A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications

Wei Yang[1,2⋆], Mukul R. Prasad[1], and Tao Xie[2⋆⋆]

[1] Software Systems Innovation Group, Fujitsu Labs. of America, Sunnyvale, CA
mukul.prasad@us.fujitsu.com
[2] Department of Computer Science, North Carolina State University, Raleigh, NC
wei.yang@ncsu.edu, xie@csc.ncsu.edu

Abstract. As the mobile platform continues to pervade all aspects of human activity, and mobile applications, or mobile apps for short, on this platform tend to be faulty just like other types of software, there is a growing need for automated testing techniques for mobile apps. Model-based testing is a popular and important testing approach that operates on a model of an app's behavior. However, such a model is often not available or of insufficient quality. To address this issue, we present a novel grey-box approach for automatically extracting a model of a given mobile app. In our approach, static analysis extracts the set of events supported by the Graphical User Interface (GUI) of the app. Then dynamic crawling reverse-engineers a model of the app, by systematically exercising these events on the running app. We also present a tool implementing this approach for the Android platform. Our empirical evaluation of this tool on several Android apps demonstrates that it can efficiently extract compact yet reasonably comprehensive models of high quality for such apps.

1 Introduction

The mobile platform is projected to overtake the desktop platform as the global Internet platform of choice in the very near future [1]. There has been a decisive shift to mobile devices in numerous application areas such as email, social networking, entertainment, and e-commerce [1, 2]. This trend has prompted an explosive growth in the number and variety of mobile apps being developed. As of June 2012, Android's Google Play had over 600,000 apps that had been downloaded more than 10 billion times in total [3]! Users typically have a choice between several apps with similar functionality. Thus developers are required to develop high quality apps in order to be competitive. On the other hand, mobile apps are usually developed in relatively small-scale projects, which may not be able to support extensive and expensive manual testing. Thus, it is particularly important to develop automated testing tools for mobile apps.

⋆ This author was an intern at Fujitsu Labs. of America for the duration of this work.

⋆⋆ This author was supported in part by NSF grants CCF-0845272, CCF-0915400, CNF-0958235, CNS-1160603, and an NSA Science of Security Lablet Grant.

For the purpose of this research, we use Android apps as a representative of mobile apps in general. Most tools and frameworks [4–8] currently available for testing Android apps are simply aids for (manual) test-case authoring, deployment, debugging, and visualization. There are no effective industrial products for automated test-case *generation* per se. Recognizing this inadequacy, researchers have very recently begun to develop such techniques [9–13]. This paper attempts to build on this fairly nascent body of research.

Mobile apps are a subset of the more general class of event-driven applications and specifically event-driven Graphical User Interface (GUI) applications. However, they have the following characteristics that make them suitable for specific automated testing techniques.

- **Small size.** Mobile apps are typically much smaller and simpler than desktop applications, both in terms of the physical footprint as well as behavior. Desktop applications can be large, feature-rich, and computationally intensive. However, a significant fraction of mobile apps are designed as "micro-apps" to solve small and specific tasks [14]. Furthermore, the size of mobile apps is constrained by the limited processing, storage, and display resources of the mobile device. The small size of mobile apps enables automatic testing techniques to be feasible and applicable to real-world apps.
- **Event-centric.** Mobile devices have evolved to be small-screen devices without a keyboard. Since typing is onerous on such devices, mobile apps are designed around a rich set of user gestures as input events. Thus, on the one hand, the role of typed data is somewhat diminished in mobile apps in contrast to desktop applications. On the other hand, the richer set of user gestures in mobile apps needs to be incorporated into any testing process.
- **Simple & Intuitive GUI.** Users of desktop GUI applications might be expected to refer to documentation or tutorials to fully comprehend how to use the applications. In contrast, mobile apps are expected to have a simple and intuitive user interface where most, if not all, usage scenarios of an app should be evident to average users, from the GUI.

