BR-State** is an abstraction of Broadcast Receiver. Unlike the three persistent states above, BR-State is non persistent. BR-State is commonly used as a program entry and a transition trigger. Its execution time can be neglected in the component-based model.

Besides states in finite state machine presented model, transition plays a key role in the model. GUI-based transitions are relying on the convert between graphical user interfaces. But in the component-based model, transitions can not be defined within the scope of the graphical user interface. Transitions must describe relations across different components in applications. To deal with this issue we extend the definition of transitions: Transitions are conditions which can be triggered to transit between states declared above. As a solution to extract all transitions in the model, Static-mapping Transition Builder is being proposed.

Static-mapping Transition Builder is responsible for transition data extraction to connect all states in component-based model. This proposed transition builder is implemented in three steps. First we detect all possible interfaces which can trigger transitions in the model. Second, initiative state, destination state and conditions will be identified for each transition by call graph iteration or smali code analysis. Finally the transitions will be combined into the model.

After processing by model constructor and static-mapping transition builder, component-based model is built. Event sequences will be generated based on model if following the conventional GUI-based model. However, event sequences will be redundant in volume following the traditional event sequence generation strategy in component-based model.

Event sequence generator & cluster is proposed to combine the BR-state, SER-state related inputs test with GUI-related event sequences for new testing sequence generation. In the proposed approach, input for GUI-related event, Service related event and Broadcast Receiver related events are generated separately. Then all the generated inputs are arranged into sequences in component-based model testing.

Algorithm 4.1 shows the execution process of the component-based model testing approach. First, the state constructor abstract the states from the application under

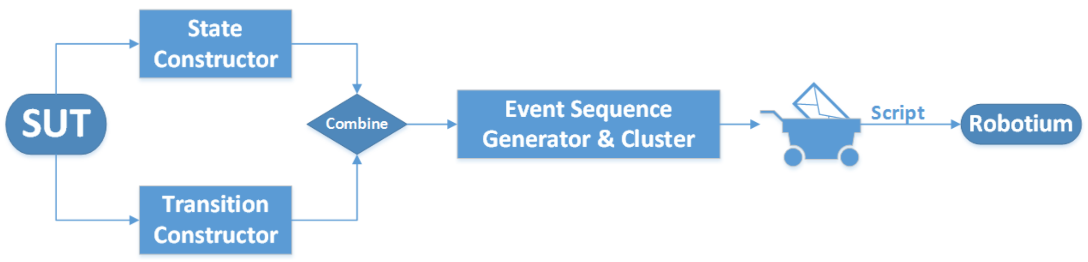


Figure 4.1: Work flow of the proposed approach.

test using static code analysis. Second, the transitions are constructed by static code analysis. Third, the sequence generator & cluster combines the states and transitions together and generates the test sample sequences which can be fed to automatic testing framework. Figure 4.1 show the work flow of the proposed approach.

This is the overview of the component-based approach for automated model generation of Android application. Then we will describe the mechanism inside the proposed approach in a more detailed way.

4.2 Model State Constructor

State abstraction and construction is pivotal in model generation. Well organized state abstraction is the first step towards a successful model generation in model-based testing. This approach uses finite state machine (FSM) as presentation of component-based model which is same as previous researches. Characteristically, there are four states included into the model covering behaviors in graphical layout, Activity, Service and Broadcast Receiver.

As mentioned above, the model needs to overcome problems of description redundancy and structure complexity. To solve these problems, the proposed approach is concentrating on the most influential functions of components into the model, then abstracting them separately.

This section will describe the component-based approach model abstraction method as following steps. First, we introduce the abstraction of proposed states. Then we investigate and concentrate on the most influential behaviors in applications of components. Finally we use our illustrative application to show the proposed model as a demonstration. This thesis focuses on the extension of model and concentrates on the BR-state, SER-state and UI-SER-state.

Figure 4.2 shows the work flow of model's state constructor. The work flow can be divided into three concurrent processes. For the first process, ClassyShark extracts data in Android Manifest file and arrange the state's meta data as a reference. The meta data contains the static registered components in Android application. For the second process, apktool reverse engineers the SUT and outputs the smali code for static code analysis. Meanwhile we use soot to locate the critical program interface and extract the transitions in the SUT. For the third process, we also uses ClassyShark to get the graphical user interface. The graphical user interfaces can be classified as Activity related graphical user interfaces and Service related graphical user interfaces. Then Activity related graphical user interface data and Service related graphical user interface data are generated as output.

After having all extracted data, we can use meta data as a blueprint to generate all static registered states' code data. Then we can generate the dynamic registered states' code data by static code analysis. Then we related the code data with the graphical user interface data together. They will be constructed as final generated states in the component-based model. This is the work flow of the model constructor. Then we will explicate the state & transition abstraction structure.

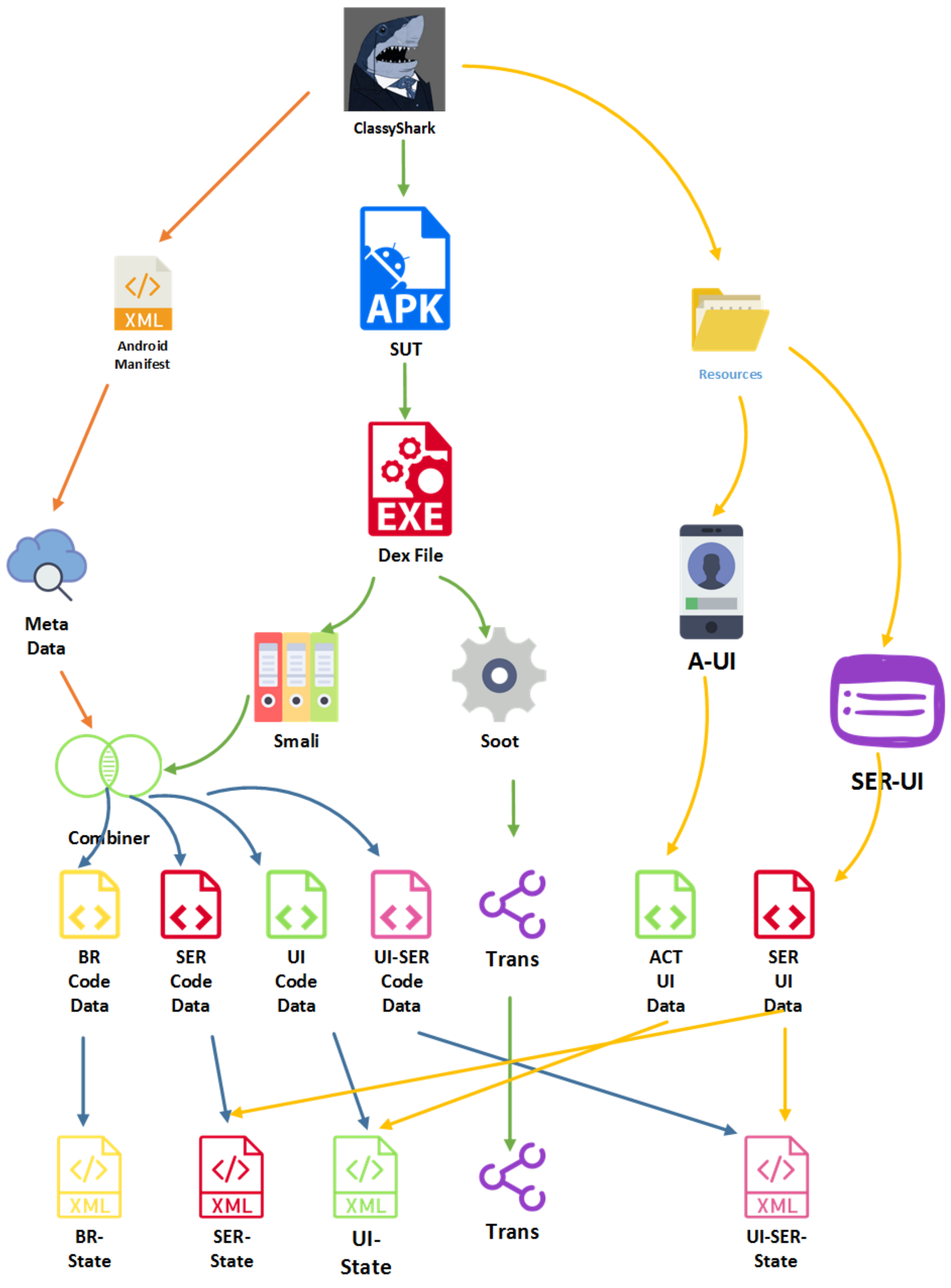


Figure 4.2: Work flow of model constructor.

4.2.1 State & Transition Abstraction

This section introduces abstraction of states and transitions in the component-based model.

BR-state is an abstracted state of Broadcast Receiver. Thus, property of BR-state is relying on the life cycle of the Broadcast Receiver. In our model, BR-state is defined as a entry state or transition trigger. Its life cycle begins when a broadcast invokes the component and ends when all commands in the onReceive function have been executed. There is a 10 seconds timeout limitation, so we define it as an instantaneous state and agree on that BR-state does not overlap with UI-state, SER-state and UI-SER-State. To construct this kind of state we extract the declarations and targeted trigger event from the AndroidManifest.xml file then detect out directed transitions by static code analysis. As for dynamic registered Broadcast Receiver, the registration programming interfaces will be scanned across all the source code of decompiled Android applications instead of extracting from AndroidManifest.xml file.

BR-State model provides information about both system or custom broadcast triggered behaviors. Normally there is no visible user interfaces in BR-State model. At same time BR-State is mainly used as a entry to receive the system events and transition triggers in application. Its property enables BR-state to be a controller over different states. In proposed approach, the model represents the BR-State as a tuple (B,E,F,T)

- B is a finite, non-empty set of Broadcast Receiver registered in a application(State Name).
- E is a finite, non-empty set of system or custom events which can trigger the Broadcast Receiver(Transition Trigger,e.g sendbroadcast).
- F is a finite set of state names whose registered actions can invoke this state

(Source state, actions).

- T is a finite set of state names can be invoked by the BR-state (Destination State).

UI-state is an abstraction of graphical user interface in an Android application. UI-state always shows up automatically and receives user′s gestures or value inputs after invocation of the launcher Activity. In the proposed approach, methodology of [16] has been applied to assist the UI-state modeling. Finally the approach uses static-mapping transition builder to collate the UI-state with BR-state,SER-state and UI-SER-state together.

SER-State is an abstraction of Service. SER-state is usually used as a graphical user interface controller, activity task extension, transition trigger. SER-state is widely used especially in applications such as music player, file storage applications such as GooglePlayMusic and DropBox. Although the life cycle of Service is relatively simpler than Activity, it is also diverse enough to accomplish different tasks functionally. We define the entry of a SER-state when the onCommandStart function is invoked and the state is ended when onDestory function is invoked. SER-state is similar to UI-state logically at some level and also execute in a more persistent way.

UI-SER-State is motivated from the situation where Activity and Service can coexist when an app is being executed. First, Service can be a graphical interface controller through notification implementation. Second, Service can partake some time-consuming tasks of Activity controlled graphical user interface through service bind method. Sometimes Service also interacts with activity through messages.

UI-State, SER-State and UI-SER-State are all persistent states and may have a visible user interface which enable users to interact with gesture input. For these three kinds of states we describe them as tuple (S,L,G,I).

- S is a finite, non-empty set of states including UI-state, SER-state, UI-SER-state(State name).
- L is a set of registered event listeners and handlers in components (With destination state for static mapping).
- G is a set of GUI elements.
- I is a set of possible inputs for each state.

The proposed definition of state abstraction enlarges the description coverage comparing to the GUI-based model. Differences between the GUI-based model and the component-based model can be found in table 4.1.

Table 4.1: Comparison between GUI-based model & component-based model

-	GUI-based Model	Component-based Model
Component Cover	Activity	Activity,Service,Broadcast Receiver
Initial State	Launcher Activity	Launcher Activity & System Events
State in Model	UI-state	UI,SER,BR,UI-SER-state
Transition	between GUI	between proposed states
Transition's Condition	startActivity	startService,stopService...

After describing states in the component-based model, we discuss the transition definitions in the model. The transitions are represented as tuples (E,A,D,F),where

- E is a finite, non-empty set of state including UI-State, BR-State or UI-SER-State which can be triggered to another state (Source State).
- A is the Actions can trigger the transitions in E (Transition Condition).
- D is the set of destination states (Destination State).
- F is the set of final states.

Figure 3.7 shows the component-based model of the illustrative example. Transitions are usually triggered by certain methods such as StartActivity, StartService and StartActivityForResult. Proposed approach iterates the call graphs of states and detects all possible triggers in app′s code to build the transitions including inter-component and inter-component transitions.

Then, the proposed approach collates the extracted data into the set defined above and prepared for processes in static-mapping transition builder.

