Automated Model-Based Android GUI Testing using Multi-level GUI Comparison Criteria

Young-Min Baek, Doo-Hwan Bae
Korea Advanced Institute of Science and Technology (KAIST)
Daejeon, Republic of Korea
{ymbaek, bae}@se.kaist.ac.kr

ABSTRACT

Automated Graphical User Interface (GUI) testing is one of the most widely used techniques to detect faults in mobile applications (apps) and to test functionality and usability. GUI testing exercises behaviors of an application under test (AUT) by executing events on GUIs and checking whether the app behaves correctly. In particular, because Android leads in market share of mobile OS platforms, a lot of research on automated Android GUI testing techniques has been performed. Among various techniques, we focus on model-based Android GUI testing that utilizes a GUI model for systematic test generation and effective debugging support. Since test inputs are generated based on the underlying model, accurate GUI modeling of an AUT is the most crucial factor in order to generate effective test inputs. However, most modern Android apps contain a number of dynamically constructed GUIs that make accurate behavior modeling more challenging. To address this problem, we propose a set of *multi-level GUI Comparison Criteria (GUICC)* that provides the selection of multiple abstraction levels for GUI model generation. By using *multi-level GUICC*, we conducted empirical experiments to identify the influence of *GUICC* on testing effectiveness. Results show that our approach, which performs model-based testing with *multi-level GUICC*, achieved higher effectiveness than activity-based GUI model generation. We also found that *multi-level GUICC* can alleviate the inherent state explosion problems of existing a single-level *GUICC* for behavior modeling of real-world Android apps by flexibly manipulating *GUICC*.

CCS Concepts

•**Software and its engineering** → **Software testing and debugging;** Empirical software validation;

ASE'16, September 3–7, 2016, Singapore, Singapore
© 2016 ACM. 978-1-4503-3845-5/16/09...$15.00
http://dx.doi.org/10.1145/2970276.2970313

Keywords

GUI testing, Android application testing, GUI model generation, GUI comparison criteria, Model-based test input generation

1. INTRODUCTION

One of the most widely used methodologies to test mobile applications (apps) is *Graphical User Interface testing (GUI testing)* [13]. GUI testing is usually conducted as a validation technique for GUI-driven software such as mobile apps, and it mainly performs functional tests for an application under test (AUT). By running test inputs (e.g., `click Button-A`) on GUIs of an AUT, GUI testing examines behaviors of the app and checks whether it behaves correctly [7, 18, 19]. Since manual GUI testing is costly, time-consuming, and not sophisticated at finding real faults, automated GUI testing techniques and tools for mobile apps have been actively researched to perform cost-effective testing [13, 14, 15]. In particular, as the Android platform has dominated the market share among mobile OS platforms, automated GUI testing for Android apps has been getting the spotlight from the researchers and practitioners [9, 13, 10, 16, 12].

As a means of GUI test automation for Android apps, current studies that deal with automated test input generation for Android apps can be classified into several approaches [9, 8]. The simplest methodology is random test input generation, which generates random events as test inputs to explore behaviors of AUTs and checks runtime errors [4, 6]. *Android Monkey* [10] is the best-known random tool for robustness testing, and Dynodroid [16] is an advanced random testing tool that guides random testing to generate only relevant events. Although these random testing techniques (or tools) could explore the behavior space of an AUT in simple ways, not only do they generate redundant tests due to the nature of randomness, but tracing suspicious paths is also more difficult than systematic approaches [9].

As a solution, model-based GUI testing techniques have been researched to facilitate the systematic exploration of a behavior space and support effective debugging [3, 15]. The model-based technique builds a GUI model that represents a behavior space of an AUT and generates test inputs based on the model [2, 1]. Compared to the random techniques, model-based approaches can generate a finite set of effective test inputs by analyzing the exploration context based on GUI models. Furthermore, they have advantages in systematic debugging by enabling their models to provide concrete execution paths related to the detected errors. Aided by these advantages, the model-based testing techniques be-

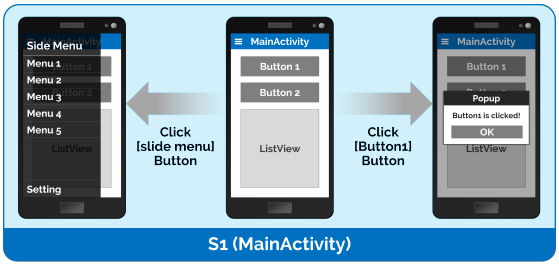
came major approaches for automated Android GUI testing, and there has been a great deal of research to improve the effectiveness of their techniques [8, 5].

For effective model-based GUI testing, reflecting a behavior space of an AUT on a GUI model is the most crucial ability to generate effective test inputs [8]. However, as many real-world Android apps contain a lot of dynamic or unpredictable GUIs to provide convenient user experience, existing model-based GUI testing techniques for Android apps have encountered difficulties in accurate modeling. More specifically, a GUI model that contains only a small range of possible behavior space can have poor the testing effectiveness, and dynamic behaviors in GUIs can cause inconsistent model generation or state explosion problems due to the non-deterministically changing GUIs.

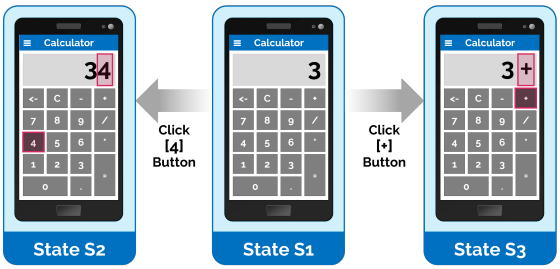
In order to tackle the behavior modeling problems of real-world apps, model-based techniques have to specify the level of behavior abstraction by clearly defining a *GUI comparison criterion*. *GUI comparison criteria (GUICC)* are the information related to the GUI to distinguish GUI states (e.g., name of activity, number of enabled widgets): whether two GUIs have an equivalent state or different GUI states. For behavior modeling and abstraction, previous studies have defined their own criteria to distinguish GUI states to generate models: an activity name, a set of event handlers, and observable graphical changes [27, 4, 6, 22]. However, they overlooked the volatile behaviors of real-world Android apps, and they have not been delved deeply into considering the influence of the *GUICC* on the testing effectiveness.

As simple examples of *GUICC*, Figure 1 illustrates two types of comparison criteria for GUI model generation. Figure 1-(a) shows a case where activity names are used as a comparison criterion, which was used in [6, 22]. The activity-based GUI model can be easily made with tools; however, an activity-level GUI model is too abstract to model dynamically constructed GUIs in recent Android apps. As illustrated in Figure 1-(a), activity-based comparison could miss a lot of information needed to adequately explore the behavior space because the three screens are not considered as distinct states. On the other hand, if the GUI comparison criterion is too sensitive, a testing tool could not complete GUI model generation due to the endlessly increasing behavior space. Figure 1-(b) shows a case where a testing tool distinguishes GUI states with every observable change in a calculator app. Since every single change in the displayed calculated value makes new GUI states, the GUI model generation may have to store redundant GUI states unnecessarily. Therefore, model-based GUI testing techniques should carefully consider the *GUICC* for model generation of real-world Android apps that contain complicated behaviors.

In this paper, we focus on the GUI model generation technique, especially on the GUI comparison criteria for model generation. This study (1) defines a ***Multi-Level GUI Comparison Criteria (Multi-Level GUICC)*** by suggesting a set of abstraction levels for automated model-based Android GUI testing. We also (2) develop an automated model-based GUI testing framework to actually perform automated GUI testing for real-world Android apps, and we evaluate the influence of *GUICC* on the testing results. In an evaluation on 20 open-source and commercial Android apps, we analyze the experimental results in terms of the modeling of behaviors, the code coverage, and the error detection.