Model-based testing [15] is a popular and important type of testing that uses a model of the application under test as a basis for constructing test cases. Automated model-generation techniques that dynamically analyze the GUI of the application have been previously developed for desktop GUI applications [16] and for AJAX web applications [17]. However, the limited degree of automation of these tools and the incompleteness of the resulting models have posed barriers for their industrial adoption. Such limitations can be attributed, in part, to the nature of their target application domains. For example, the GUIs of feature-rich desktop applications or web applications can have a large, potentially unlimited, number of states. Thus, techniques such as Crawljax [17] either bound their exploration or require user-specified state abstractions to extract a finite model. Techniques such as GUITAR [16], on the other hand, resort to more imprecise event-based models. By contrast, as observed above, mobile apps have substantially smaller and simpler GUIs. This characteristic raises the possibility of more complete and automated GUI state-space exploration in mobile apps. Second, automated crawling techniques typically require knowledge of the set of GUI widgets supporting actions (e.g, clicks) and precisely what actions are supported on each such widget. For web applications, much of this information is represented in client-side JavaScript code, which is notoriously difficult to analyze. Thus, this information needs to be manually specified. For desktop GUI applications, this analysis is not that important since user actions are mostly simple mouse clicks. However, as noted above, supporting a rich array of user gestures is an integral part of mobile-app design. Further, as we demonstrate in this work, mobile app development frameworks are quite amenable to automatic analysis and extraction of this information. The objective of this work is to build a novel, customized and more efficacious automated GUI-model generator for mobile apps, particularly Andriod apps, by exploiting these observations.

Our approach uses static analysis of the app source code to extract the actions supported by the GUI of the app. This information is typically not available to a purely black-box analysis and is far more expensive to extract through a dynamic white-box approach [9]. Next, we use dynamic crawling to build a model of the app by systematically exercising the extracted events on the live app. We concur with the view of previous work [16, 17] that a dynamic analysis is far simpler and more precise than static analysis for analyzing GUIs. However, we exploit the smaller, simpler, and highly event-centric interface of mobile apps to build a more efficient and automated crawler.

Specifically, this paper makes the following main contributions:

– A dynamic, grey-box GUI reverse-engineering approach for mobile apps, which we identify as a specialized type of event-driven GUI apps.
– A novel static analysis to support the dynamic GUI crawling.
– A tool implementing this grey-box approach of automated model extraction for Android apps.
– An evaluation of this tool on several real-world Android apps for demonstrating its efficacy at generating high-quality GUI models.

2 Background & Problem Definition

Model-based testing [15] is an approach for software testing orchestrated around a model of the application under test. The model is typically an abstract representation of the application behavior and may be constructed either manually [18] or using automatic techniques [16]. This model is used to construct a suite of test cases to test the application. Various techniques of model-based testing have been proposed in the literature [19, 20].

One of the crucial steps in model-based testing is the creation of the model itself. When performed manually, it is usually a laborious and error-prone process. There is a body of work [11, 16, 17] that tries to partially or completely automate the process of extracting models from GUI applications. The general approach is to automatically and systematically interact with the GUI of the live, running application, in an attempt to extract and record a model of the usage scenarios supported by it. GUI applications are a subset of general event-driven applications and include types of applications such as web applications and desktop GUI applications as well as mobile apps.

As discussed in Section 1, mobile apps have special characteristics that distinguish them from other types of event-driven applications. This paper addresses the problem of automated GUI-model generation for mobile apps.

Problem Definition. *Given a mobile app, efficiently generate a high-quality model representing the valid input event sequences accepted by the app, where quality is measured by the following criteria:*

1. ***Coverage.*** *Every reachable program statement of the app should be executed by running at least one of the event sequences included in the model.*
2. ***Precision.*** *The model should not include invalid events, i.e., events that are not supported by widgets on a given screen.*
3. ***Compactness.*** *The size of the model, in relation to the number of event sequences that it represents, should be as small as possible.*

Note that the above problem definition uses statement coverage as the coverage criterion. However, the approach presented here would be equally applicable to any other suitable code coverage criteria.