4.2.2 Functional Behavior Analysis of States

The description abstraction level can affect the complexity of states in the model. In the proposed approach, only the most important functions of each components will be introduced into state for better testing efficiency. In this section, we will discuss the functions introduced into the component-based model by targeting dilemmas proposed in the introduction.

(1)Functional Behavior Patterns of Broadcast Receiver

Two functions of Broadcast Receiver are introduced into the BR-state in the component-based model.

Program Entry & Exit

Conventionally defined, a program entry is a user interface or signal which can invoke apps and lead them to other states. Normally, researchers define the launcher Activity as solo program entry of an Android application. It is true when most applications are triggered and interacting with users' input through a launcher activity in last few years. There is a trend that more and more applications are responsive for different system events. Therefore Broadcast Receiver can also play a role as program entry and invoke the app by system events.

Broadcast Receiver can be divided into two classifications by functions. They are the system predefined Broadcast Receiver and developer custom Broadcast Receiver. Most program entry tasks are handled by the system event Broadcast Receiver.

Some system events can change the states inside the Android system as well as trigger the Broadcast Receiver registered in the application. Although the Broadcast Receiver's life cycle is not persistent, it can play a role of the program entry. Broadcast Receiver can redirect to any states in the application. The predefined trigger in the Android application may contain some hidden malicious behaviors such as logic bomb [24]. It is meaningful to include these functions into the component-based model's state.

Table 4.2: Commonly used system events & Broadcast Receivers

System Event	Triggered Event Condition
Boot Completed	When device restarts
Package Removed	When app's package is removed
Package Replaced	When app's package is replaced
Connectivity Change	When network status changes
Battery Changed	When battery status changes
New Outgoing Call	When a call comes
SMS Received	When receives a message
Widget Update	When widget's status updates
Media Mounted	When media is mounted
Media Unmounted	When media is unmounted
Media Removed	When media is removed
Time Changed	When time is changed
Time Zone Changed	When time zone is changed

As shown in table 4.2, Broadcast Receiver related system events are commonly used in Android applications. They are including the events of boot, phone call, data connection, storage card as well as time zone changes. To fit them into the model, CamDroid abstracts them into a state which is monitored by the control flow. This function mainly covers the system pre-defined Broadcast Receiver. Then we discuss the role of custom defined broadcast in Android applications.

Transition Trigger

The developer can also customize their own broadcast and Broadcast Receivers to create transitions between different components. These transitions are easy to omit if the model only includes the graphical user interfaces.

In other words, transitions may hide into the Broadcast Receiver to avoid the scan of model construction to hide its behaviors. These kind of Broadcast Receivers are normally defined by developers and triggered by a custom counterpart broadcast. This kind of transition is contained in the Broadcast Receiver and invisible if the model only includes the graphical user interfaces.

More importantly, this function can be implemented to cause the silent code problem by dynamic registration. Some hidden behaviors are usually implemented under this function. Isolated state in chapter 3 is a typical example caused by neglect of Broadcast Receiver in the GUI-based model.

In this section, we analyze important behavioral patterns of Broadcast Receiver. These behaviors are important and they are out the description of the GUI-based model.

(2).Functional Behavioral Patterns of Service

Basically Service's functional behavior patterns can be classified into three categories. They are the graphical user interface controller, transition trigger and activity tasks extension. These three functions can cover most application situations. As explained above, Service relates to two different states: SER-state and SER-UI-state. Actually, Service is usually controlled by Activity functionally. So this thesis will introduce Service as trigger firstly. Then we will discuss the Service as graphical user interface and activity task extension.

Transition Trigger

As discussed last section, SER-state may be a transition trigger in Android applications. Unlike Broadcast Receiver as transition trigger which can only process some short term tasks in time scale, Service as transition trigger can run long persistent process. Then direct to destination state after task's completeness. Comparing to Service in life cycle complexity, Broadcast Receiver can only register the transition into the broadcast handler method. On the contrary, the Service can register the transition on any part its life cycle. For example, if a net disk such as Google Drive is downloading and opening a pdf file, the downloading task need to be handled background and the application directs to pdf reading user interface when the downloading task has been completed.

That is also the differences of transition trigger between Service and Broadcast Receiver. CamDroid needs to monitor all the behaviors in Service so as to get all the possible transition triggers. At the same time, a Service can handle one or more transitions at once. In this way, Service can be a solo transition trigger or a transition scheduler at the same time.

More importantly, Service can be triggered at any component in an Android application. After running the tasks in Service, the state will be conducted to another state.

Switching and scheduling tasks and states in application is a commonly used function in Android. In some other situations, Service is used to control the graphical user interface.

Graphical User Interface Controller

In the real life, there is an increasing demand for users to get control of the background tasks. For example, music player give users a control panel to control the background music playing. Timer application also gives control panel when timer is running background. To meet this demand, the graphical user interface is controlled by Service. It is defined as Service controlled graphical user interface.

A standard process of Service visualization can be stated as following. First, the application starts a background task then transfers the control priority to the Service component. Normally, the Service instances the user interfaces in notification bar of Android system while the traditional graphical user interface is no longer displaying in the foreground. In simple words, the user interface is controlled by Service and the application is in the SER-state in this stage. Service also handles the interactions from the Service controlled graphical user interfaces. The interactions also can be divided into two classifications. They are divided by whether triggering the transition conditions or not.

More importantly, Service controlled graphical interface is short in control flow. So we apply the coverage-first strategy for Service controlled graphical user interface. Actions as transitions will be triggered at last. This will be implemented in Event Sequence Generator & Cluster.

In summary, we can conclude that Activity is the controller for foreground process graphical user interface while Service is the controller for background interface process interface. This graphical user interface data is not scanned in the traditional GUI-based model. As a result, some GUI data in SER-state may be his kind of situation.

Activity Task Extension

In some situations, Activity keeps communicating with one or more Services. Meanwhile, Service partakes some time-consuming tasks for activity. Service acts like an extension of the Activity.

Activity binds with Service through the interface binder in Android. Then the Service can be controlled by Activity in some level through service connection interface. To achieve this goal, the proposed approach needs to monitor the interfaces such as onServiceDisconnected, onServiceConnected ,bindService, unbindService of which process will be explicated in next section.

Although this mechanism helps in software robustness, it may cause many defects in testing as the testers cannot distinguish where the response is coming from.

4.2.3 Modeling Strategy

This section introduces the modeling strategy of Service and Broadcast Receiver in CamDroid.

BroadcastReceiver Modeling

Following the criteria above, the modeling implementation process has the format of tuple (B,E,F,T). The Broadcast Receiver is classified into two categories. The Broadcast Receiver is responsible for changing states and variable values in Application if both F and T equals to null. The Broadcast Receiver is designed as a program entry if F equals null when T does not equal to null. The Broadcast Receiver is a transition trigger if neither F or T is null. The modeling process is shown in algorithm below.

From algorithm 4.2, we can see the process of modeling the Broadcast Receiver. In first step, the proposed approach collects all Broadcast Receivers to put them into

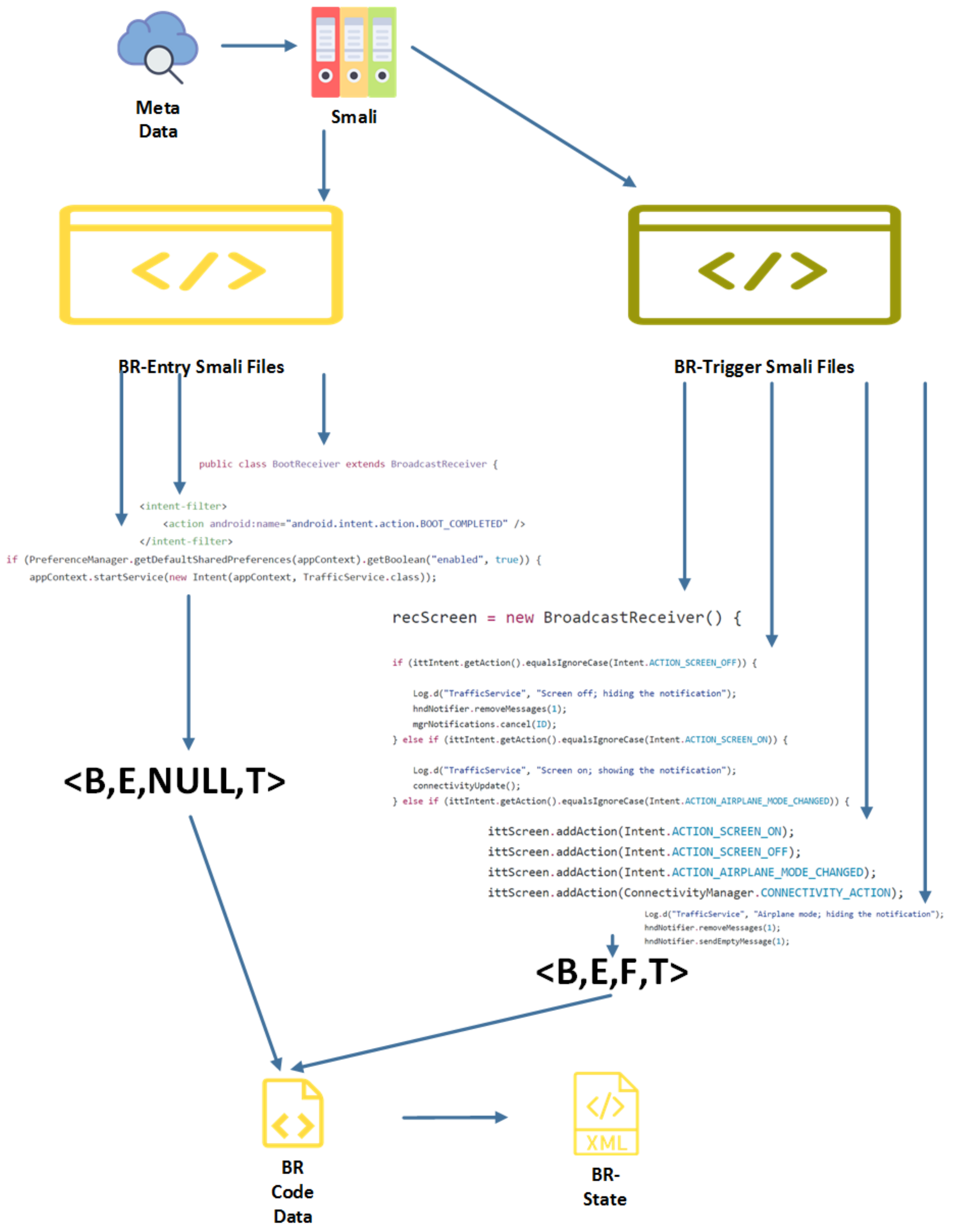
Algorithm 4.2 BR-state Modeling Algorithm

```
Require:
    The set of BroadcastReceiver, $Br_i$;
Ensure:
    The set of Generated BR-state , $BR_m$;
 1: for $i = 0$ to $n$ do
 2:     if $Br_i$ is program entry then
 3:         Event e = ExtractEvent($Br_i$);
 4:         Transition t = ExtractTransition($Br_i$);
 5:     else {$Br_i$ is transition trigger}
 6:         State s = SenderDetect($Br_i$);
 7:         Transition t = ExtractTransition($Br_i$);
 8:     end if
 9: end for
10: return  $BR_m$;
```

the set B. Then CamDroid classifies the elements in set B. If the Broadcast Receiver is a transition trigger or program entry, CamDroid will continues to process, otherwise it will be terminated.

In the second step, if the Broadcast Receiver is a program entry, the proposed approach will extract the call graph of the Broadcast Receiver then detect the next state while assigning it to set T. CamDroid also get the trigger action in static code to assign it to set E.

If the Broadcast Receiver is a transition trigger. It will first extract the call graph of the Broadcast Receiver's handler method. Then it detects the possible triggers and assign them to T. Then CamDroid applies the static-mapping transition builder to the targeted Broadcast Receiver to connect it with the source components to restore the full transition, then assigns the set F and T. In this kind of situation, the event in this Broadcast Receiver is the event which can trigger the invocation of the Broadcast Receiver. Figure 4.3 shows the work flow of BR-state modeling based on the illustrative example.



```
public class BootReceiver extends BroadcastReceiver {

<intent-filter>
    <action android:name="android.intent.action.BOOT_COMPLETED" />
</intent-filter>
if (PreferenceManager.getDefaultSharedPreferences(appContext).getBoolean("enabled", true)) {
    appContext.startService(new Intent(appContext, TrafficService.class));
```

```
recScreen = new BroadcastReceiver() {

if (ittIntent.getAction().equalsIgnoreCase(Intent.ACTION_SCREEN_OFF)) {

    Log.d("TrafficService", "Screen off; hiding the notification");
    hndNotifier.removeMessages(1);
    mgrNotifications.cancel(ID);
} else if (ittIntent.getAction().equalsIgnoreCase(Intent.ACTION_SCREEN_ON)) {

    Log.d("TrafficService", "Screen on; showing the notification");
    connectivityUpdate();
} else if (ittIntent.getAction().equalsIgnoreCase(Intent.ACTION_AIRPLANE_MODE_CHANGED)) {
```

```
ittScreen.addAction(Intent.ACTION_SCREEN_ON);
ittScreen.addAction(Intent.ACTION_SCREEN_OFF);
ittScreen.addAction(Intent.ACTION_AIRPLANE_MODE_CHANGED);
ittScreen.addAction(ConnectivityManager.CONNECTIVITY_ACTION);
```

```
Log.d("TrafficService", "Airplane mode; hiding the notification");
hndNotifier.removeMessages(1);
hndNotifier.sendEmptyMessage(1);
```

Figure 4.3: Work flow of Broadcast Receiver modeling.