(a) Behavior modeling with weak a GUI comparison criterion *(Activity name)*



(b) Behavior modeling with a strong (sensitive) GUI comparison criterion *(Observable changes in screenshots)*

Figure 1: Behavior modeling of Android GUIs based on GUI comparison criteria

This paper is organized as follows. Chapter 2 provides background information of model-based GUI testing for Android apps. Chapter 3 introduces our main approach, *Multi-level GUICC*, and Chapter 4 presents an automated model-based GUI testing framework that contains *Multi-level GUICC*. Chapter 5 provides our research questions and experimental setup, and Chapter 6 discusses the results of the experiments. Chapter 7 provides related work, and Chapter 8 concludes this paper with future research directions.

2. BACKGROUND

2.1 GUI Model

In model-based GUI testing, a key element that enables systematic test input generation is the underlying GUI model. The GUI models are usually in the form of Finite State Machines (FSMs), and a GUI model represents a behavior space of the target AUT with GUI states and event-driven transitions between GUI states [18]. In this study, we call a GUI model a "*GUI graph*," and it is defined as follows.

Definition. A *GUI graph* ***G*** is a directed graph that represents distinct GUI states, distinguished by a specific *GUI comparison criterion (GUICC)*, as *ScreenNodes*, and represents transitions between GUI states, which are triggered by events, as *EventEdges*. Simply, a *GUI graph* ***G*** is defined as ***G=(S,E)***, where ***S*** is a set of *ScreenNodes* (***S***=$\{s_1, s_2, ..., s_n\}$) and ***E*** is a set of *EventEdges* (***E***=$\{e_1, e_2, ..., e_n\}$).

- A *ScreenNode* represents a GUI state that contains abstracted GUI information of an execution screen of an AUT, and *ScreenNodes* are distinct from each other based on a *GUICC*.
- A *GUI Comparison Criterion (GUICC)* represents a specific type of GUI information to determine equivalence/difference between GUI states.
- An *EventEdge* represents a single event that can be triggered both by a user (e.g., `click`) and by a system (e.g., `SMS, phone calls`), and each *EventEdge* links between *ScreenNodes*.

In remaining chapters, we construct GUI graphs with *ScreenNodes* and *EventEdges* to model event-driven behaviors of an AUT and generate test inputs. Therefore, the GUI graph should represent as much behavior space (i.e., GUI states and event interactions) as possible, in order to generate adequate test inputs to explore the behavior space.

2.2 GUI Comparison Criteria (GUICC)

As mentioned in Section 2.1, a *GUICC* designates specific GUI information to distinguish the equivalence or difference between multiple GUI states. When a model-based testing tool generates or updates a GUI graph while exploring the behavior space, *GUICC* distinguishes a GUI of the current screen on Android device after a test input is executed. By comparing GUIs with *GUICC*, the testing tool determines how to update the GUI graph. If a testing tool uses a strong (sensitive) *GUICC* (e.g., observable graphical change), the abstraction level of generated GUI models decreases because more GUI states are regarded as distinct ones, and vice versa. In other words, a *GUICC* determines the granularity of *ScreenNodes* in a GUI graph, which can affect test input generation. For this reason, the *GUICC* could significantly affect the behavior modeling.

More specifically, a model-based testing tool repeatedly executes test inputs on a running AUT and analyzes the execution results to generate and update the GUI model. After an event $\boldsymbol{e}$ is executed on a certain screen s_c, the GUI graph should be updated to reflect the currently exercised behavior (i.e., a behavior triggered by $\boldsymbol{e}$) from the observed results (i.e., a state after event execution). The update process is illustrated in Figure 2, and it displays two cases: (1) the GUI state changes to an existing state or stays the same; (2) the GUI state changes to a new state that has a different GUI from other *ScreenNodes* in the GUI graph.

In case (1), if a GUI state after event execution $\boldsymbol{s'}$ is judged to be the same GUI state as a certain visited screen s_i of the GUI graph (i.e., $\boldsymbol{s'} = s_i$), then only a new *EventEdge* is added into the graph and it links two *ScreenNodes* s_c and s_i. Otherwise, as in case (2), if the event execution causes a change to a new GUI state, then a new *ScreenNode* (s_5) is added into the GUI graph, and a new *EventEdge* is also added into the graph to link s_c and s_5.

During the update process, the *GUICC* determines whether a screen after event execution $\boldsymbol{s'}$ has a new GUI state, which differs from other GUI states of visited *ScreenNodes*. For the comparison of GUI information between *ScreenNodes*, a model generation module inserts specific GUI information into each *ScreenNode* based on the chosen *GUICC* when a new node is added into the graph. For example, if a testing tool utilizes an activity name as a *GUICC*, then the activity information such as `com.example.MainActivity` is included in *ScreenNodes*.

3. METHODOLOGY

3.1 Multi-Level GUI Comparison Criteria

In the previous sections, we emphasized the role and the importance of *GUICC* in model-based GUI testing, and we mentioned some previous studies that did not define the criteria clearly or used improper ones. To come up with a clear definition of GUICC for the modeling of real-world Android apps, we propose a set of *Multi-Level GUI Comparison Criteria (Multi-Level GUICC)* for Android apps based on the empirical investigation of real-world apps.

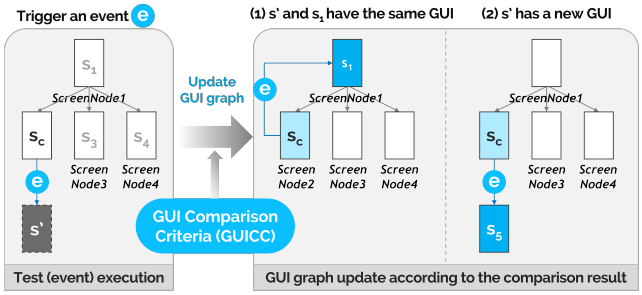


Figure 2: GUI graph update according to the comparison result

3.1.1 Investigation of Comparison Criteria

The multi-level GUI comparison technique was designed based on a semi-automated investigation with 93 real-world Android commercial apps registered in *Google Play*. The investigation was conducted as follows.

App selection. In order to obtain the characteristics of Android GUIs in commercial Android apps, we selected 93 real-world apps that were registered in *Google Play*. First, we collected the 20 most popular apps in 12 categories (total 240 apps), and then we excluded the apps in the <*Game*> category because most of them were not implemented as native apps. We selected 9 out of 12 categories, and we excluded apps that were downloaded fewer than 10,000 times. Finally, we chose 93 target apps: 13 apps in the <Book> category, 12 in <Business>, 10 in <Communication>, 9 in <Medical>, 10 in <Music>, 9 in <Shopping>, 7 in <Social>, 13 in <Transportation>, and 10 in <Weather>.

Manual exploration. After selecting the 93 apps, we investigated composed widgets of the apps by manual exploration. For the exploration, we visited 5-10 main screens of the target apps in an end user's view. Not just the functionality was exercised; the constituent widgets in GUIs were examined to extract GUI-related information. In order to obtain the information, we used the `UIAutomator` tool in Android devices, and we generated `uidump.xml` files to analyze the GUIs of the screens. By automatically parsing the `uidump.xml` files, we analyzed the structure of widgets and their properties, and values. In addition, the system information was extracted through `dumpsys system diagnostics`

Classification of GUI information. After extracting and collecting the GUI information from the selected apps, we classified the GUI information. Using the extracted GUI information, we could find hierarchical relationships among several types of GUI information. For example, a package includes multiple activities, and an activity includes multiple widget structures. At the same time, we filtered out redundant GUI information that highly depends on the device or the execution environment, such as coordinates or screenshot images. In addition, we merged some GUI information into a single property. For instance, event-related properties `<clickable>`, `<long-clickable>`, `<checkable>`, and `<scrollable>` were merged into an `<executable>` property.