3 Related Work

Automated model extraction. Our work falls under the broad category of automated model-generation techniques. The GUITAR [16] tool by Memon et al. is one of the earliest and most prominent representatives of this category. GUITAR reverse-engineers a model of a GUI application directly from the executing GUI. A recent extension of the tool, Android-GUITAR [21] supports Android apps. GUITAR uses formalisms of GUI forests and event-flow graphs to represent the structure and execution behavior of the GUI, respectively. However, the event-flow graph representation typically includes many false event sequences, which may need to be weeded out later.

The Crawljax [17] tool by Mesbah et al. is an automatic model extractor targeted to AJAX web applications. In contrast to GUITAR, it uses a state-machine representation to capture the model because of the stateful nature of AJAX user-interfaces. However, AJAX applications present particularly challenging targets for automatic model extraction because of their large (sometimes unbounded) state space. Therefore, in practice, manually specified state abstractions are required to extract a model with high coverage but manageable size. WebMate [22] is another, more recent, model extractor for web applications. The iCRAWLER [13] tool by Joorbachi et al. is a reverse-engineering tool for iOS mobile apps and such tool also uses a state-machine model. The emphasis there is on dealing with the idiosyncrasies of the iOS platform. All of the above tools have no means of deducing actionable GUI elements and supported actions on each screen. This information typically needs to be supplied to the tools. Some tools, such as Android-GUITAR, exercise only the default *tap* action on widgets. However, doing so provides less than optimal coverage of the behavior. Our proposed approach is unique in that it uses an efficient static analysis to automate and solve this aspect of model discovery for the Android platform.

Automated testing of mobile apps. Hu and Neamtiu [23] propose an approach that exercises the app under pseudo-random event sequences produced by the Android *Monkey* tool and analyzes the log files of this execution for certain kinds of faults. The AndroidRipper [11] tool also performs stress testing of an Android app but by systematically crawling its GUI. These approaches can sometimes reveal unexpected and interesting faults. However, their objective is to stress-test the app rather than to create a reusable model for use in future testing, as in our case. Takala et al. [18] present a case study of applying model-based testing for testing Android apps. The M[agi]C [10] tool is used to generate test cases for apps using a combination of model-based testing and combinatorial testing. Previous approaches [10, 18] work off a GUI model of the app and such model could potentially be generated using our proposed approach. More recently, Anand et al. [9] have applied concolic execution to generate feasible event sequences for Android apps. However, the computation-intensive nature of symbolic analysis coupled with an explosion in the sheer number of event sequences being enumerated limits their approach to fairly short event sequences. Our approach, by contrast, can efficiently exercise fairly deep event sequences. Mirzaei et al. [12] use static analysis to deduce the set of feasible event sequences and represent them using a context-free grammar (CFG). The deduced event sequences are then analyzed through symbolic execution. Their proposed static analysis is conceptually a generalization of our proposed action-inference analysis. However, the lack of algorithmic details and limited evaluation there makes a direct comparison with our approach difficult.

4 A Motivating Example

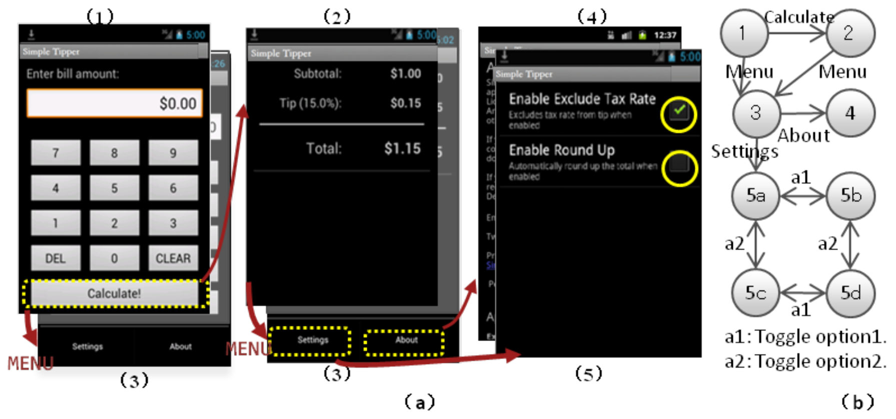


Fig. 1. Overview of SimpleTipper(a) and its state graph(b)