Service Modeling

The Service Modeling is a model construction process of SER-state and UI-SER-state. From the discussion above, Service as transition trigger and Service controlled graphical user interface are SER-state. In the same time, the Service as extended Activity is a UI-SER-state. Based on the differences in the structure of various functions, we use different methods to formalize the component for fitting them into the model.

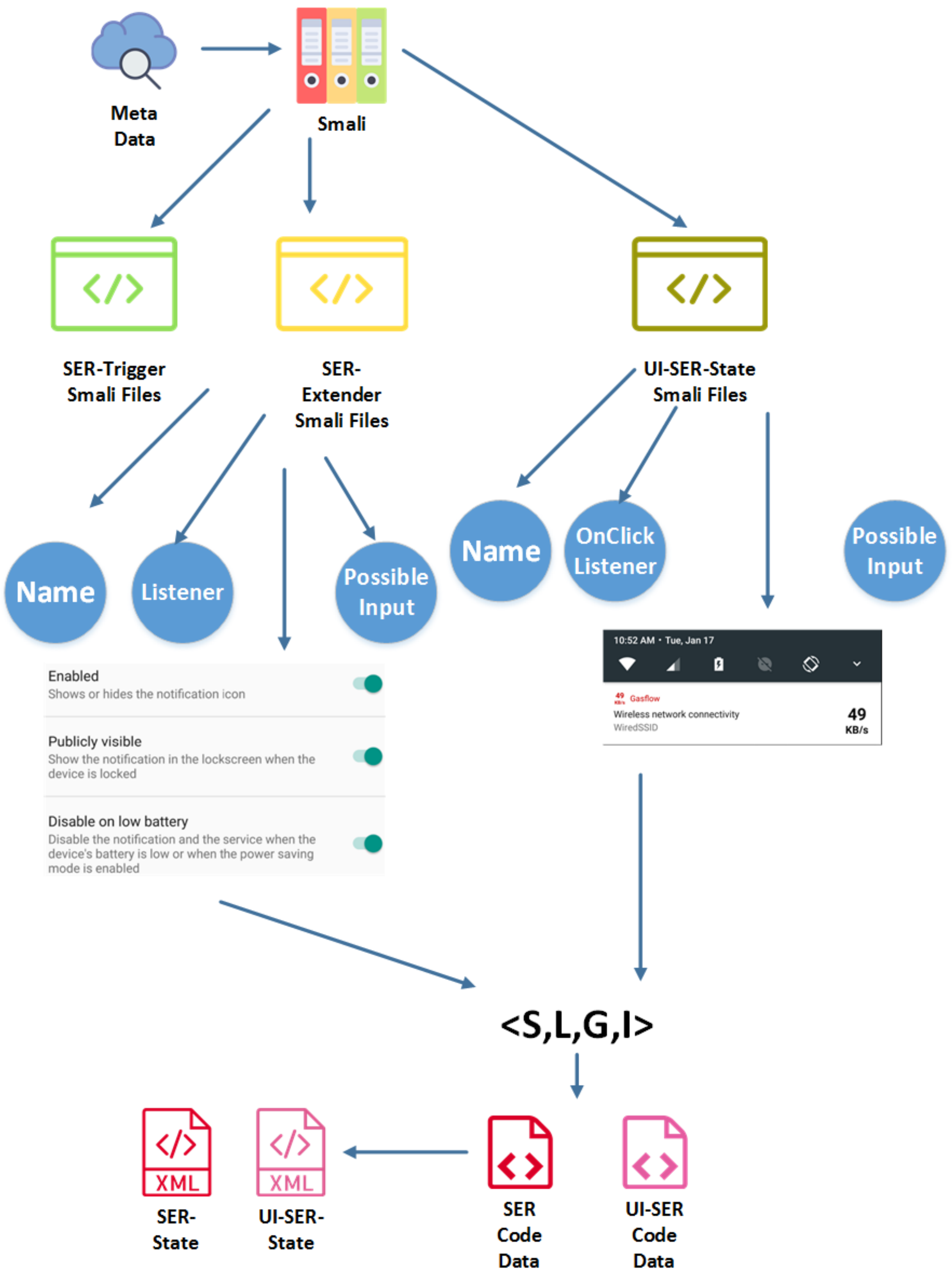
As shown in Algorithm 3, we can see that the algorithm is divided by three conditional statements. As the structure of SER-state is (S,L,G,I), the algorithm is designed to fill data into each state. If the Service is a transition trigger, the related graphical user interface element is put into G. The related input event is put to the set I. Set L is used to store the transition destinations. If the Service is a Service controlled graphical user interface, the registered listeners are assigned to the set L while all the graphical user elements are assigned to set G. Finally, the possible input events are assigned to the set I. If the Service is an Activity extended Activity task handler, G is a set of graphical user element related to the extended Service while L is the transition destinations.

Figure 4.4 shows the work flow of the Service modeling. After gathering the data needed for Service modeling, we need to distinguish the functional behaviors proposed before. If the Service is used as transition trigger, the modeling work flow is almost same as BR-state. If the Service is GUI controller or Activity extender, the Service will be modeled as shown in figure 4.4.

From the description above, the approach is able to construct the state including UI-state,SER-state,BR-state,UI-SER-state. In the model construction we use the static mapping strategy in the algorithm when data connects the components. In the next section we will discuss static mapping transition builder in a more detailed way.

Algorithm 4.3 SER-state Modeling Algorithm

```
Require:
    The set of Services, $Ser_i$,$UI_i$;
Ensure:
    The set of Generated SER-state , $SER_m$;
 1: for $i = 0$ to $n$ do
 2:   if $Ser_i$ is GUI Controller then
 3:     Widget w = ExtractGUI($Ser_i$);
 4:     Event e = ExtractEvent($Ser_i$);
 5:     Transition t = ExtractTransition($Ser_i$);
 6:     State $SER_m$ = StateCombiner(e,w,t);
 7:   else {$Ser_i$ is transition trigger}
 8:     State s = SenderDetect($Ser_i$);
 9:     Transition t = ExtractTransition($Ser_i$);
10:   else {$Ser_i$ is GUI Task Extender}
11:     Widget w = IncludeGUI($UI_i$);
12:     Event e = ExtractEvent($Ser_i$);
13:     State $SER_m$ = StateCombiner(e,w);
14:   end if
15: end for
16: return $SER_m$;
```



Enabled
Shows or hides the notification icon

Publicly visible
Show the notification in the lockscreen when the device is locked

Disable on low battery
Disable the notification and the service when the device's battery is low or when the power saving mode is enabled

Figure 4.4: Work flow of Service modeling.

4.3 Static-mapping Transition Builder

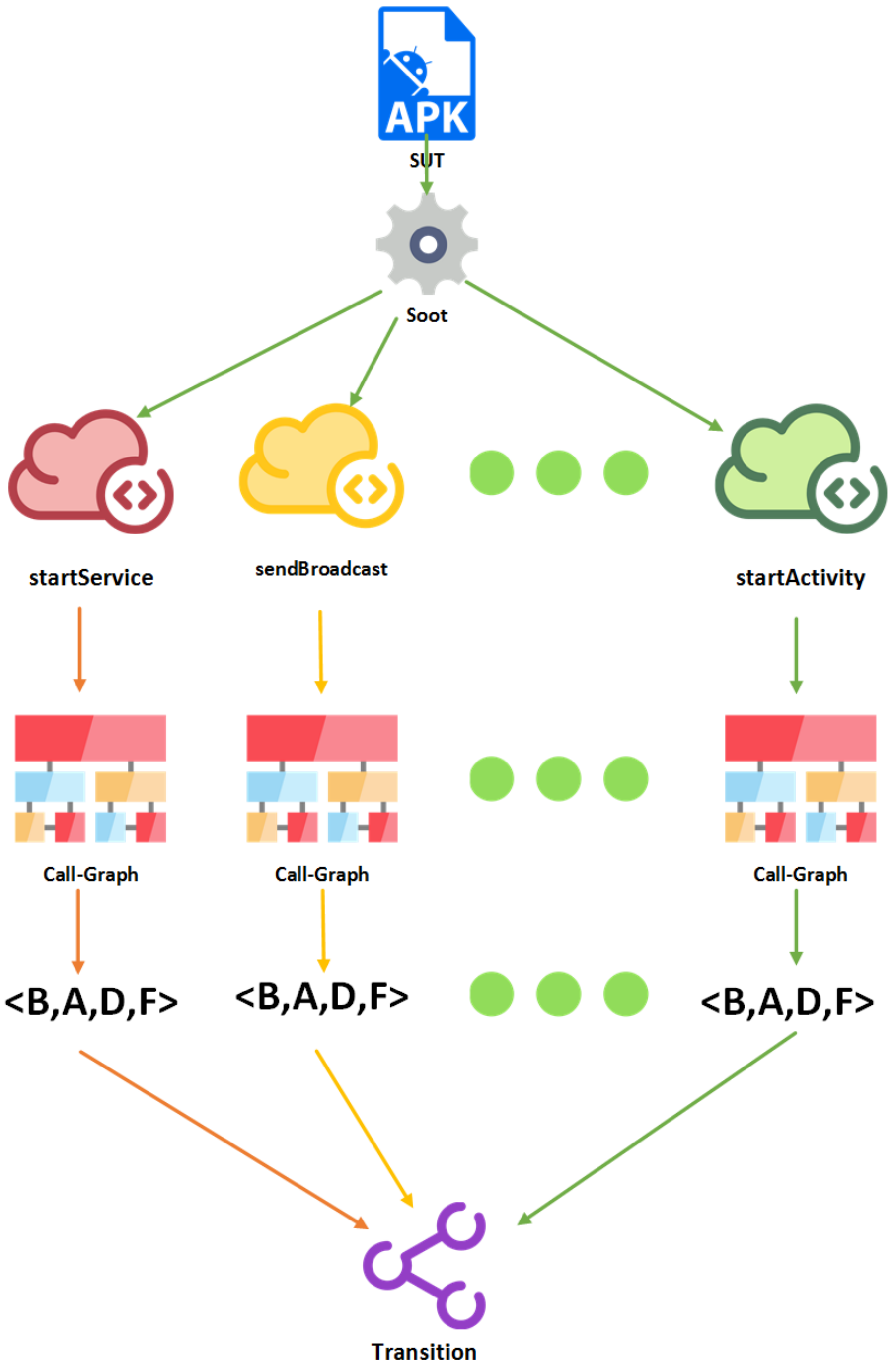


Figure 4.5: Work flow of static-mapping transition builder.

In Android application analysis, control flow graph such as call graph is often used to describe the connections between interfaces. As our model is containing components crossing Activity, Service and Broadcast Receiver, there are two difficult points in the process of building complete relations in the Android applications cross multiple components. The first difficult point is the complexity of the transition triggers. More methods can be used as transition triggers in the model-based testing. The second difficult point is that some transitions are custom defined by developers which is hard to be found by conventional GUI-based model. To solve this problem, Static mapping strategy is raised.

There are two steps to implement the static mapping transition builder. The first step is interface selection. This step is used to narrow down the range of monitors in the application to decrease complexity. Then the second step is transition construction used to construct the transitions in the component-based model.

Figure 4.5 shows the work flow of the static mapping transition builder. At first, soot will locate all the selected transition triggers defined in the component-based model. Then we extract the call-graph of the program interfaces and get the trigger's possible events and destination states. Finally we can have the data as tuple (B,A,D,F). The tuples are constructed as transitions in the component-based model.

4.3.1 Interface Selection

The selected interface are all related to data or control flow across different components. The table below shows some commonly used interfaces related to cross component invocation.

Table 4.3 shows the commonly monitored interfaces in the Static Mapping transition builder. These interfaces are all related to transition switches and triggers.

Table 4.3: Interface monitored by static-mapping strategy

Interface Name	Interface Usage
SendBoradcast	Send Broadcast
onReceive	Receive the broadcast
StartService	Invocation of a Service
onBind	Bind a service with an activity
onCreate	Create a service
onStartCommand	Begin execution of a certain
onDestory	Destroy a Service
stopService	Stop a service
onServiceConnected	Connect a Service
onServiceDisconnected	Disconnect a Service

4.3.2 Transition Construction

After getting all required data of intra-component transitions, the proposed approach will construct the extracted data into transitions to fit into the model.The process of the transition construction is shown in Algorithm 4.