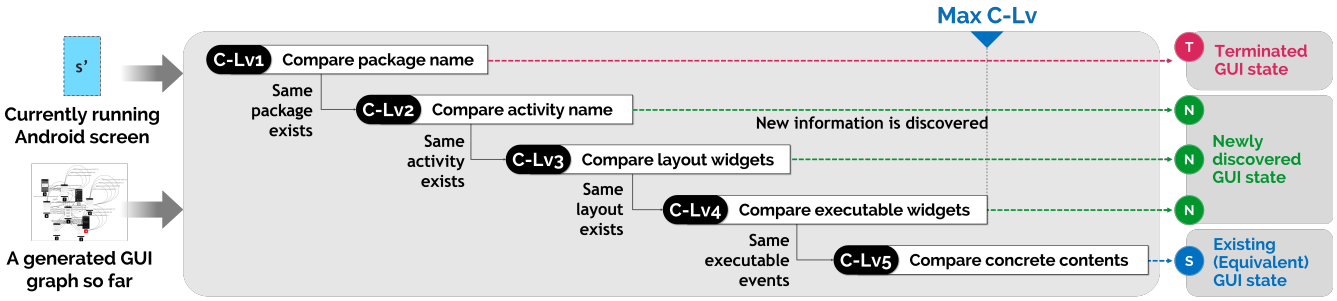


Figure 3: A GUI comparison model using *multi-level GUI comparison criteria (GUICC)* for Android apps

Definition of GUICC model. Based on the classification of extracted GUI information, we finally designed a *multi-level GUICC model* that contains hierarchical relationships among the information. The comparison model is shown in Figure 3. The comparison model has 5 comparison levels (**C-Lv**), and it has 3 types of outputs according to the comparison result: **T** (Terminated), **S** (Same), or **N** (New). The default comparison level is set to **C-Lv5**, but the maximum comparison level (**Max C-Lv**) can be flexibly modified if a tester wants to adjust the abstraction level of a GUI model. Finally, these *multi-level GUICC* were embedded in *GraphGenerator* of *TestingEngine* of our testing framework, and the details of the framework are explained in Section 4.

3.1.2 GUI Comparison Criteria for Android Apps

As **C-Lv** increases from **C-Lv1** to **C-Lv5**, more concrete (i.e., fine-grained) GUI information is used to distinguish GUI states. Following are the detailed comparison techniques for each **C-Lv**.

C-Lv1: Compare Package Names.
First of all, our comparison model compares the package name of the currently running screen to visited *ScreenNodes* of the GUI graph. In order to efficiently explore a behavior space (i.e., explore only the relevant space), a testing engine must be able to clearly distinguish the boundary of the space of an AUT. To avoid exploration outside of the app boundary, the testing engine analyzes which app package is currently being focused and run by the Android device. We used `dumpsys` again to dump the status of the Android system and extracted the package name of the currently focused app. By parsing `mFocusedApp` information analyzed by `dumpsys`, the testing engine identifies which package is running in the foreground. If the device does not focus on the package of the target AUT, the current status of the target app is considered as a terminated state **T** (i.e., an exit state). This exit case can be caused by transitions to a third-party app, shutdown, or crashes by an executed event.

C-Lv2: Compare Activity Names.
After the comparison in **C-Lv1** is passed, the comparison of activity names is performed. The activity name of the currently running screen is compared with the names of activities of other *ScreenNodes* in the GUI graph. If not all *ScreenNodes* in the GUI graph have the same activity name, the current screen is considered as a newly discovered GUI state. On the contrary, if one or more *ScreenNodes* that have the same activity name are discovered in the GUI graph, the next level of comparison (**C-Lv3**) is taken. Because each activity of Android apps is independently implemented in different source code files (e.g., `MainActivity.java` and `SecondActivity.java`) and the life cycle of each activity is managed individually, the activity name can be used to distinguish the physical difference in GUI states. Similar to the package name extraction in **C-Lv1**, the name of the currently running activity in the foreground can be obtained by parsing `mCurrentFocus` information from `dumpsys` system diagnostics.

C-Lv3, C-Lv4: Compare Widget Composition.
Even though the activity is the basic and simplest information to distinguish GUI states of Android, the activity name still could not distinguish detailed GUI states in real-world apps. In particular, modern Android apps show dynamic and complicated user interfaces in a single activity to provide better experience as well as good modularity. For example, many commercial apps utilize the `ViewPager` widget, which contains multiple different pages to show. If a testing tool cannot distinguish each page, then a generated GUI graph cannot exercise some events in abstracted pages. Therefore, it is imperative to compare GUI states with more detailed information at the widget level, because GUIs differ depending on the widget composition even though they are running on the same activity (see Figure 1-(a)). In other words, GUI graphs based on activities can be inadequate for the effective GUI model generation of real-world apps.

In order to compare two GUIs based on the composition of widgets, a widget hierarchy extracted by `UIAutomator`, a UI testing support tool of Android, is used. `UIAutomator` automatically analyzes a widget hierarchy from the screen of an Android device, and constructs a widget hierarchy tree, as illustrated in Figure 4. Every widget node has parents-children or sibling relationships with each other, and the relationships are encoded in an `<index>` property. `<index>` represents the order of child widget nodes; for example, if the value of an index of a certain widget w_i is 0, w_i is the first child of its parent widget node. In addition, by using index values, each widget (*node X*) can be specified as an index sequence that accumulates the indices from the root node to the target node. For instance, a widget *node M* in Figure 4 has its index sequence `[0,0,2,1]` that accumulates the indices from *node A* to *node M*.

By using these index sequences, our comparison model obtains the composition of specific types of widgets. Table 1 shows two types of widget composition extracted from the widget tree of Figure 4—composition of *layout widgets* and *executable widgets*. From the widget tree, we refer to non-leaf widget nodes as *layout widgets*, and leaf widget nodes whose event properties (e.g., `clickable`) have at least one "true" value as *executable widgets*. If a non-leaf widget node has an executable property (e.g., if a `ListView` is `clickable`, while its children are not executable), its child leaf

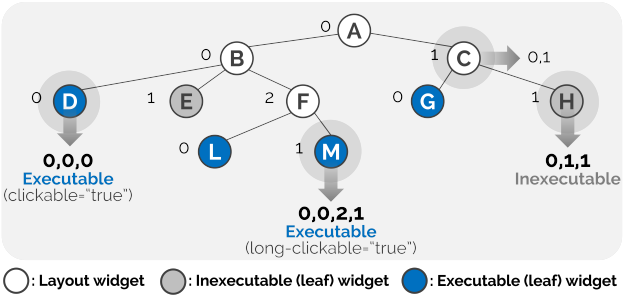


Figure 4: A widget hierarchy tree analyzed by *UIAutomator*

Table 1: Widget composition of a GUI of Figure 4

Widget type	Nodes	CIS
Layout	*A, B, C, F*	[0]-[0,0]-[0,1]-[0,0,2]
Executable	*D, G, L, M*	[0,0,0]-[0,1,0]-[0,0,2,0]-[0,0,2,1]

nodes are considered as *executable widgets*. In order to utilize the extracted index sequences as the widget information, we store *cumulative index sequences (CIS)* into newly added *ScreenNodes*. For the above example, two *CISs* are included in a *ScreenNode* to represent the composition of *layout widgets* and *executable widgets*.