We use an Android app called *SimpleTipper* as an example to illustrate our approach. SimpleTipper is a simplified version of the open-source app, TippyTipper (http://code.google.com/p/tippytipper/), used to calculate the tip amount for a meal. Figure 1 illustrates its functionality. It consists of five screens. On the opening (*Input*) screen, the user enters the meal bill amount through a numeric keypad. The `DEL` button erases one digit. The `CLEAR` button or a `longClick` on `DEL` clears the textfield. Clicking the `Calculate` button takes the user to the second (*Result*) screen, which shows the total cost including the calculated tip. The third screen is the *Menu* screen. It is opened by clicking the `Menu` button on either the *Input* or the *Result* screen. The `About` option on the menu leads to the fourth screen, *About*, with information about the app. The `Settings` option on the menu directs the user to the fifth screen, *Settings* with two setting options. Checking either of them on or off influences the tip calculation.

5 Proposed Approach

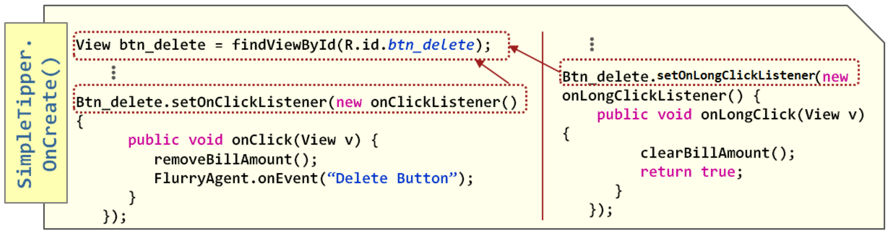
We propose a grey-box approach for automatically extracting a model of a given mobile app. First, we use static analysis of the app's source code to extract the set of user actions supported by each widget in the GUI. Next, a dynamic crawler is used to reverse-engineer a model of the app, by systematically exercising extracted actions on the live app. Our model has been designed to provide sufficient state abstraction for compactness, without unduly compromising its precision. The following sections describe these elements of our approach.

5.1 Action Inference Using Static Analysis

As explained earlier, supporting a wide array of user gestures is an integral aspect of mobile app design. A model representing only the default *click* action would miss a significant portion of the app's behavior. For example, in Figure 1(a), the `longClick` behavior of the `DEL` button on screen (1) would be omitted. Further, the `Settings` and `About` screens of the app cannot be accessed without the `Menu` button. These states constitute much of the app's state space, as shown in Figure 1(b). On the other hand, simply firing all possible actions on each widget would bring in invalid actions into the model and lower its precision. Thus, knowledge of the precise set of GUI actions is essential to generating a high-quality model.

Our approach uses static analysis to infer these actions. We make the observation that in the Android framework a user action is defined by either (a) registering an appropriate event listener for it or, (b) by inheriting the event-handling method of an Android-framework component. We term the former as *registered action* and the latter *inherited action*. For both these categories, identifying an action involves three basic steps: (1) identify the place where an action is instantiated or registered; (2) locate the component on which the action would be fired; (3) extract an identifier of the component that the crawler can later use to recognize the corresponding object and fire the action.

Algorithm 1 presents the analysis to detect registered actions. It essentially iterates over all program-entry points (*EntryPoints*) and all actions (*ActionSet*) supported by the mobile framework (Lines 6-16). For each entry point $\mathcal{P}$ and action $\mathcal{X}$, it extracts the call graph of the app (Line 5) and locates a set of



```
View btn_delete = findViewById(R.id.btn_delete);
   ⋮
Btn_delete.setOnClickListener(new onClickListener()
{
      public void onClick(View v) {
         removeBillAmount();
         FlurryAgent.onEvent("Delete Button");
      }
   });
```

```
   ⋮
Btn_delete.setOnLongClickListener(new
onLongClickListener() {
   public void onLongClick(View v)
{
        clearBillAmount();
        return true;
     }
  });
```