Algorithm 4.4 Transition Extraction Algorithm

```
Require:
    The set of component, $Ser_i$,$UI_i$,$BR_i$;
Ensure:
    The set of Generated Transitions , $TRAN_m$;
1: for $i = 0$ to $n$ do
2:     TranInter t = TranInterDector($Ser_i$,$UI_i$,$BR_i$);
3:     Event e = TriggerEvent(t);
4:     Transition TRAN = StartEndState(t,e);
5: end for
6: return $TRAN_m$;
```

From the algorithm 4, CamDroid extracts the monitored program interfaces and event triggers from the static code, then figures out the relations of the interfaces across components to build the full control flow of the components. From here we can build the cross component call graph to describe the model in a more detailed method.

4.4 Sequence Generation & Event Cluster

4.4.1 Sequence Generation

Normally input generation automation tools investigate possible paths in the finite state machine represented model of applications, then the tools generate tests inputs sequences based on the prime paths. Different members in a sequence is a possible input for a certain state.

Figure 4.6 shows the work flow of proposed sequence generation method. After processed by the state constructor and static mapping transition builder, we can have four sets of generated states and one set of generated of transition data. Then we will classify the states into three categories. BR-Entry states and the launcher activity will be classified as entry state. The states which do not have any transitions to other states will be classified as final state. The state has largest depth will become the finial state if each state is connected by transitions. Others will be classified as non-entry states.

Then we will use transitions to bridge the states in the component-based model. Then we will use the illustrative example to demonstrate the sequence strategy in component-based model.

As discussed in chapter 3 , there are eight states in Gasflow totally. At the first step, we classify the states in Gasflow into three categories. Battery Broadcast Receiver, Power Broadcast Receiver, Boot Broadcast Receiver and Setting Activity are classified as entry state. Traffic Service and Screen Broadcast Receiver are classified as non-entry states. Lock Screen is classified as finial state. The classification result is shown in figure 4.7 (a). At the second step, the transitions whose source state is the entry state will be included into the model to connect the entry state and non-entry state. At the third step, transitions whose source state is traffic service are included

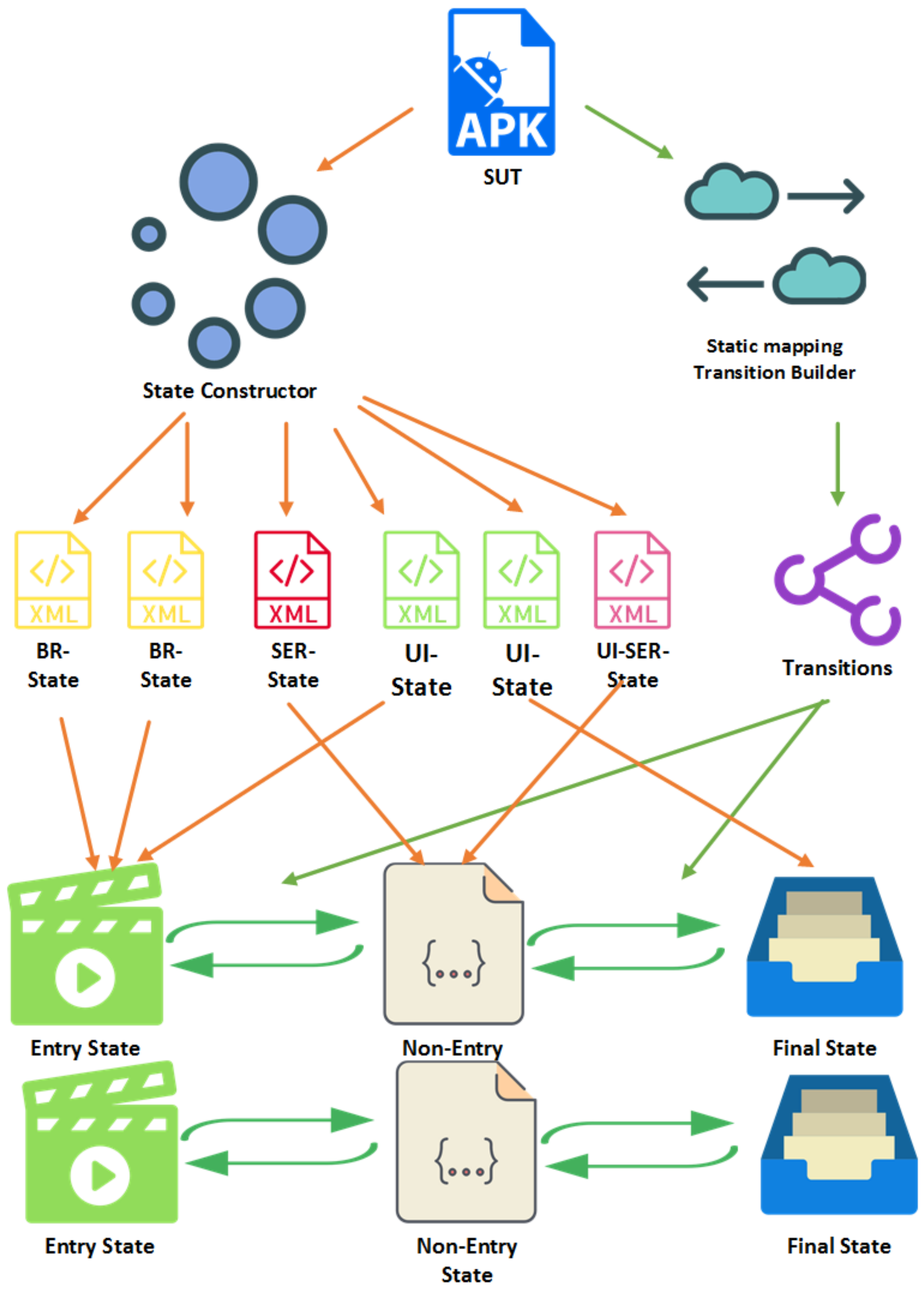


Figure 4.6: Work flow of sequence generator.

into the model as shown in figure 4.7 (c). Finally, transitions derivate from the final state are included for a complete component-based model.

Although the sequence generation method can generate the sequences successfully.

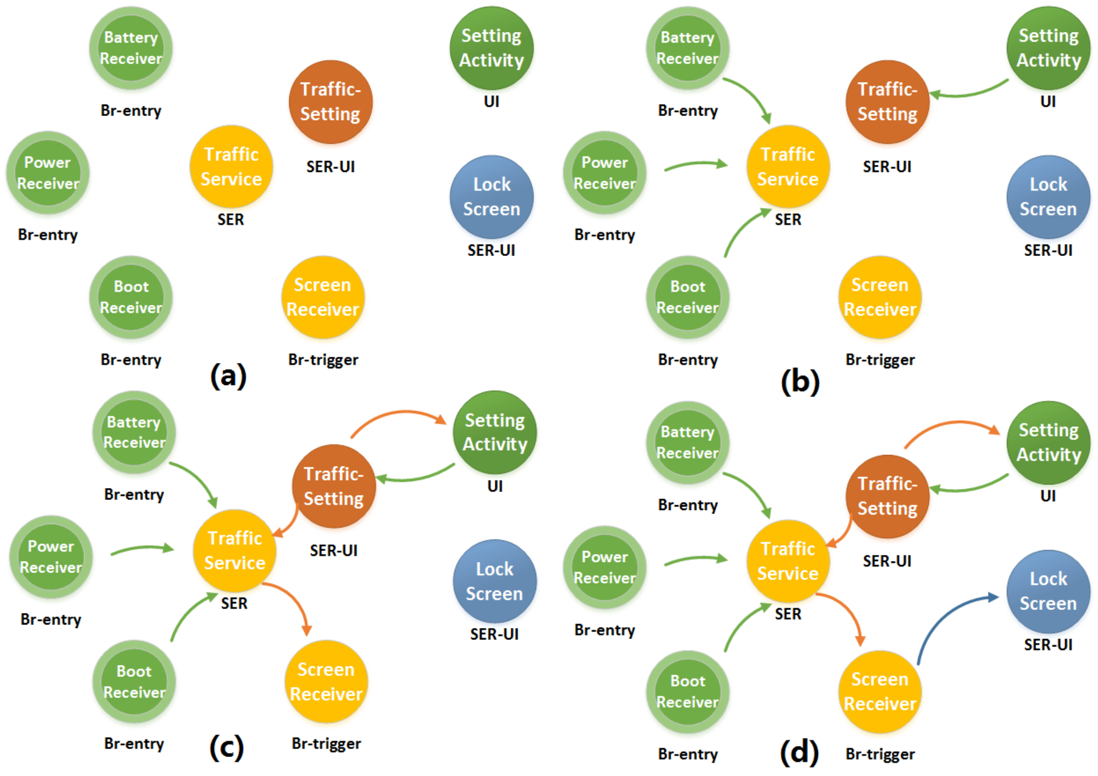


Figure 4.7: Notification graphical user interface interaction.

There is a problem in the test event sequence generation in component-based model. It is the state complexity in the notification or lock screen graphical user interface. In most notification and lock screen graphical user interface interaction, most GUI elements are not controlled by event listeners registered in program. They are normally sending a broadcast to a Service or Activity to consign the tasks to others as shown in figure 4.8.

However, this process is constituted by two states and transitions. This phenomenon lengthens the sequence of testing samples. It will decrease the efficiency of component-based model testing.

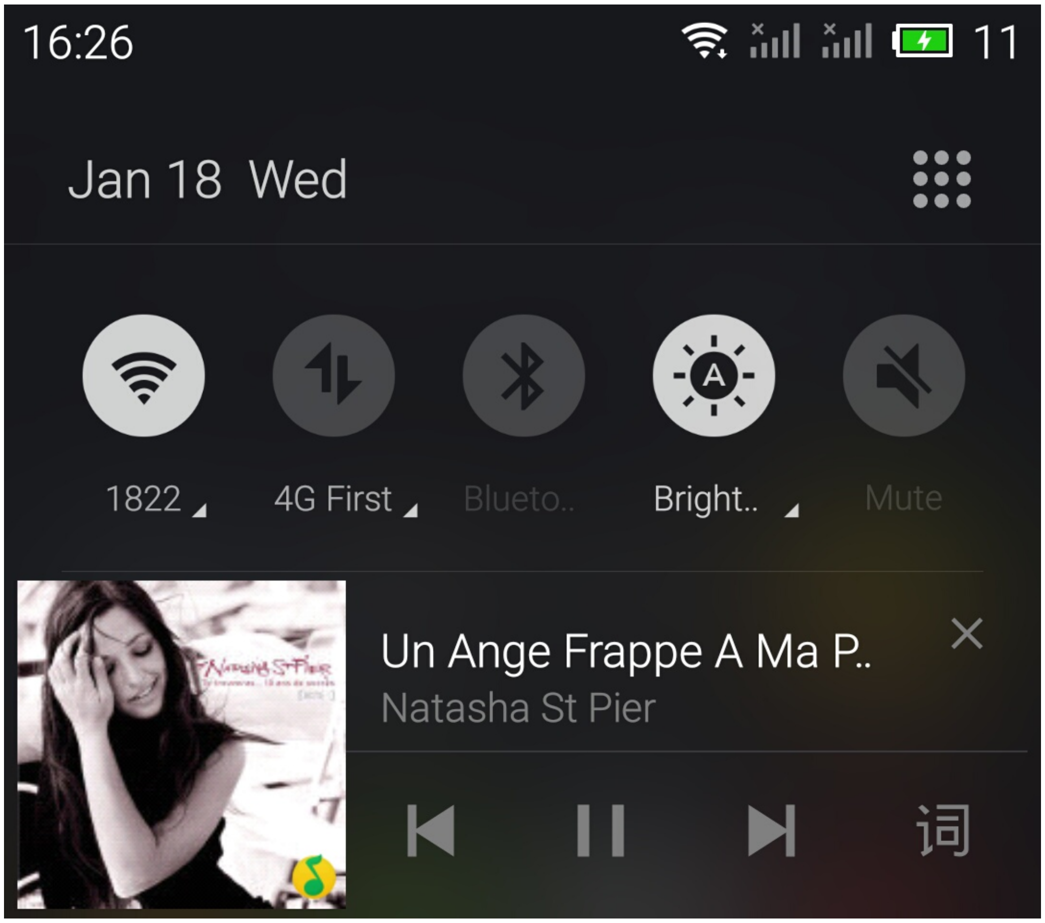


Figure 4.8: Music player control panel in notification bar.

4.4.2 Event Cluster

To avoid this phenomenon, we proposed the event cluster to solve the problem of the testing targeting notification bar. First, we detect all the Broadcast Receivers implemented in the Service. If the Broadcast Receiver can meet the transition condition and switch the state in the SUT, we will include the Broadcast Receiver as transition trigger in component-based model. If they trigger the transition, but it still return to the original state, we regard them as a part of the state instead of a transition in component-based model. As for the lock screen, we can regard it as a isolated state and test it as a solo graphical user interface, as most lock screen is prevented from transiting by the lock screen screen protector.