At **C-Lv3**, the layout *CIS* is used as *GUICC* of *ScreenNodes*. If a screen has the same layout *CIS* as *CISs* of other visited *ScreenNodes* in a GUI graph, then the executable *CIS* is compared at **C-Lv4** in succession. Otherwise, the screen is regarded as a new *ScreenNode* because a screen that has different widget composition is likely to have a different GUI state. Comparison of executable widgets at **C-Lv4** is similar to the existing comparison technique that compares a set of event handlers between GUI states [8]. Since difference in composition of executable widgets indicates a different set of executable events, a target screen that has a different executable CIS must be regarded as a new GUI state.

C-Lv5: Compare Contents.

The last comparison step is performed to detect a GUI that can be distinguished only by the contents information, such as text. In the case that multiple screens have separated context even if there is no difference in their widget compositions of two screens, modeling GUIs for each screen separately is necessary to represent behaviors that can act differently in different contexts.

Using the `UIAutomator` again, the contents information is obtained by extracting contents values such as text (`<text>`) and description of contents (`<content-desc>`). For the abstract representation of the text values and fast comparison, we simply use the length of the textual information (i.e., if the value of `<text>` is "`MainActivity`" and the value of `<content-desc>` is "`Click here`", then we use $12+10=22$).

There is an additional need to distinguish GUIs before and after the execution of scroll events (e.g., `swipe-down`, `swipe-up`). **C-Lv5** also performs the comparison that examines the first item of `ListView` (`android.widget.ListView`) in GUIs. Because the first item of `ListView` is changed by scroll events if visible items of the `ListView` are changed after the scrolling, the comparison of the visible first list item is intuitively reasonable. By distinguishing a new GUI state after scroll events, our testing tool is able to obtain some possible events to be executed beneath the visible area of the device screen. This comparison step can be applied to all other list widgets, such as `GridView` (`android.widget.GridView`).

However, this comparison step (**C-Lv5**) does not always work as expected for every GUI in real-world apps. As explained in Chapter 1, comparison of the detailed information can lead to the state explosion problem during the exploration. If a tester runs GUI testing with **C-Lv5** for the calculator app shown in Figure 1-(b), the model generator will add new *ScreenNodes* infinitely because the text on the screens can be endlessly updated by user inputs. Therefore, a tester has to carefully select a proper comparison level to generate a GUI model efficiently. Also, GUI information used in **C-Lv5** can be altered to examine specific information according to the type of apps. For example, a gallery app can utilize image recognition or comparison between screens to compare contents in **C-Lv5**.

3.1.3 *Multi-Level GUI Graph Generation*

To summarize: the five levels of hierarchical GUI comparison criteria were implemented as a *multi-level GUI comparison model* of *ModelGenerator*. A tester (or testing tool), who performs model-based GUI testing from a black-box view, can adjust the level of testing thoroughness by simply adjusting the maximum level of *GUICC* (**Max C-Lv**). After the testing engine is configured with a given level, the *ModelGenerator* automatically compares GUI states based on the comparison criteria of the designated **C-Lv** while exploring the behavior space. As a result, the selection of a **C-Lv** not only impacts the behavior modeling by a GUI graph, but also determines the feasibility of GUI graph generation. Therefore, a tester should carefully consider an adequate criterion that is proper to the target AUT.

4. TESTING FRAMEWORK

4.1 Framework Overview

In this study, we mainly refer to the recent researches that describe a generic concept of model-based techniques in test input generation aspects [9]. Because most Android apps do not have their own GUI models beforehand, we use the term "*model-based Android GUI testing*" as a model-learning GUI testing technique that dynamically constructs a GUI model through reverse-engineering (i.e., *GUI Ripping* [4, 3]). These model-based techniques do not only perform testing, but also build GUI models while learning a model from runtime behaviors of an AUT at the same time.

The ultimate goal of model-based Android GUI testing is to detect runtime errors using systematically generated test inputs from a GUI model [6]. Keeping the basics in the model-based testing approaches, we develop an automated model-based Android GUI testing framework with *Multi-Level GUI Comparison Criteria (Multi-Level GUICC)* as illustrated in Figure 5.

This framework receives two inputs: (1) an installation file (`.apk` file) on the Android side, and (2) a selected level of *GUICC* (**Max C-Lv**) to configure the abstraction level of a GUI graph. After the testing engine sets a target AUT, it starts a reverse-engineering process from a black-box view to generate a GUI graph of the target app without source code. During the model generation process, the engine automatically explores the behavior space of an AUT by executing events sequentially on the Android device—this process is generally called *online test input generation* in [9, 14]. As

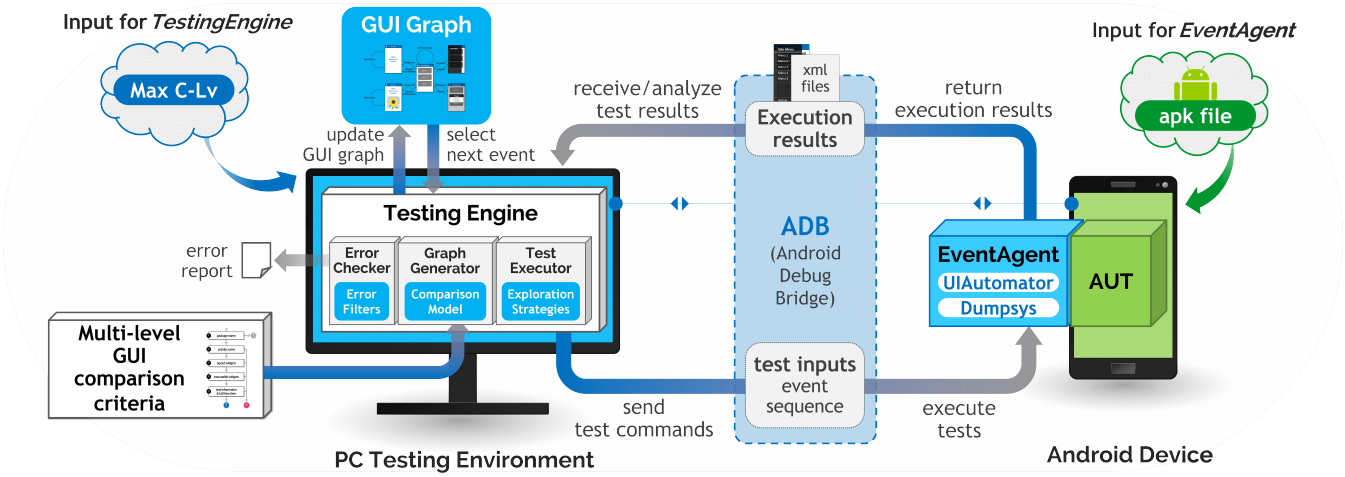


Figure 5: Model-based GUI testing framework for Android apps using *Multi-level GUICC*

outputs of testing, our testing engine finally produces (1) a set of event sequences that were executed on the AUT, (2) a GUI graph representing the explored behavior space and executed tests, and (3) an error report if the engine detects any error during the exploration.

4.2 Modules of Testing Framework

As Figure 5 displays, our testing framework consists of three modules: (1) *TestingEngine* running on a desktop PC, (2) *EventAgent* executed on an Android device, and (3) a *communication layer* between them. From Section 4.2.1 to 4.2.3, we provide detailed information about each module of the framework.

4.2.1 Communication Layer

First of all, to generate a GUI graph dynamically, a means of online communication between a desktop PC and an Android device must be constructed. Commands of test inputs (events) generated by *TestingEngine* are sent to the AUT running on the Android device through the communication layer. After *EventAgent* on the Android device receives the commands and executes them, *EventAgent* returns the execution results (i.e., GUI information of the current screen of the Android device), and the *TestingEngine* analyzes the results sent by the device. The *TestingEngine* also updates the GUI graph based on the execution results.