Fig. 2. An illustration of using static analysis for action inference

Algorithm 1: registeredActionDetection

```
   Input  : $\mathcal{A}$: app source code
   Output: $\mathcal{E}$: action map
1  begin
2      $ActionSet \leftarrow getAllActions()$
3      $EntryPoints \leftarrow getAllEntryPoints()$
4      foreach $\mathcal{P} \in EntryPoints$ do
5          $CG \leftarrow makeCallGraph(\mathcal{A}, \mathcal{P})$
6          foreach $\mathcal{X} \in ActionSet$ do
7              $\mathcal{L} \leftarrow getEventRegMethod(\mathcal{X})$
8              $PNodeSet \leftarrow getParentNode(CG, \mathcal{L})$ // Get all $\mathcal{L}$'s callers
9              foreach $PNode \in PNodeSet$ do
10                 $s \leftarrow findCallTo(PNode, \mathcal{L})$
11                 $v \leftarrow getCallingObject(s)$
12                 $i \leftarrow backLocate(v, \mathcal{A})$
13                 $ID \leftarrow getParameter(i)$
14                 $\mathcal{E}.add(ID, \mathcal{X})$
15             end
16         end
17     end
18 end
```

statements *PNodeSet* (Line 8) containing instances of a valid event-listener registering statement $\mathcal{L}$ for action $\mathcal{X}$. Finally, for each statement *PNode* in *PNodeSet* it performs a backward slice on *PNode* to locate an initialization statement of the widget on which the instance of $\mathcal{L}$ was called (Lines 10-12). The backward slice is used to get an identifier *ID* of the component (Line 13) that is registered in the action map $\mathcal{E}$ with the action $\mathcal{X}$. Figure 2 shows a code snippet where the developer defines `click` as well as `longClick` actions on the button `DEL` shown on screen (1) in Figure 1(a). To identify components on which to fire *longClick*, we first use the call graph to find the methods where *setOnLongClickListener* is called. It happens to be called in method *onCreate* of activity *SimpleTipper*. Then we locate the statement calling *setOnLongClickListener* in *onCreate* and get object *btn_delete* that the listener is registered to. Finally we backslice to get the initialization statement of *btn_delete*, get its ID `btn_delete`, and add the ID-action pair to the action mapping used by the crawler. Thus, when the crawler encounters a screen with component `btn_delete`, it fires a `longClick` on it.

Algorithm 2 describes the inherited action detection. We first get class hierarchy *CH* of the whole app (Line 3). Then, we use app's namespace to filter non-user-defined classes (Line 4). For each of the user-defined classes, if the class

Algorithm 2: inheritedActionDetection

```
   Input  : $\mathcal{A}$: app source code
   Output: $\mathcal{E}$: action map
1  begin
2      $ActionSet \leftarrow getAllActions()$
3      $CH \leftarrow getClassHierarchy(\mathcal{A})$
4      $Klass \leftarrow getUserClass(CH)$ // Get user defined classes
5      foreach $Class \in Klass$ do
6          foreach $\mathcal{X} \in ActionSet$ do
7              $\mathcal{L} \leftarrow getActionHandlMethod(\mathcal{X})$
8              $\mathcal{M} \leftarrow getDeclaredMethod(Class, \mathcal{L})$
9              if $\mathcal{L} \in \mathcal{M}$ then
10                 $ID \leftarrow getNameOrID(\mathcal{L}, \mathcal{M})$
11                 $\mathcal{E}.add(ID, \mathcal{X})$
12             end
13         end
14     end
15 end
```

overrides the action handling method $\mathcal{L}$ (Line 8), we regard the action $\mathcal{X}$ as valid, then we extract the Activity name or registered ID of the class (Line 10), and add the ID-action pair in the action mapping (Line 11).

5.2 Model Definition

We model the GUI behavior of an Android app as a finite-state machine. As noted by others [16, 17], GUI apps in general could have a large, potentially infinite number of UI states. However, our aim is to exploit the simple and intuitive GUI design of mobile apps to derive a compact yet high-quality model.