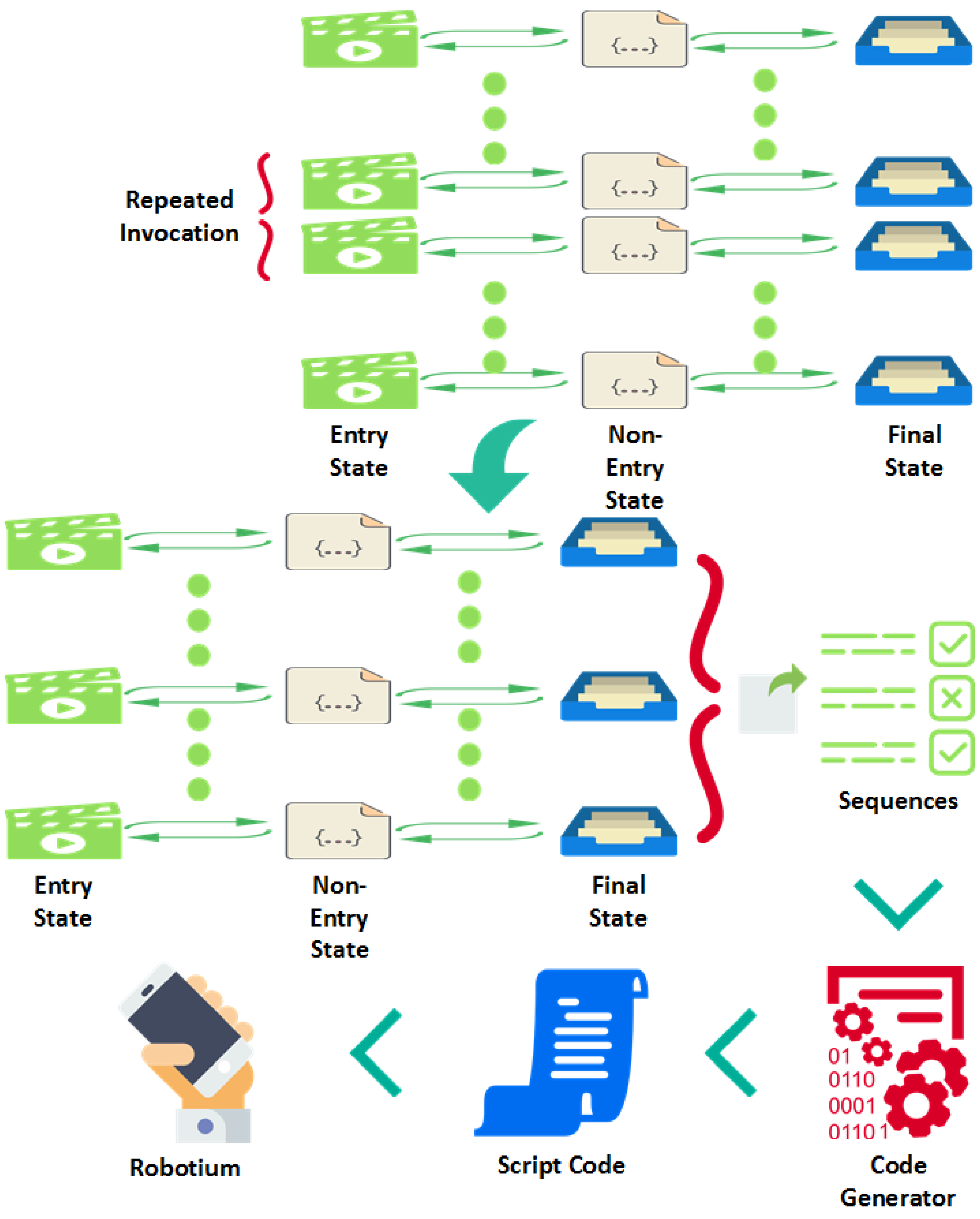


Figure 4.9: Music player control panel in notification bar.

Second, the notification's functions are relatively homogeneous. Most notifications triggered by same state always have similar functions. We regard this phenomenon as repeated multiple invocation of notification as repeated states in testing. We need to cluster them together for better performance in testing. The process of event cluster is shown in figure 4.9.

Figure 4.9 shows the process of event cluster. The event cluster takes the generated sequences as input. It then detects the repeated invocations in the sequences and deletes the repeated sequences. Then it generates the clustered sequences for test sample generation. Then the code generator translates the xml represented event sequences into script code for test case generation framework.

Finally, the script files can be fed to the automatic test samples generation framework to perform the automatic testing.

Chapter 5

Evaluation

In this chapter,we aims to prove the effectiveness & testing performance of the proposed approach. This chapter introduces the evaluation in two aspects.

In the first part, we discuss the work flow of our proposed approach as well as the implementation details. Then we use Gasflow as a study case to prove the effectiveness of our approach.

In the second part, we design three experiments to evaluate the performance of CamDroid. We also compare the proposed approach with traditional model-based testing method & classic fuzz testing method.

5.1 Effectiveness Evaluation

After explanation of methodology of the component-based model testing approach, we discuss the process of CamDroid implementation and design philosophy first. Then we use the illustrative app to demonstrate the work flow of CamDroid. Implementation aims to prove the effectiveness of the component-based model testing.

5.1.1 CamDroid Design

We have implemented the component-based modeling approach called CamDroid for component-based model test input generation.

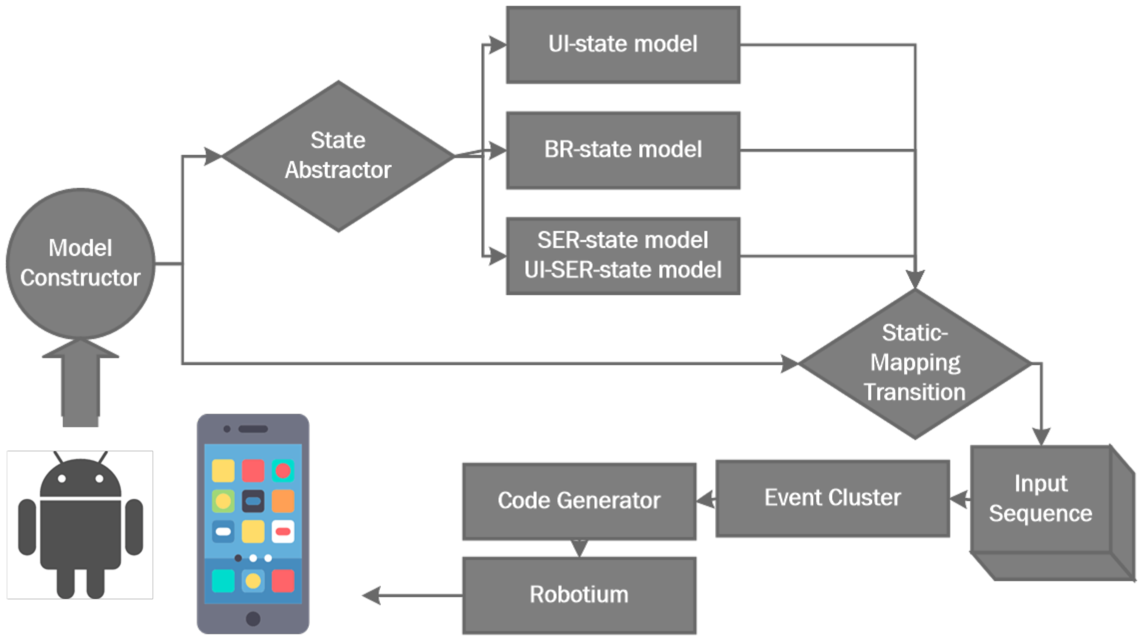


Figure 5.1: A high-level overview of CamDroid.

As shown in figure 5.1, three functional modules are defined in CamDroid. They are model constructor, static-mapping transition builder and event sequences generator & cluster. Model constructor can be further classified as UI-state constructor, SER-state constructor, BR-state constructor and UI-SER-state constructor. UI-state constructor is implemented on the top of TrimDroid [17]. Meanwhile SER-state,BR-state,UI-SER-state are implemented following the criteria of functional behavior analysis and state abstraction structure. Static-mapping transition builder and event sequences generator & cluster are implemented as stated in chapter 4.

CamDroid's work flow can be divided into four stages : Processing Initialization, Date Gathering, Data processing and Script Code Generation.

Processing Initialization uses ClassyShark and apktool to reverse engineer the APK file for further processing.

Data gathering mines the targeting program state and transition data both from smali code and soot analysis. Then CamDroid writes the gathered data to xml represented states template files for further processing.

Data processing mainly connects and clusters the data and translates them into format of test sample sequences.

Script code generation translate the sequence data into Robotium scripts and ADB shell if the event cannot be generated by Robotium.

This section discusses the design and work flow of CamDroid. In next section, we will discuss some implementation details of CamDroid.

5.1.2 Implementation Details

Process and methodology of CamDroid have been explicated above. We discusses some details in CamDroid's implementation in this part. This section explains CamDroid in implementation's perspectives.

For the convenience of implementation, we define five xml files containing states and transitions structure constrains following the criteria of the proposed approach. The smali code extraction related constrains will be put into code-constrain.xml file. The five predefined xml files describe the model structure of the component-based model and its abstraction level.

CamDroid detects all the states from the AndroidManifest.xml file. And the states are classified as entry state & non-entry states by category. Based on the registered data in the AndroidManifest.xml file. CamDroid can determines the number of different states preliminary. At the same time, Broadcast Receiver as transition trigger and SER-UI-State cannot be detected directly from AndroidManifest file. Soot will

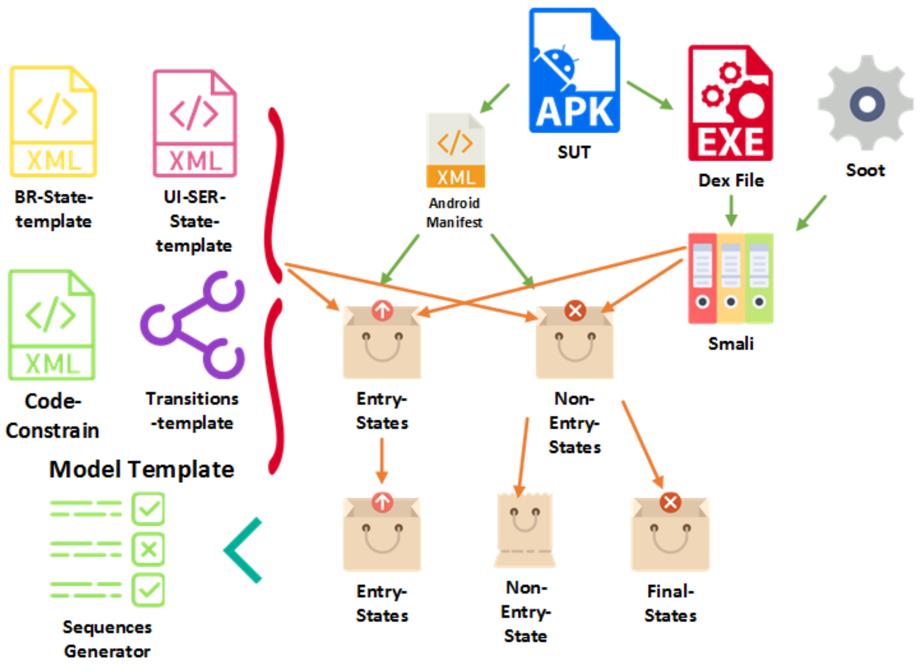


Figure 5.2: Implementation details of CamDroid.

detect them through program interface monitoring.

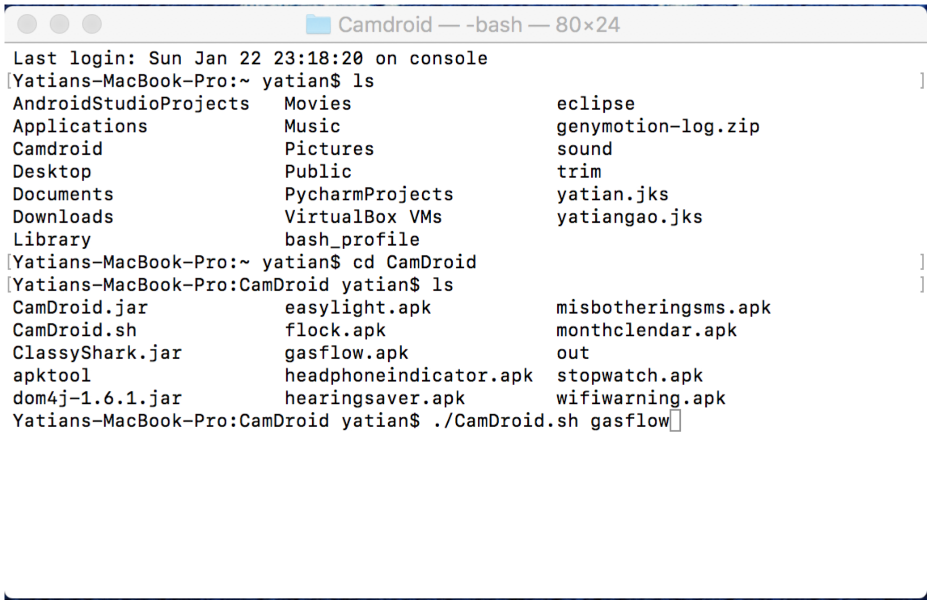
After knowing the states categories and numbers, corresponding states and transitions templates are created for further construction as shown in figure 5.2. Then the smali code analysis & soot analysis extract target data to fit into the templates. Based on the code investigation, the states & transitions templates are converted to states & transitions data for sequences generator. Then the data will be fed to event sequences generator & cluster.