As shown in Figure 5, the `Android Debug Bridge (ADB)` is used to facilitate the communication between two different environments—a desktop and an Android device. `ADB`, a tool that supports command-line communication between a server (Android device or emulator) and a client (PC), is used to deliver both event commands and the execution result files. In addition, TCP socket communication is also used to send or receive (de)activation messages between them. Every time the *TestingEngine* generates a test input, *EventAgent* is activated by a wake-up message sent through a TCP socket. After the *EventAgent* completes the test execution, it generates several result files and returns them to the *TestingEngine* using `ADB` commands.

4.2.2 TestingEngine

Basically, by repeating ***(1) test input generation using a GUI graph*→*(2) transmission of test inputs to EventAgent*→*(3) event execution on AUT*→*(4) transmission of execution results to PC*→*(5) result analysis*→*(6) GUI graph update***, our *TestingEngine* automatically explores the behavior space of a running AUT. Details of submodules are explained below.

TestExecutor. The first submodule of *TestingEngine* is *TestExecutor*, which generates test inputs and sends command messages to the connected Android device to explore the behavior space. Based on the generated GUI graph at a certain time, *TestExecutor* manages a set of executable events extracted from the visited *ScreenNodes*, and generates event sequences as next test inputs. The current version of *TestExecutor* only supports encoding of the UI events such as `click`, `long-click`, `swipe`, and `text inputs`; it does not deal with system events.

The automated exploration algorithm is another crucial technique to efficiently generate a GUI graph. Thus, various kinds of exploration techniques have been researched [15, 2, 11], and they are usually called crawling algorithms. Our testing framework implemented a *Breadth-first-search (BFS)* algorithm using an event queue, called *working list*, for the GUI crawling in *TestExecutor*. The *working list*, which contains test inputs to be executed, determines the order of test selection and execution.

GraphGenerator. The *GraphGenerator* of our testing framework generates and manages a GUI graph of an AUT while the *TestExecutor* is exploring the behavior space. The main goal of the *GraphGenerator* is to update the GUI graph accurately with the received information (execution results) from the Android device. In other words, the *GraphGenerator* determines how to update the graph by analyzing the execution results, and it suggests the next test input to the *TestExecutor* based on the current GUI graph.

In order to obtain a set of executable events for test input generation, *TestingEngine* dynamically analyzes GUI states to extract executable events. Previous studies call the dynamic extraction of a set of executable (or fireable) events from GUIs <*action deduction*> or <*action inference*> [27]. In this study, we also call the process of extracting events from GUIs <*event inference*>, and `UIAutomator`'s API is used again to perform the event inference dynamically. From the `uidump.xml` file, the *TestingEngine* can get the structural information of GUI components on the screen of the Android device and also acquire the detailed properties and

values related to the events. `UIAutomator` extracts information from each widget node as follows.

```
<node bounds="[160,246][560,458]" selected="false"
     password="false" long-clickable="false"
    scrollable="false" focused="false" focusable="
    true" enabled="true" clickable="true" checked
    ="false" checkable="false" content-desc=""
    package="com.se.toyapp" class="android.widget.
    Button" resource-id="" text="A Button" index
    ="0"/>
```

By analyzing the values of properties related to user-events (`clickable`, `long-clickable`, `checkable`, `scrollable`), executable events can be inferred from the widgets shown in a GUI. In the case of the above example, the widget is a button named `<A Button>` that is located in `[160,246][560,458]` of the device screen; this button is `clickable` but neither `long-clickable` nor `scrollable`. The extracted property information is used as attributes to represent a GUI state of each *ScreenNode*, and the inferred events are stored in the *working list* of the *TestExecutor* to generate subsequent test inputs.

After every event execution, the *GraphGenerator* checks whether or not the current screen of the Android device is displaying a newly discovered GUI state using *Multi-Level GUICC*. According to the comparison result, the *GraphGenerator* updates its GUI graph by adding only an *EventEdge* or both an *EventEdge* and a *ScreenNode*. When adding a new *ScreenNode* to the graph, *GraphGenerator* stores the GUI information of the node to the disk with a screenshot image and its `uidump.xml` file. Moreover, if it receives an error report from the *ErrorChecker*, the *GraphGenerator* highlights corresponding *ScreenNodes* as problematic ones and creates an error log file. After all, if the *TestingEngine* finishes the testing process, a new GUI graph that stores all information about visited *ScreenNodes* and executed *EventEdges* is saved to an `xml` file by *GraphGenerator*.

ErrorChecker. While analyzing the result files given by the device, the *GraphGenerator* continually receives a log file made by Android `Logcat`. The *ErrorChecker* analyzes the file to detect suspicious execution traces or logs using *error filters*. Basically, the *ErrorChecker* is equipped with *error filters* that catch `<E/AndroidRuntime>, <D/CrashAnrDetector>, and <F/>` logs. In other words, the *ErrorChecker* detects runtime errors, ANR (Application Not Responding) reports, and fatal errors in the Android system. If the *ErrorChecker* finds erroneous logs after the event execution, the detected error is reported by a text file.

4.2.3 EventAgent

EventAgent is a tool that runs on an Android device to receive commands of test inputs from the *TestingEngine* (*TestExecutor*). The *EventAgent* was implemented as `UiAutomatorTestCase`; it transforms test commands into corresponding method calls. In order to wait to receive a test command, the *EventAgent* waits until a command message has come from *TestingEngine*. For the wake-up process of *EventAgent*, the TCP socket communication was used (See 4.2.1). After the *EventAgent* receives the message, it is translated to a sequence of corresponding method calls using APIs of `UiAutomatorTestCase`. To avoid the waste of the memory resource in an Android device, the *EventAgent* is only activated to receive and execute test inputs for a while, and it is automatically deactivated after the test execution.

5. EXPERIMENTS

5.1 Research Questions

The goal of our experiments is to observe the effects of the *GUI comparison criteria (GUICC)* on the testing effectiveness by performing actual automated model-based Android GUI testing. In our experiments, we mainly evaluated the results based on the generated GUI graphs according to the target level of *GUICC* (**Max C-Lv**) while keeping the test case generation algorithm. To this end, we investigated the following questions:

RQ1. How does the *GUICC* affect the behavior modeling of Android apps?

RQ2. Does the *GUICC* affect the achieved code coverage in model-based GUI testing?

RQ3. Does the *GUICC* affect the error detection ability in model-based GUI testing?

RQ1 is an elementary question to evaluate the relationship between the comparison level (**C-Lv**) and the behavior modeling by generated GUI graphs. By identifying the correlation between **C-Lv** and modeled GUI graphs, we answer the effect of the *GUICC* on behavior modeling of subject Android apps. Only for RQ1, we conducted the graph generation not only on the open source apps, but also on the real-world commercial apps to discover the practical application of our testing framework. In particular, we analyze the difference between activity-based GUI graphs (**C-Lv2**) and other GUI graphs with the higher level of *GUICC* to figure out the limitations of model-based testing techniques that use activity-based GUI models.

RQ2 investigates the code coverage achieved by the model-based GUI testing with our framework. Since the code coverage is the most frequently used metric to assess the effectiveness of testing [9, 16], we evaluated how much behavior is covered at a specific level of *GUICC* by measuring code coverage. We used the *Emma*[1] code coverage measurement tool, which is officially supported by the Android platform. For the measurement of the code coverage, the whole project files with the source code files of the target apps are required. Thus, we only measured the coverage of open-source apps to answer *RQ2*.