The model design is inspired by the UI-design principles espoused by the Android team. The Android User Experience Team [4] suggests that developers should “make places in the app look distinct” to give users confidence that they know their way around the app. In other words, different screens of the app should and typically do have stark *structural* differences not just minor stylistic ones. In addition, we would like to capture and reflect important differences such as a button being enabled or disabled. Such differences are reflected in the attributes of GUI components that support user actions. Finally, to keep the model compact, we ignore differences in the UI state resulting from different data values input by the user.

We use these principles to define a UI state, which we term as a *visual observable state*. Our model is a finite-state machine over these states with the user actions constituting the transitions between these states. The structure of a GUI screen in Android is represented by a tree of different GUI components, called a *hierarchy tree*. Further, we classify GUI components as *executable components* and *display components*. The former support user actions (which are detected by our static analysis) while the latter are just for display purposes. Thus, a visual observable state in our model is composed of the hierarchy tree of the UI screen, as well as a vector of attribute values of each of the executable components. The chosen attributes are ones that result in an observable change to the GUI component but excluding ones bearing user-supplied text values or values derived from them. It is fairly easy to manually identify the relevant attributes for each type of UI component, once, for all apps.

Figure 1(b) shows the state model of SimpleTipper. Each of screens (1) and (2) correspond to a unique state. Note that different values of the bill amount, input by the user in screen (1) do not give rise to different states. Further the pop-up dialog box launched by hitting the `Menu` button corresponds to state (3), irrespective of whether it is launched from states (1) or (2). For screen (5), the `Settings` screen, the two checkboxes are executable components. Their state changes give rise to the four different states 5a-5d for the app.

5.3 Crawling Algorithm

The objective of the crawling algorithm is to exhaustively explore all the app's states by firing *open actions*, *i.e.,* actions that have previously not been exercised by the crawler, on each observed state. The crawling process ends if the model has no *open states*, *i.e.,* states that have open actions to be fired. This process can potentially be done through a simple depth-first search (DFS) on the UI states. However, the key challenge here is the backtracking step, *i.e.,* undoing the most recent action done by DFS, on reaching a previously seen state. Crawljax [17] solves this issue, in the case of web applications, by re-loading the initial state and replaying all but the last action leading up to the current state. This strategy is possible in our case too, but can be fairly expensive, as shown in our evaluation. We refer to this strategy as standard DFS in the sequel.

Algorithm 3: crawlapp

```
Input  : $\mathcal{A}$: app under test, $\mathcal{E}$: action map
Output: $\mathcal{M}$: crawled model
1 begin
2     $\mathcal{M} \leftarrow \emptyset$; $s \leftarrow getOpeningScreen(\mathcal{A})$
3     while $s \neq null$ do
4         $s \leftarrow forwardCrawlFromState(s, \mathcal{A}, \mathcal{M}, \mathcal{E})$ // forward crawl from $s$
5         $s \leftarrow backtrack(s, \mathcal{A})$
6         if $isInitialState(s)$ then $s \leftarrow findNewOpenState(s, \mathcal{M}, \mathcal{A})$
7     end
8 end
```

Mobile platforms, such as Android, provide a `Back` button to undo actions. But this button is designed for app navigation and is context-sensitive. Thus, it is not a reliable mechanism for backtracking to precisely the previous state. For example, on state 5d of Figure 1, pressing the `Back` button will not lead us back to previous state 5b or 5c, not even to the previous screen (3), but to the screen (1) or (2) from where it was reached. Thus, the `Back` button need not take the navigation back to the immediately preceding state but to any of its ancestors. Hitting `Back` a finite number of times will eventually take the app to the initial screen. Our crawler uses a modified depth-first search, which tries to crawl only "forward" as much as possible using the `Back` button to backtrack when needed.

Algorithm 3 describes this strategy. It repeats a sequence of three steps till it can make no further progress at which point it terminates. The first step is a forward-crawling step implemented by function *forwardCrawlFromState()*

(Line 4). In this step the algorithm recursively visits states with open actions. It fires an open action and continues crawling till it reaches a state with no open actions. At this point function *backtrack()* (Line 5) is called to backtrack from the current state till another open state is found or one of the initial states of the crawl model is reached. In the former case forward crawling is resumed from this open state. In the latter case the function *findNewOpenState()* (Line 6) is used to find and crawl to a new open state and forward crawling is continued from there.