This section discusses the implementation details of CamDroid. In next section, Gasflow will be executed in CamDroid to prove the effectiveness of CamDroid.

5.1.3 Case Study

This section uses gasflow as SUT to demonstrate the functions step by step. CamDroid is a command line tool tested on MAC OS. We can use the following command to execute the tool.

- # ./CamDroid.sh gasflow (App's name)



```
Last login: Sun Jan 22 23:18:20 on console
Yatians-MacBook-Pro:~ yatian$ ls
AndroidStudioProjects   Movies                  eclipse
Applications            Music                   genymotion-log.zip
Camdroid                Pictures                sound
Desktop                 Public                  trim
Documents               PycharmProjects         yatian.jks
Downloads               VirtualBox VMs          yatiangao.jks
Library                 bash_profile
Yatians-MacBook-Pro:~ yatian$ cd CamDroid
Yatians-MacBook-Pro:CamDroid yatian$ ls
CamDroid.jar            easylight.apk           misbotheringsms.apk
CamDroid.sh             flock.apk               monthclendar.apk
ClassyShark.jar         gasflow.apk             out
apktool                 headphoneindicator.apk  stopwatch.apk
dom4j-1.6.1.jar         hearingsaver.apk        wifiwarning.apk
Yatians-MacBook-Pro:CamDroid yatian$ ./CamDroid.sh gasflow
```

Figure 5.3: Demonstration of CamDroid execution (1).

After executing CamDroid, components in the application will be abstracting as states in the component-based model. As shown in figure 5.4, states and transitions are classified into seven different folders in the output directory. Model data is organized as extensible markup language.

CamDroid abstracts the components into states in the component-based model successfully. In the next section, we will have a detailed investigation on the component-based model.

BR-state in Gasflow

After CamDroid scanning gasflow, four Broadcast Receivers are detected in this application. PowerReceiver, BatteryReceiver and BootReceiver are registered statically. RecScreen receiver is registered dynamically.

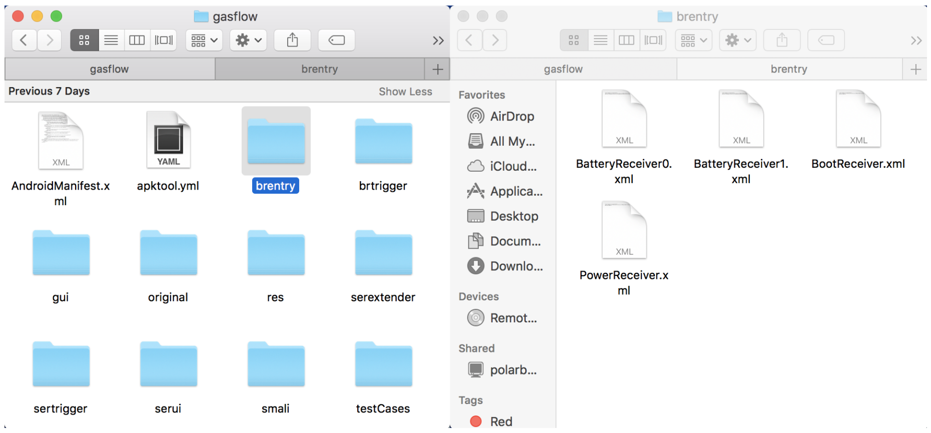
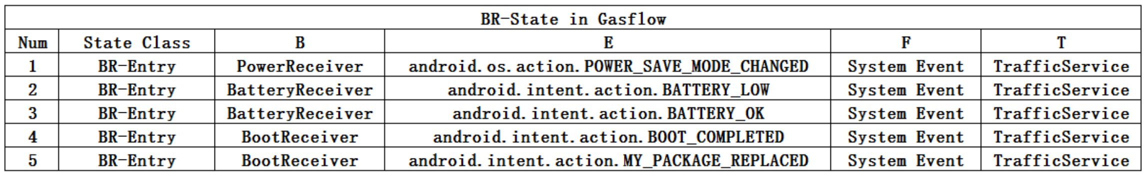


Figure 5.4: Demonstration of CamDroid execution (2).



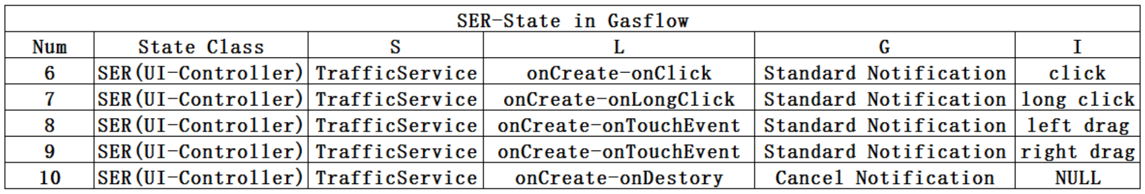
BR-State in Gasflow					
Num	State Class	B	E	F	T
1	BR-Entry	PowerReceiver	android.os.action.POWER_SAVE_MODE_CHANGED	System Event	TrafficService
2	BR-Entry	BatteryReceiver	android.intent.action.BATTERY_LOW	System Event	TrafficService
3	BR-Entry	BatteryReceiver	android.intent.action.BATTERY_OK	System Event	TrafficService
4	BR-Entry	BootReceiver	android.intent.action.BOOT_COMPLETED	System Event	TrafficService
5	BR-Entry	BootReceiver	android.intent.action.MY_PACKAGE_REPLACED	System Event	TrafficService

Figure 5.5: Broadcast Receiver abstracted states in Gasflow.

The BR-states can be classified as BR-Entry and BR-Trigger functionally. BR-Entry is defined as program entry and can invoke the applications by receiving outside events. BR-Trigger is a transitional state and usually bridges different kinds of states in the component-based model. This is a intermediate process of CamDroid.

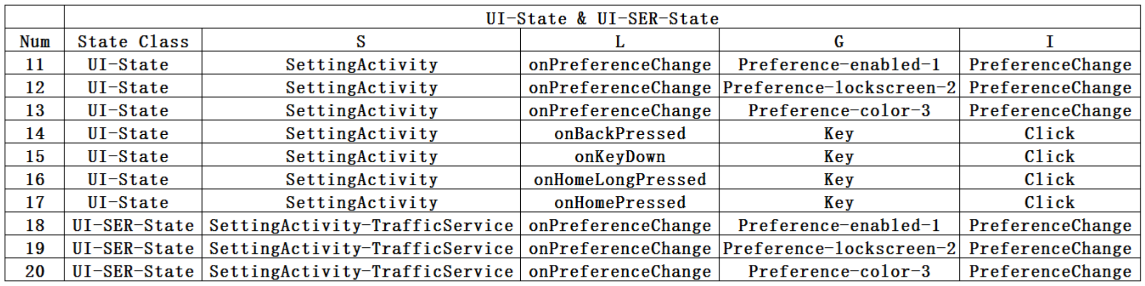
SER-State & UI-SER-State in Gasflow

After scanning by CamDroid, one Service is detected as SER-State. As there is only one standard notification controlled by Service. Affiliated data gathered from gasflow is shown in figure 5.6.



SER-State in Gasflow					
Num	State Class	S	L	G	I
6	SER(UI-Controller)	TrafficService	onCreate-onClick	Standard Notification	click
7	SER(UI-Controller)	TrafficService	onCreate-onLongClick	Standard Notification	long click
8	SER(UI-Controller)	TrafficService	onCreate-onTouchEvent	Standard Notification	left drag
9	SER(UI-Controller)	TrafficService	onCreate-onTouchEvent	Standard Notification	right drag
10	SER(UI-Controller)	TrafficService	onCreate-onDestory	Cancel Notification	NULL

Figure 5.6: Service abstracted states in Gasflow.



	UI-State & UI-SER-State				
Num	State Class	S	L	G	I
11	UI-State	SettingActivity	onPreferenceChange	Preference-enabled-1	PreferenceChange
12	UI-State	SettingActivity	onPreferenceChange	Preference-lockscreen-2	PreferenceChange
13	UI-State	SettingActivity	onPreferenceChange	Preference-color-3	PreferenceChange
14	UI-State	SettingActivity	onBackPressed	Key	Click
15	UI-State	SettingActivity	onKeyDown	Key	Click
16	UI-State	SettingActivity	onHomeLongPressed	Key	Click
17	UI-State	SettingActivity	onHomePressed	Key	Click
18	UI-SER-State	SettingActivity-TrafficService	onPreferenceChange	Preference-enabled-1	PreferenceChange
19	UI-SER-State	SettingActivity-TrafficService	onPreferenceChange	Preference-lockscreen-2	PreferenceChange
20	UI-SER-State	SettingActivity-TrafficService	onPreferenceChange	Preference-color-3	PreferenceChange

Figure 5.7: Service & Activity abstracted states in Gasflow.

UI-state & SER-UI-State in Gasflow

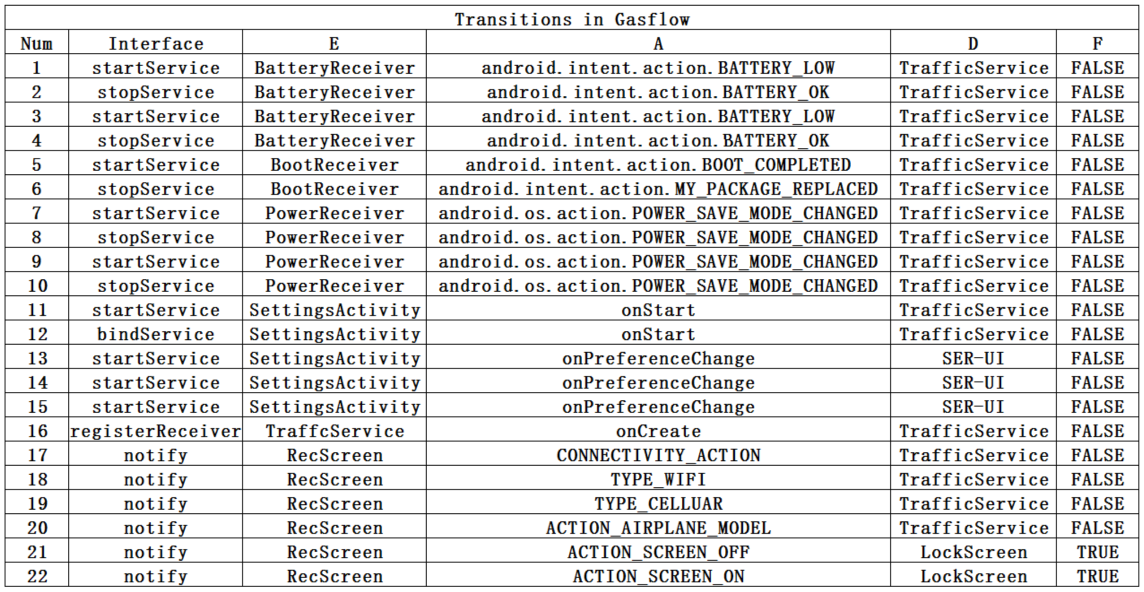
Figure 5.7 shows the UI-state and SER-UI-State in gasflow. SettingActivity is the only activity which abstracted as UI-state. Then the traffic Service and setting Activity are combined as SER-UI-State.

Transition in Gasflow

Figure 5.8 shows the transitions in Gasflow. As stated above, multiple program interfaces are defined as transition triggers.

After getting the models and transitions, we can use this data to construct a component-based model as shown in figure 3.7.

From the component-based model, five prime paths are detected as shown in figure 5.9. Event sequences are generated following the prime paths. Then Service and Broadcast Receiver related are generated towards the model. Then the events are clustered and inserted into the sequence.