Lastly, for *RQ3*, we investigated the detected errors during the testing, since the error detection ability can vary with the *GUICC*. More specifically, we observed which level of the *GUICC* can detect runtime errors and failures with our *error filters* explained in 4.2.2. By analyzing the detected errors considering the **C-Lv**, we expect to point out the limitations of existing activity-based GUI models and figure out the importance of the selection of the *GUICC* in model-based GUI testing for Android apps.

5.2 Experimental Setup

5.2.1 Experimental Environment

We built a testing environment on both a PC and an Android device (or Genymotion emulator[2]), and the experimental environment is described in Table 2. *Communication layer* and every other module in our testing framework was implemented in Java—*TestingEngine* and *EventAgent*.

[1] *Emma Java code coverage measurement tool*: http://emma.sourceforge.net

[2] *Genymotion*: https://www.genymotion.com

Table 2: Experimental environment of the desktop PC and the Android device

Desktop	**OS**	Windows 7 Enterprise K SP1 (64-bit)
	Processor	Intel(R) Core(TM) i5 CPU 750@2.67GHz
	Memory	16.0GB
Android device	**Emulator**	Genymotion 2.5.0 with Oracle VirtualBox 4.3.12
	Android virtual device	Samsung Galaxy S4
		Android 4.3 (API 18)
		2 Processors
		3,000 MB Memory

5.2.2 Benchmark Applications

In order to evaluate the influence of the *GUICC* on the GUI graph generation and testing effectiveness, we needed a set of benchmark Android apps. We investigated several earlier papers [8, 27, 6, 16] that performed their experiments with open-source apps for the evaluation. Most studies used the distributed open-source Android apps provided by *F-Droid*[3], and Shauvik's study [9] combined them to evaluate the strengths and weaknesses of the existing techniques. Among the benchmark apps used in [9], we randomly selected 10 medium-sized (1,000-10,000 LOC) apps for our experiments. Also, we collected another 10 real-world commercial apps that were registered in *Google Play*. The collected open-source apps and their sizes are listed in Table 3, and commercial benchmark apps with their download (#Download) and recommendation (#Recomm.) counts in Table 4.

6. RESULTS

Once the testing framework is ready with the **Max C-Lv** and a given installation `apk` file of an application under test, a specific level of GUI graph is automatically generated[4]. By analyzing the generated GUI graphs and related result files, we answered our research questions. Note that, for assessing the influence of *GUICC*, the same test case generation algorithm was used independently of the chosen **C-Lv**.

6.1 RQ1. GUI Graph Generation by *GUICC*

In order to identify the effect of the *GUICC* on the GUI graph generation for *RQ1*, we manipulated the **Max C-Lv** from 2 to 5 for every benchmark app and generated multiple GUI graphs. After we generated the graphs, we compared the number of *ScreenNodes* (#SN) and *EventEdges* (#EE) among them. Since the first comparison level (**C-Lv1**; package name comparison) is only used to check the boundary of the behavior space, **C-Lv1** was only used for the detection of app termination or runtime errors and not included in the results. The other remaining four comparison levels (**C-Lv2**-**C-Lv5**) were used to analyze the effect of the *GUICC*, but we omitted the result graphs of **C-Lv3** due to lack of space. Table 5 and Table 6 show the size of the generated GUI graphs of open-source apps and commercial apps respectively.

Unsurprisingly, for every open-source benchmark app, the number of *ScreenNodes* and *EventEdges* tended to increase as the **C-Lv** increased. For some apps, a generated GUI graph with **C-Lv5** contained more than three times as many nodes and edges as a GUI graph with **C-Lv2**. Because the number of *EventEdges* (#EE) indicates the number of executed tests, graphs with more edges are more likely to have better ability to infer the test events. In addition, the results showed that the activity-based GUI graphs (**C-Lv2**) could not exercise a lot of behaviors. The *GUICC* based on the activity name was too abstract to model the behaviors adequately. On the other hand, the graphs using concrete *GUICC* could reflect more behaviors than the activity-based graphs. From the results, we found that the higher level of the fine-grained *GUICC* that we had defined were generally more effective to adequately model the behavior space than the lower level of the *GUICC*.

Additionally, we found some unusual and notable results during the graph generation. First, the graphs of *Sanity* (`cri.sanity`) showed a dramatic change between **C-Lv4** and **C-Lv5**. This result indicates that differentiating the *GUICC* can affect the ability of exploration of fully automated model-based GUI testing. Second, the generated GUI graphs of *DalvikExplorer* (`org.jessies.dalvikexplorer`) showed that the GUI graph with **C-Lv5** caused state explosion (S/E) due to the infinite exploration. Since *DalvikExplorer* has a continuously changing `TextView` that monitors the status of the *Dalvik virtual machine* in an Android device, the comparison of text contents produced an infinite number of GUI states while exploring the behavior space. We regarded the graph generation as a state explosion if it took more than three hours and no sign to finish the exploration. These unexpected results imply that an optimal or adequate *GUICC* can vary depending on the behaviors of an AUT, so *GUICC* must be carefully selected considering the target app. Furthermore, higher-level *GUICC* do not always perform better because the comparison of detailed information can be disruptive for efficient testing.

For the GUI graph generation of commercial benchmark apps, we discovered similar results to the experiment with open-source apps. The higher levels of *GUICC* usually made larger GUI graphs including more different GUI states and transitions (*ScreenNodes* and *EventEdges*) to represent the behavior space. For some commercial apps (*Advanced Task Killer*, *Alarm Clock Xtreme Free*), on the other hand, **C-**

Table 3: Open-source benchmark Android apps

No	Application package	LOC	# of classes
1	*org.jtb.alogcat*	1516	21
2	*com.example.anycut*	1094	6
3	*com.evancharlton.mileage*	4581	47
4	*cri.sanity*	8145	74
5	*org.jessies.dalvikexplorer*	2199	23
6	*i4nc4mp.myLock*	1352	16
7	*com.bwx.bequick*	6339	63
8	*com.nloko.android.syncmypix*	7178	38
9	*net.mandaria.tippytipper*	1880	13
10	*de.freewarepoint.whohasmystuff*	1126	8

Table 4: Commercial benchmark Android apps

No	Application name	#Download	#Recomm.
1	*Google Translate*	300,000,000	766709
2	*Advanced Task Killer*	75,000,000	141237
3	*Alarm Clock Xtreme Free*	30,000,000	128005
4	*GPS Status & Toolbox*	30,000,000	52867
5	*Music Folder Player Free*	3,000,000	8977
6	*Wifi Matic*	3,000,000	7081
7	*VNC Viewer*	3,000,000	4791
8	*Unified Remote*	3,000,000	23568
9	*Clipper*	750,000	3760
10	*Life Time Alarm Clock*	300,000	570

[3] *F-Droid*: https://f-droid.org/

[4] The generated graphs are uploaded to our website (https://sites.google.com/a/se.kaist.ac.kr/ymbaek-ase2016-gui-testing-android/)

Table 5: Result GUI graphs of *open-source* apps (*S/E stands for the State Explosion)

No	App name	C-Lv2		C-Lv4		C-Lv5	
		#SN	#EE	#SN	#EE	#SN	#EE
1	*Alogcat*	5	45	15	247	76	269
2	*Anycut*	8	33	8	33	9	42
3	*Mileage*	16	117	69	532	81	618
4	*Sanity*	1	4	2	7	49	552
5	*DalvikExplorer*	16	178	30	301	**S/E**	**S/E**
6	*MyLock*	2	24	5	51	10	101
7	*Bequick*	2	7	60	250	71	351
8	*Syncmypix*	4	11	20	96	20	115
9	*TippyTipper*	4	29	13	102	19	175
10	*WhoHasMyStuff*	7	37	24	143	26	180

Table 6: Result GUI graphs of *commercial* apps

No	App name	C-Lv2		C-Lv3 ∼ C-Lv5		
		#SN	#EE	C-Lv	#SN	#EE
1	*Google Translate*	5	41	***4***	***55***	***871***
2	*Advanced Task Killer*	4	27	***3***	***12***	***178***
3	*Alarm Clock Xtreme Free*	4	81	***4***	***23***	***363***
4	*GPS Status & Toolbox*	3	12	***5***	***49***	***592***
5	*Music Folder Player Free*	7	37	***5***	***30***	***295***
6	*Wifi Matic*	2	9	***5***	***20***	***160***
7	*VNC Viewer*	6	43	***5***	***7***	***60***
8	*Unified Remote*	6	94	***5***	***20***	***160***
9	*Clipper*	2	17	***5***	***36***	***195***
10	*Lifetime Alarm Clock*	2	5	***5***	***60***	***529***

Lv4 and **C-Lv5** led to state explosion due to the dynamically changing contents (e.g., the list of running processes, the current time).

The answer to *RQ1* can be summarized as follows: The results of behavior modeling in model-based GUI testing can be significantly influenced by the selection of the *GUICC*, but higher levels of the *GUICC* do not always mean more optimized solutions.

6.2 RQ2. Code Coverage by *GUICC*

We answered *RQ1* to investigate how *GUICC* affects the behavior modeling. However, a GUI graph with more *ScreenNodes* and *EventEdges* does not guarantee better coverage to explore the behaviors. If some *ScreenNodes* have different GUIs but the same set of executable events, the increasing number of *EventEdges* does not improve the testing effectiveness. Duplicated GUI states and redundant tests can even lower the testing efficiency because the testing tool unnecessarily repeats the execution of the same test inputs. In order to figure out how much the behavior implemented in source code is covered, we measured the code coverage. Since higher code coverage has a better potential to detect faults in the source code, the code coverage is widely used as a criterion to evaluate the testing effectiveness.

The four types of code coverage were measured by *Emma*, and we listed the results of method coverage and statement coverage in Table 7. The *Sanity* app, unfortunately, had to be excluded due to its internal errors while instrumenting the coverage measurement classes. The column labelled ***A*** shows measured coverage values that achieved by activity-based (**C-Lv2**) GUI graphs. Column ***M*** shows maximum coverage values that achieved by the graphs with the higher levels of *GUICC* (**C-Lv3**-**C-Lv5**). Column ***C-Lv*** shows the minimum comparison level (**C-Lv**) to achieve the maximum coverage ***M***. For example, if ***M*** is *80%* and ***C-Lv*** is *3*, it means that the lowest **C-Lv** that achieved the maximum coverage was **C-Lv3**, and the highest coverage achieved was *80%*.

Table 7: Achieved code coverage for *open-source* apps

No	Package name	Method coverage			Statement coverage		
		A	***C-Lv***	***M***	***A***	***C-Lv***	***M***
1	*Alogcat*	*46%*	***4***	***65%***	*39%*	***5***	***56%***
2	*Anycut*	*23%*	***4***	***69%***	*19%*	***4***	***55%***
3	*Mileage*	*22%*	***5***	***43%***	*18%*	***5***	***33%***
4	*Sanity*	*n/a*					
5	*DalvikExplorer*	*65%*	***4***	***70%***	*57%*	***4***	***64%***
6	*MyLock*	*11%*	***4***	***12%***	*10%*	***4***	***11%***
7	*Bequick*	*24%*	***5***	***39%***	*21%*	***5***	***39%***
8	*Syncmypix*	*10%*	***4***	***24%***	*6%*	***4***	***17%***
9	*TippyTipper*	*42%*	***5***	***65%***	*36%*	***5***	***61%***
10	*WhoHasMyStuff*	*39%*	***5***	***62%***	*35%*	***4***	***51%***
Average		**31%**	**50%**		**27%**	**43%**	

**A: Activity-based (C-Lv2), M: Maximum coverage*

The table shows that the level of *GUICC* could also affect the code coverage, such as the method and statement coverage. Also, increasing the **C-Lv** improved the achieved code coverage for all apps, and some apps (*Anycut*, *Syncmypix*) showed substantial difference between ***A*** and ***M***. In the table, the *Mileage*, *MyLock*, and *Syncmypix* apps showed relatively low coverage values. Here are the reasons for the disappointing results. *Mileage* contained a lot of behaviors that were dependent on the sophisticated text input, and this app also required a license file to access the core features. *MyLock* performed a lot of tasks through services running in the background of the Android system, but our testing framework does not support such service-based apps. Lastly, the main functionalities of *Syncmypix* were started only after a picture file was selected, but our framework did not support interoperability testing with 3rd-party apps.

The answer to *RQ2* can be summarized as follows: The achieved maximum code coverage was also affected by the level of *GUICC*, and existing activity-based GUI models could not cover a lot of behaviors in the source code.

6.3 RQ3. Error Detection Ability by *GUICC*

The ultimate goal of the automated GUI testing is to detect errors while exploring the behavior space of the target AUT by exercising test inputs [14]. During our experiments, the framework had detected four reproducible runtime errors in open-source benchmark apps; they are listed in Table 8. *Mileage* was crashed by fatal errors (`SIGSEV Fatal Signal` logged by `<F/libc>`), and *Sanity* was terminated by runtime exceptions (`NullPointerException` logged by `<E/AndroidRuntime>`).

As expected, the results show that the runtime errors were only revealed in the fine-grained GUI graphs with **C-Lv4** and **C-Lv5**, but the lower level of *GUICC* could not detect them. In other words, the ability to detect faults was also affected by the *GUICC*, and existing *GUICC* lower than **C-Lv4** including activity-based graphs could not detect the listed runtime errors we found. This error detection result shows that determining the level of behavior abstraction affects not only the achieved code coverage, but also the error detection ability. Also, an improperly selected coverage criterion lowers the behavior space to be explored, and the ability to detect errors depends on the model as a result.

Table 8: Detected runtime errors by our testing framework

No	C-Lv	App name	Error type
1	C-Lv5	*Mileage*	Fatal error (Fatal signal 11 <F/libc>)
2		*Sanity*	Fatal exception (RuntimeException <E/AndroidRuntime>)
3		*Sanity*	Fatal exception (RuntimeException <E/AndroidRuntime>)
4	**C-Lv4**	*Mileage*	Fatal error (Fatal signal 11 <F/libc>)

7. RELATED WORK

Model-based Android GUI testing. Modeling the GUIs of Android apps can be challenging because the precision of an approximation technique that abstracts the program flow determines the completeness of a testing framework [23]. In order to tackle the modeling issues, a great deal of model-based GUI testing techniques have been actively researched [19, 25]. They commonly generate finite state machine-based GUI models to generate test inputs [20]. Among various approaches, the most well-known reverse-engineering techniques for GUI testing are *GUI crawling* and *GUI ripping* [19, 3]. A testing tool developed by D. Amalfitano [3], called *AndroidRipper*, realized and utilized those model-based techniques for Android apps, and it became the root of our work.