Transitions in Gasflow					
Num	Interface	E	A	D	F
1	startService	BatteryReceiver	android.intent.action.BATTERY_LOW	TrafficService	FALSE
2	stopService	BatteryReceiver	android.intent.action.BATTERY_OK	TrafficService	FALSE
3	startService	BatteryReceiver	android.intent.action.BATTERY_LOW	TrafficService	FALSE
4	stopService	BatteryReceiver	android.intent.action.BATTERY_OK	TrafficService	FALSE
5	startService	BootReceiver	android.intent.action.BOOT_COMPLETED	TrafficService	FALSE
6	stopService	BootReceiver	android.intent.action.MY_PACKAGE_REPLACED	TrafficService	FALSE
7	startService	PowerReceiver	android.os.action.POWER_SAVE_MODE_CHANGED	TrafficService	FALSE
8	stopService	PowerReceiver	android.os.action.POWER_SAVE_MODE_CHANGED	TrafficService	FALSE
9	startService	PowerReceiver	android.os.action.POWER_SAVE_MODE_CHANGED	TrafficService	FALSE
10	stopService	PowerReceiver	android.os.action.POWER_SAVE_MODE_CHANGED	TrafficService	FALSE
11	startService	SettingsActivity	onStart	TrafficService	FALSE
12	bindService	SettingsActivity	onStart	TrafficService	FALSE
13	startService	SettingsActivity	onPreferenceChange	SER-UI	FALSE
14	startService	SettingsActivity	onPreferenceChange	SER-UI	FALSE
15	startService	SettingsActivity	onPreferenceChange	SER-UI	FALSE
16	registerReceiver	TraffcService	onCreate	TrafficService	FALSE
17	notify	RecScreen	CONNECTIVITY_ACTION	TrafficService	FALSE
18	notify	RecScreen	TYPE_WIFI	TrafficService	FALSE
19	notify	RecScreen	TYPE_CELLUAR	TrafficService	FALSE
20	notify	RecScreen	ACTION_AIRPLANE_MODEL	TrafficService	FALSE
21	notify	RecScreen	ACTION_SCREEN_OFF	LockScreen	TRUE
22	notify	RecScreen	ACTION_SCREEN_ON	LockScreen	TRUE

Figure 5.8: Transitions in Gasflow.

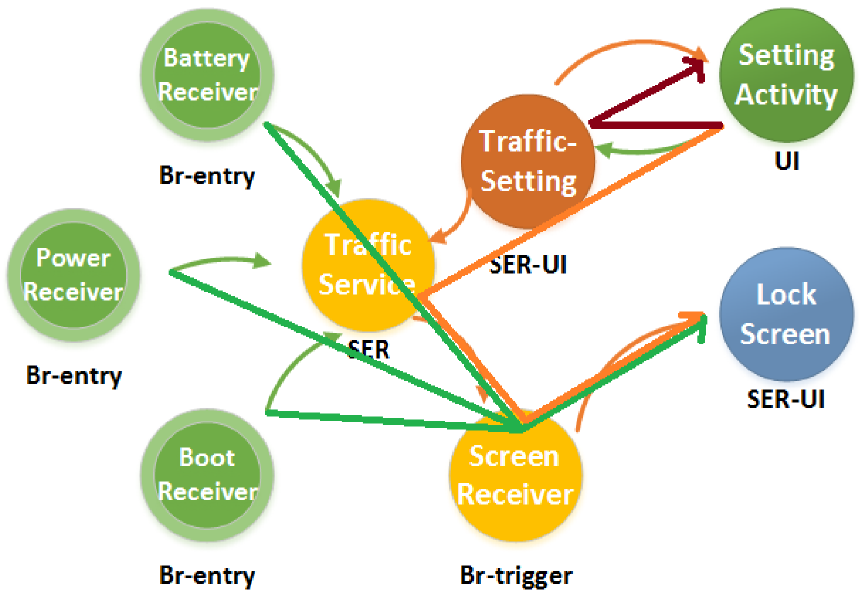


Figure 5.9: Prime paths in Gasflow.

Figure 5.10 shows the complete component-based model of Gasflow. The model show the behaviors across different components. Finally, the sequences are fed to the Android emulator to test the applications.

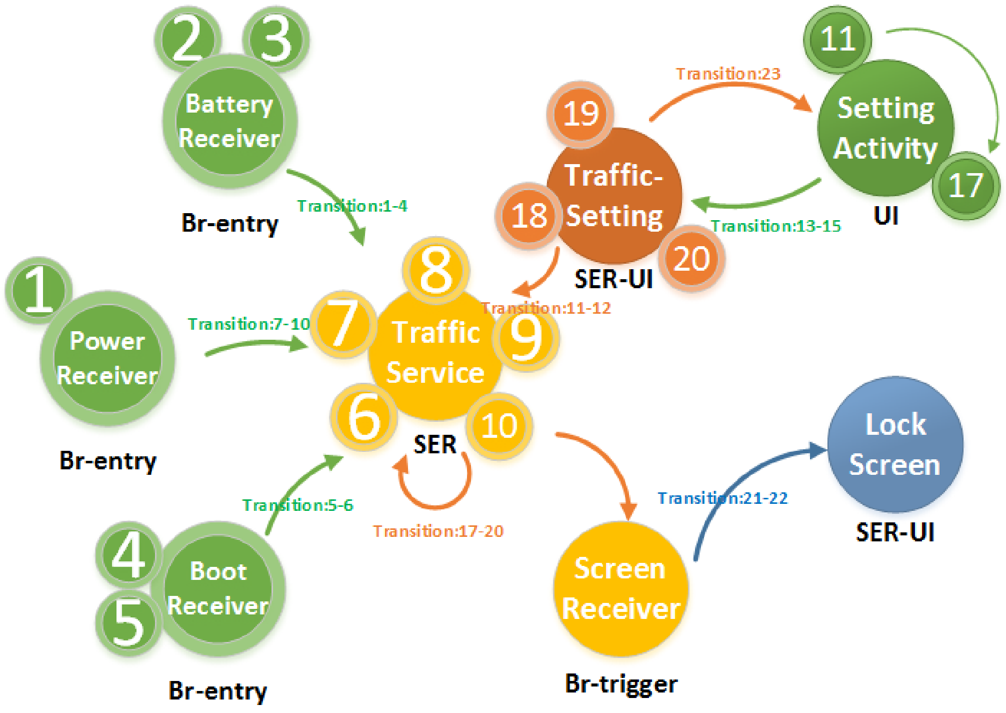


Figure 5.10: Component-based model of Gasflow.

This section demonstrates functions using the illustrative application. We will discuss the effectiveness in evaluation.

5.2 Performance Evaluation

Performance evaluation aims to prove the efficiency and progressiveness of the component-based model.

5.2.1 Requirement of Evaluation

In this section we evaluate the CamDroid′s ability of model construction and code coverage of input generation. To evaluate in a clear way we conduct case studies focusing on the following research questions:

- RQ1: Can the proposed component-based model cover the behaviors of Service and Broadcast Receiver in SUT successfully?
- RQ2: Can the transitions in the model constructed successfully by static-mapping transition builder?
- RQ3: Can the proposed approach generated inputs achieve a better coverage than previous model-based techniques?
- RQ4: Can the proposed approach overcome the challenges in the GUI-based model testing?

As CamDroid is practical in input generation, we select some released applications from GooglePlay and F-Droid. Selecting open source samples is for the convenience of testing and code coverage investigation.

5.2.2 Experiment

For analysis of proposed requirements, we choose 10 real Android applications as test samples and design three experiments to investigate all processes in model generation. Method in [33] are used to measure state coverage and code coverage in model-based testing. Emma[58] is used in our experiments for code coverage analysis.

We select 10 real Android applications released in GooglePlay & Fdroid as listed in table 5.1. We select these applications based on the following reasons. First, all chosen applications are real released application versions. They can check the practicability of our approach. Second, all the chosen applications are in different functional categories. They can represent the overall testing requirement on Android platform. Thirdly, the chosen applications cover all the component registration patterns in the

Table 5.1: Survey of problems in apps before testing

App Name	Problems			
-	Entry&Exit	Trans Neglect	Isolated State	Pseudo-GUI
Easylight(V1.1)	✓	✓	✓	✓
FLock(V1.0)	✓	✓	✓	✗
Gasflow(V1.0)	✓	✓	✓	✓
Headphone Indicator(V1.2)	✓	✓	✓	✗
Wifi warning(V1.0)	✓	✓	✗	✓
Calender(V1.1)	✗	✗	✓	✗
Hearing Saver(V1.0)	✓	✓	✓	✗
MisbotheringSMS(V1.0)	✓	✓	✗	✗
StopWatch(V1.4)	✓	✗	✓	✓
AnyCut(V1.0)	✗	✗	✗	✗

development stage. For example, AnyCut only registers one Activity, StopWatch register both Activity & Service. Wifi Warning, Calender and MisbotheringSMS register both Activity & Broadcast Receiver. The rest of the applications are registering all kinds of components in our model. They can cover all the situations in the development stage. As a summary, our testing samples selection can represent most testing situations on Android platform and prove the practicability of our approach.

In experiment 1, we investigate the generated model and inputs to determine whether the generated inputs can describe UI-state, BR-state, SER-state and UI-SER-state. Besides state abstraction, we test whether the generated states can be connected by generated transitions in experiment 2. Error transition generation may lead isolated state which cannot be covered in testing.

Experiment 1 & 2 are designed to prove functions in CamDroid qualitatively. In experiment 3, generated input samples will be fed to Robotium to execute lively for code coverage investigation (Some events cannot be executed in Robotium are executed manually).

Experiment 1 & 2 are designed to evaluate the modeling coverage and test ef-

Table 5.2: Survey of problems in apps after testing

App Name	Problems			
-	Entry&Exit	Trans Neglect	Isolated State	Pseudo-GUI
Easylight	✕	✕	✕	✕
FLock	✕	✕	✕	✕
Gasflow	✕	✕	✕	✕
Headphone Indicator	✕	✕	✕	✕
Wifi warning	✕	✕	✕	✕
Calender	✕	✕	✕	✕
Hearing Saver	✕	✕	✕	✕
MisbotheringSMS	✕	✕	✕	✕
StopWatch	✕	✕	✕	✕
AnyCut	✕	✕	✕	✕

fectiveness. Specifically we also need to evaluate whether the approach can solve the problem we rise in introduction and solve the dilemmas defined in the problem definition on the Android platform.

Experiment 3 is designed to investigate whether the proposed problems are solved or not.

From table 5.1 we can see most samples in experiment 3 has the proposed problem in the GUI-based model. Then we census whether the problems are solved after testing by component-based approach.

From table 5.2 we can see all the defined problems are eliminated by the component based model generated test. The identified problems are not problems in our component-based model testing.

5.2.3 Result Summary & Discussion

Table 3 shows result summary of experiments 1,3. State coverage shows the functional ability in the model constructor. Based on experiment 1, the average abstraction coverage is 100% in component-based model comparing to 47.3% average coverage

Table 5.3: Experiment result summary

App Name	Com-Number				State Cov		Code Coverage		
-	A	S	B	C	GUI-M	Com-M	Monkey	Trim	Cam
Easylight	2	1	1	0	50%	100%	43%	47%	74%
FLock	1	2	2	0	20%	100%	15%	35%	68%
Gasflow	1	1	4	0	20%	100%	35%	28%	77%
Headphone Indicator	1	1	1	0	33%	100%	39%	39%	83%
Wifi warning	2	0	2	0	50%	100%	31%	54%	69%
Calender	1	0	1	0	50%	100%	26%	40%	55%
Hearing Saver	1	1	2	0	25%	100%	28%	36%	46%
MisbotheringSMS	1	0	1	0	50%	100%	43%	56%	81%
StopWatch	3	1	0	0	75%	100%	46%	55%	78%
AnyCut	1	0	0	0	100%	100%	8%	58%	58%

in the GUI-based model. As for the the isolated state for showing the function of transition construction, the component-based model enables to build transitions across different components.

In experiment 3, two baseline methods are introduced for comparison. They are Monkey and TrimDroid. Monkey is a testing method following random input generation strategy. TrimDroid is one most updated tool with functional advantages from model-based and combinatorics testing. Model-based techniques are used to describe behaviors in apps while combinatorics testing for complex interface analysis.

As for a horizontal comparison, Monkey's average code coverage is 31.4 % and TrimDroid's average code coverage is 44.8 %. The Camdroid's average code coverage is 68.9 %. Our proposed approach has an overall advantage towards the traditional model-based testing. Specifically, the code coverage has a different increase towards different applications. For example, there is no code coverage increase in AnyCut because there is only one Activity registered in AnyCut. In this situation, component-based model is equal to the GUI-based model. In other words, the proposed component-based model doesn't benefit the code coverage since the components newly introduced into the model don't exist in AnyCut. Some code coverage increase is slight when only a few SER-states or BR-states are registered. Moreover, some code coverage increase is huge when the application is highly relying on SER-state , BR-state and has a large number in Services & Broadcast Receivers. Based on these facts, we can conclude that the more complex the model's structure is, the more effective our proposed approach is.

As a result, code coverage efficiency differences is clear. CamDroid helps to achieve better code coverage in real world testing.