Existing model-based GUI testing techniques [27, 24, 4, 6] focused on the functional correctness [26], and they have struggled to improve the testing effectiveness. For example, in the study by R. Mahmood et al. [17], they developed a testing framework to generate effective test cases using GUI-based models (*call-graph model*) utilizing a white-box analysis. For another example, W. Yang et al. improved the accuracy of the GUI model generation using a gray-box approach that performs the static analysis of source code to assist both GUI crawling and GUI model generation [27]. Also, several studies use symbolic execution [21] and concolic testing [5] for model-based testing. Although they improved the fault/error detection ability and revealed errors that other tools could not discover, those frameworks require relatively complicated data or analysis techniques, such as static analysis using the complete source code. Meanwhile, our testing framework performs fully automated testing with no additional data to be analyzed; only an `apk` file is required. In order to address the complexity of white-box or gray-box approaches, some crawling-based black-box GUI testing approaches such as *MobiGUITAR* [4] have paved the way for efficient model-based Android GUI testing.

GUI comparison criteria. In this study, we considered the *GUICC* as an essential requisite for effective model-based GUI testing. As mentioned in previous sections, existing model-based GUI testing techniques (or tools) have their own *GUICC*, even though the *GUICC* are not explicitly specified. Among the available tools, the activity-based *GUICC* have frequently been used to distinguish GUI states [6, 22]. However, as our experimental results implied in Section 6, the activity-based modeling can lead to a low testing effectiveness (i.e., the code coverage or the fault detection ability) since the majority of behaviors can be missed due to the complicated GUIs in real-world Android apps. The recent work by Azim and Neamtiu [6], for example, modeled Android apps into activity transition graphs, which are almost the same as our GUI graphs with **C-Lv2**. However, the GUI graphs with **C-Lv2** turned out to show lower effectiveness than other graphs using a more adequate *GUICC* through our evaluation. On the contrary, *GUICC* of some other techniques are too sensitive to efficiently cope with dynamically changing GUIs of real-world apps. Those volatile GUIs can cause a state explosion problem during the model-based testing, but they only provide fixed single-level criteria (e.g., every observable change in GUIs) [8, 27, 4]. Since an adequate *GUICC* of one app is not always the best one for other apps, providing a flexible configuration of *GUICC* is needed. In that sense, our model-based testing technique could improve testing efficiency, flexibility, and practicality as well as effectiveness.

8. CONCLUSION

Most existing techniques for model-based Android GUI testing have not pointed out the limitations for practical application. They have overlooked dynamic and volatile behaviors of real-world Android apps that make GUI model generation challenging or infeasible. However, the influence of the behavior abstraction has not previously been evaluated. In this study, we focused on GUI model generation, which is a key feature of model-based testing, and we introduced an automated testing framework with *multi-level GUI comparison criteria (Multi-Level GUICC)* for Android apps. Based on our proposed approach, we conducted empirical experiments to identify the influence of *GUICC* on testing effectiveness of model-based GUI testing for Android apps.

As a result of the experiments that performed automated model-based GUI testing with proposed *multi-level GUICC*, we observed the visible influences of *GUICC* on generated GUI models, achieved code coverage, and error detection. Model-based testing with *multi-level GUICC* could generate more effective GUI models than activity-based GUI models in terms of code coverage and error detection ability. Moreover, our testing framework with *multi-level GUICC* could support more flexible model-based GUI testing by manipulating comparison levels. The results will shed light on the behavior modeling of automated model-based GUI testing for both open-source and real-world Android apps.

In our future research, we expect to develop an automatic *GUICC* selection technique for automated model-based GUI testing that analyzes the inherent characteristics of the target app and selects adequate criteria. We have been investigating specific behaviors and widget compositions of real-world commercial Android apps, and this information is expected to be utilized in the *GUICC* selection technique. As a short-term goal of our research, we have a plan to open our testing framework to the public as soon as possible to share our knowledge and to contribute to mobile app testing researchers and practitioners.

9. ACKNOWLEDGMENTS

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIP) (No.R0126-16-1101, (SW Star Lab) Software R&D for Model-based Analysis and Verification of Higher-order Large Complex System). The authors would like to thank Gwangui Hong, Dongwon Seo, and Cheolwoo Chae for building a foundation for this research.

10. REFERENCES

[1] D. Amalfitano, A. R. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261, March 2011.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and G. Imparato. A toolset for gui testing of android applications. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 650–653, Sept 2012.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. Using gui ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 258–261, New York, NY, USA, 2012. ACM.

[4] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, Sept 2015.

[5] 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. ACM.

[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. *SIGPLAN Not.*, 48(10):641–660, Oct. 2013.

[7] G. Bae, G. Rothermel, and D.-H. Bae. Comparing model-based and dynamic event-extraction based {GUI} testing techniques: An empirical study. *Journal of Systems and Software*, 97:15 – 46, 2014.

[8] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. *SIGPLAN Not.*, 48(10):623–640, Oct. 2013.

[9] S. R. Choudhary, A. Gorla, and A. Orso. Automated test input generation for android: Are we there yet? (e). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 429–440, Nov 2015.

[10] GoogleDevelopers. Ui/application exerciser monkey (http://developer.android.com/intl/ko/tools/help/monkey.html), 2015.

[11] F. Gross, G. Fraser, and A. Zeller. Exsyst: Search-based gui testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 1423–1426, Piscataway, NJ, USA, 2012. IEEE Press.

[12] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. 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, pages 204–217, New York, NY, USA, 2014. ACM.

[13] C. Hu and I. Neamtiu. Automating gui testing for android applications. In *Proceedings of the 6th International Workshop on Automation of Software Test*, AST '11, pages 77–83, New York, NY, USA, 2011. ACM.

[14] A. Jaaskelainen, M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, T. Takala, and H. Virtanen. Automatic gui test generation for smartphone applications - an evaluation. In *Software Engineering - Companion Volume, 2009. ICSE-Companion 2009. 31st International Conference on*, pages 112–122, May 2009.

[15] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 67–77, New York, NY, USA, 2013. ACM.

[16] A. Machiry, R. Tahiliani, and M. 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, pages 224–234, New York, NY, USA, 2013. ACM.

[17] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou. A whitebox approach for automated security testing of android applications on the cloud. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, pages 22–28, Piscataway, NJ, USA, 2012. IEEE Press.

[18] A. M. Memon. An event-flow model of gui-based applications for testing: Research articles. *Softw. Test. Verif. Reliab.*, 17(3):137–157, Sept. 2007.

[19] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *Proceedings of The 10th Working Conference on Reverse Engineering*, Nov. 2003.

[20] A. M. Memon, M. L. Soffa, and M. E. Pollack. Coverage criteria for gui testing. *SIGSOFT Softw. Eng. Notes*, 26(5):256–267, Sept. 2001.

[21] N. Mirzaei, S. Malek, C. S. Păsăreanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *SIGSOFT Softw. Eng. Notes*, 37(6):1–5, Nov. 2012.

[22] E. Nijkamp. Supermonkey (https://github.com/testobject/supermonkey), 2014.

[23] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, CODASPY '13, pages 209–220, New York, NY, USA, 2013. ACM.

[24] Q. Xie and A. M. Memon. Model-based testing of community-driven open-source gui applications. In *Proceedings of the 22Nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 145–154, Washington, DC, USA, 2006. IEEE Computer Society.

[25] Q. Xie and A. M. Memon. Using a pilot study to derive a gui model for automated testing. *ACM Trans. Softw. Eng. Methodol.*, 18(2):7:1–7:35, Nov. 2008.

[26] S. Yang, D. Yan, and A. Rountev. Testing for poor responsiveness in android applications. In *Engineering of Mobile-Enabled Systems (MOBS), 2013 1st International Workshop on the*, pages 1–6, May 2013.

[27] W. Yang, M. R. Prasad, and T. Xie. *Fundamental Approaches to Software Engineering: 16th International Conference, FASE 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, chapter A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications, pages 250–265. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.