To aim at a better evaluation effectiveness, some factors need to be discussed. For state abstraction in experiment one, functional inclusion determines the abstraction coverage. Elaborate abstraction can increase the model accuracy but implementation work load increases exponentially. For experiment 2, transitions build across components relies on the monitored program interface. The more program interfaces and code patterns are monitored, the more transitions across components will be built by CamDroid. In the third experiment, we can see the model-based testing techniques have a systematic advantage towards random-based testing techniques. Furthermore CamDroid achieves a better code coverage with an average code coverage of 68.9%.

Based on evaluation, CamDroid can accomplish its tasks functionally and meet the RQ 1,2,3. Although CamDroid is still a research prototype of the proposed approach, it has revealed enormous potential in future testing techniques.

Chapter 6

Conclusions & Future Work

In this chapter, we discuss the conclusions, limitations & our future works.

6.1 Conclusion

We present an automated component-based model generation approach in this thesis. Our approach enables to construct a novel model covering the behaviors of Activities, Services and Broadcast Receivers. CamDroid can meet the evaluation requirements as seen by results of experiments.

Based on the experiments and case study in this work, we are able to conclude towards our study. They are summarized as following:

- The component-based model generation approach enables to describe application as a component-based model in a larger scope compared to the GUI-based model. The component-based model can abstract Activity, Service and Broadcast Receiver as states. These newly introduced states can describe the behaviors in Services and Broadcast Receivers.
- The component-based model solve the defined problems of testing in Android

system successfully. As shown in evaluation experiments, the component-based model testing can cover most behaviors in the samples. It also can eliminate the problems in the GUI-based model.

- The component-based model achieves a better code coverage comparing to the GUI-based model in model-based testing. Based on our evaluation, CamDroid achieves a average code coverage of 68.9% . Its efficiency is better than random test generation techniques' code coverage (31.4%) and traditional model-based testing (44.8%).

6.2 Limitations

The proposed component-based approach ameliorates traditional GUI-based model, but it is still not a final solution of model-based testing on Android platform. There are several limitations in the prototype of CamDroid. These limitations directs the future works of model-based testing on Android platform.

6.2.1 Component Absence

Normally there are four components in the Android system. They are Activity, Service, Broadcast Receiver and Content Provider as shown in figure 6.1.

The component-based model does not include the Content Provider as its state. Neglecting Content Provider in the component-based model may cause some problems in testing effectiveness.

- The component-based model cannot describe the behaviors in the Content Provider.

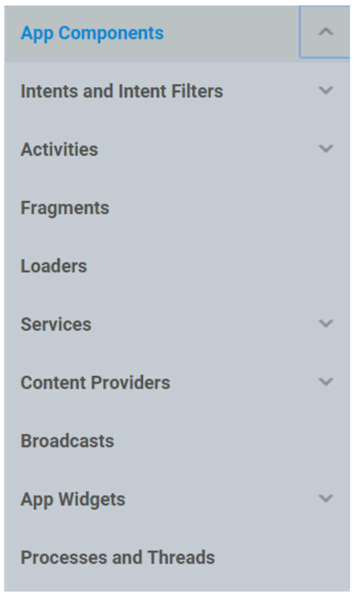


Figure 6.1: App components in Android system.

- The generated test's invalidity will decrease when dealing with the Content Provider related graphical user interfaces and behaviors.

6.2.2 System Event Generation

All the model-based testing techniques are dependent on the test generation frameworks to generate test samples. As the GUI-based model develops for a long periods of time, most of them are powerful in graphical user interface related events. Almost no test generation tools are efficient in system event generation.

As more and more Android applications' transition conditions are triggered by system events. So the component-based model needs a generation tool which can generate system events efficiently. However, almost all existing tools cannot generate system event efficiently shown in table 6.1.

Table 6.1: Survey of automatic test sample generation tools

App Name	Survey				
-	Robotium	UIautomator	Espresso	Appium	Calabash
Android	✓	✓	✓	✓	✓
IOS	✗	✗	✗	✓	✓
Mobile Web	✓	x,y click	✗	✓	✓
Scripting	Java	Java	Java	Multiple	Ruby
Test Tools	Recorder	Viewer	Viewer	Appium.app	CLI
API Level	All	15 Above	15 Above	All	All
Community	Contributors	Google	Google	Active	Not Active

6.3 Future Work

The limitations provide us some direction to the future work. Thus the limitations are concentrating on the test sample generation and model structure. So our future work will focus on the further amelioration of model structure and test generation strategy optimization.

6.3.1 Model Structure Amelioration

The component-based model had ameliorated the model's structure based on the GUI-based model. But the Content Provider is still out the scope of the component-based model.

As a result, the proposed component-based model is still imperfect in model-based testing. We need to include more components into the model for better testing effectiveness in the future work.

6.3.2 Test Generation Optimization

As the GUI-based model is popular in Android application's testing, most test generation tools are concentrating on the GUI-related events. Although Monkey[35] can generate system events, its random test generation cannot satisfy the model-based testing requirements.

At last, we need to summarize the demand of model-based testing in test generation. And we need to make the test generation processes more efficient and automatic in the near future.

6.3.3 Application's Security Testing

Based on the evaluation above, we can see the ability to achieve a better code coverage in the testing. In other words, the proposed approach has the ability to find hidden behaviors defined in the Android application. This is also the basis of application's security analysis.

In the near future, we can include the suspicious behaviors patterns into our model to perform the security centric Android application testing.

Bibliography

[1] Q2,2016,http://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

[2] 2015,http://www.idc.com/getdoc.jsp?containerId=prUS41061616

[3] Android Wear, https://www.android.com/wear

[4] Raspberry Pi, https://www.raspberrypi.org/

[5] Ten million Raspberry Pi, https://www.raspberrypi.org/blog/ten-millionth-raspberry-pi-new-kit/

[6] Activity, Service, Broadcast Receiver, Content Provider, https: //developer.android.com/guide/components/index.html

[7] Loklok, https://play.google.com/store/apps/details?id=co.loklok&hl=en

[8] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In Proceedings of the 30th international Design Automation Conference (DAC '93). ACM, New York, NY, USA, 86-91, 1993.

[9] System Under Test, https://en.wikipedia.org/wiki/System under test

[10] http://www.opera.com/

[11] https://play.google.com/store/apps/details?id=com.google.android.apps.messaging&hl=en

[12] https://google-maps.en.softonic.com/android

[13] https://play.google.com/store/apps/details?id=com.android.chrome&hl=en

[14] https://play.google.com/store/apps/details?id=com.UCMobile.intl&hl=en

[15] https://play.google.com/store/apps/details?id=com.supercell.clashofclans&hl=en

[16] Wei Yang, Mukul R. Prasad, and Tao Xie. A grey-box approach for automated GUI-model generation of mobile applications. In Proceedings of the 16th international conference on Fundamental Approaches to Software Engineering (FASE'13), Vittorio Cortellessa and Dániel Varró (Eds.). Springer-Verlag, Berlin, Heidelberg, 250-265, 2013.

[17] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in GUI testing of android applications. In Proceedings of the 38th International Conference on Software Engineering (ICSE '16). ACM, New York, NY, USA, 559-570, 2016.

[18] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86, 8 (August 2013), 1978-2001.

[19] http://classyshark.com/

[20] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative

research (CASCON '99), Stephen A. MacKay and J. Howard Johnson (Eds.). IBM Press 13-, 1999.

[21] http://alloy.mit.edu/alloy/

[22] Mu Zhang, Yue Duan, Qian Feng, and Heng Yin. Towards Automatic Generation of Security-Centric Descriptions for Android Apps. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15). ACM, New York, NY, USA, 518-529, 2015.

[23] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta and A. M. Memon, "MobiGUITAR: Automated Model-Based Testing of Mobile Apps," in IEEE Software, vol. 32, no. 5, pp. 53-59, Sept.-Oct. 2015.

[24] W. E. C. K. Y.Fratantonio, A.Bianchi and G.Vigna. Triggerscope: Towards detecting logic bombs in android applications. in Proc. 37th IEEE Symposium on Security and Privacy (SP'16), May 2016.

[25] GUIRipper *https : //www.cs.umd.edu/ atif/GUITAR−Web/gui−ripper.htm*

[26] Wontae Choi, George Necula, and Koushik Sen. Guided GUI testing of android apps with minimal restart and approximate learning. In Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications (OOPSLA '13). ACM, New York, NY, USA, 623-640, 2013.

[27] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In Proceedings of the 12th annual international conference on Mobile systems, applications, and services (MobiSys '14). ACM, New York, NY, USA, 204-217, 2014.

[28] https://www.whatsapp.com/

[29] https://www.google.com/drive/

[30] https://play.google.com/store/apps/details?id=com.mridang.speedo

[31] https://github.com/RobotiumTech/robotium

[32] https://developer.android.com/guide/components/activities.html

[33] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for Android apps. In Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013). ACM, New York, NY, USA, 224-234, 2013.

[34] Widget Guide https://developer.android.com/design/patterns/widgets.html

[35] Monkey https://developer.android.com/studio/test/monkey.html

[36] Android Not Response https://developer.android.com/training/articles/perf-anr.html

[37] Ali Mesbah, Arie van Deursen, and Stefan Lenselink. Crawling Ajax-Based Web Applications through Dynamic Analysis of User Interface State Changes. ACM Trans. Web 6, 1, Article 3 (March 2012), 30 pages, 2012.

[38] Atif Memon, Ishan Banerjee, and Adithya Nagarajan. GUI Ripping: Reverse Engineering of Graphical User Interfaces for Testing. In Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03). IEEE Computer Society, Washington, DC, USA, 260-, 2003.

[39] Valentin Dallmeier, Martin Burger, Tobias Orth, and Andreas Zeller. WebMate: a tool for testing web 2.0 applications. In Proceedings of the Workshop on JavaScript Tools (JSTools '12). ACM, New York, NY, USA, 11-15, 2012.

[40] Joorabchi, M.E., Mesbah, A.: Reverse engineering iOS mobile applications. In: Proc.19th Working Conference on Reverse Engineering.WCRE '12 (2012) 177-186, 2012.

[41] Arilo C. Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H. Travassos. A survey on model-based testing approaches: a systematic review. In Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007 (WEASELTech '07). ACM, New York, NY, USA, 31-36, 2007.

[42] Takala, T., Katara, M., Harty, J. Experiences of system-level model-based GUI testing of an Android application. In: Proc. 2011 4th IEEE International Conference on Software Testing, Verication and Validation. ICST '11 (2011) 377-386, 2011.

[43] A. Jaaskelainen, A. Kervinen, and M. Katara, "Creating a test model library for GUI testing of smartphone applications," in Proc. QSIC 2008 (short paper). IEEE CS, 2008, pp. 276–282, 2008.

[44] Mark Utting, Alexander Pretschner, and Bruno Legeard. A taxonomy of model-based testing approaches. Softw. Test. Verif. Reliab. 22, 5 (August 2012), 297-312, 2012.

[45] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of An-

droid applications. In Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012). ACM, New York, NY, USA, 258-261, 2012.

[46] Cu D. Nguyen, Alessandro Marchetto, and Paolo Tonella. Combining model-based and combinatorial testing for effective test case generation. In Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA 2012). ACM, New York, NY, USA, 100-110, 2012.

[47] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 641–660, New York, NY, USA, 2013.

[48] H. Ye, S. Cheng, L. Zhang, and F. Jiang. DroidFuzzer: Fuzzing the Android Apps with Intent-Filter Tag. In Proceedings of International Conference on Advances in Mobile Computing & 38; Multimedia, MoMM '13, pages 68:68–68:74, New York, NY, USA, 2013.

[49] R. Sasnauskas and J. Regehr. Intent Fuzzer: Crafting Intents of Death. In Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA), WODA+PERTEA 2014, pages 1–5, New York, NY, USA, 2014.

[50] Intent fuzzer, 2009. http://www.isecpartners.com/tools/mobile-security/intent-fuzzer.aspx.

[51] T. Azim and I. Neamtiu. Targeted and Depth-first Exploration for Systematic Testing of Android Apps. In Proceedings of the 2013 ACM SIGPLAN Inter-

national Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA '13, pages 641–660, New York, NY, USA, 2013. ACM.

[52] R. Mahmood, N. Mirzaei, and S. Malek. EvoDroid: Segmented Evolutionary Testing of Android Apps. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2014, New York, NY, USA, 2014.

[53] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated Concolic Testing of Smartphone Apps. In Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE '12, pages 59:1–59:11, New York, NY, USA, 2012.

[54] H. van der Merwe, B. van der Merwe, and W. Visser. Verifying Android Applications Using Java PathFinder. SIGSOFT Softw. Eng. Notes, 37(6):1–5, Nov. 2012.

[55] Pokemon Go : http://www.pokemongo.com/fr-ca/

[56] Rinna : http://www.msxiaoice.com/

[57] http://mse.isri.cmu.edu/software-engineering/documents/faculty-publications/miranda/ kuhnintroductioncombinatorialtesting.pdf

[58] http://emma.sourceforge.net/