Algorithm 4: forwardCrawlFromState

```
Input  : $s_c$: state to crawl forward from, $\mathcal{A}$: app under test
         $\mathcal{M}$: crawled model being generated, $\mathcal{E}$: action map
Output: $s$: current state at the end of crawling
begin
    $s_x \leftarrow s_c$
    while $s_x \neq null$ do
        $s \leftarrow s_x$
        if $isNewState(s)$ then
            $initActions(s, \mathcal{E}, \mathcal{A})$
            $addToModel(s, \mathcal{M})$
        end
        $e \leftarrow getNextOpenAction(s)$
        if $e = null$ then $s_x \leftarrow null$
        else
            $s_x \leftarrow execute(s, e, \mathcal{A})$
            $updateOpenActions(s, e)$
            $addToModel(s, e, s_x, \mathcal{M})$
        end
    end
    return $s$
end
```

Algorithm 4 implements the function *forwardCrawlFromsState()* for forward crawling from a given state s_c. It iterates Lines 4-15 on the current state s, obtaining an open action e on s (*getNextOpenAction()*, Line 9) and executing it, to potentially reach another open state (function *execute()* on Line 12). The set of open actions of s is accordingly updated by function *updateOpenActions()* (Line 13) to reflect the changes. Further, the executed transition $s \xrightarrow{e} s_x$ is added to the model $\mathcal{M}$ by function *addToModel()* on Line 14. As an illustration, to completely crawl the sub-graph formed by states 5a-5d in Figure 1(b), the standard DFS would need to backtrack several times whereas our algorithm would cover it in a single forward crawl through the sequence `Menu` $\rightarrow$ `Settings` $\rightarrow$ `a1` $\rightarrow$ `a2` $\rightarrow$ `a1` $\rightarrow$ `a2` $\rightarrow$ `a2` $\rightarrow$ `a1` $\rightarrow$ `a2` $\rightarrow$ `a1` , by continuing to fire open actions.

6 Tool Implementation

We have implemented our reverse-engineering approach in a tool called ORBIT. It is composed the action detector and the dynamic crawler. Figure 3 shows an overview of ORBIT.

Action Detector. The action detector is implemented using the *WALA* static-analysis framework [24]. Android apps are event-driven and therefore organized as a set of event-handler callback methods. Thus, static analysis of just

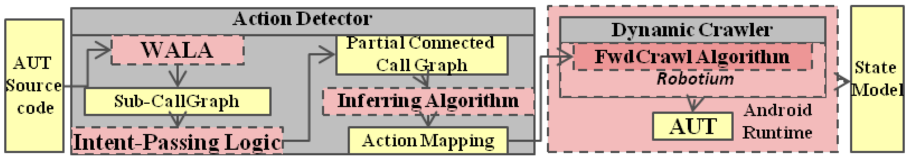


Fig. 3. Overview of the ORBIT tool

the app code gives a set of partial, disconnected sub call-graphs. The remaining behavior resides in the Android SDK, which we do not explicitly analyze. However, our tool incorporates an *intent-passing logic* module, created based on our knowldege of the Android SDK. For a given app, this module automatically builds a mapping of intent-sending methods and intent filters by analyzing the app's source code and manifest file. This mapping essentially connects the sub call graphs into a *partial connected call graph*. It is partial because for some intent-passing mechanisms like intent broadcasting whose behavior is affected by the runtime state of the Android system, we are unable to infer this information statically. Then we apply the action-inference algorithm described in Section 5.1 on the partial connected call graph to generate the *action mapping*.

Dynamic Crawler. Our crawler is built on top of the Robotium [5] Android test framework and implements the algorithms explained in Section 5.3. Although the crawler gets the list of actions from the Action Detector, it implements special handling for certain components such as *dynamically-created GUI components* and *system-generated GUI components* that are not statically declared.

Dynamically-created GUI components typically appear in Android containers like *ListView*, as a list of dynamically-created child components. Each child has the identical behavior, defined by the container. In such cases, the crawler represents the container as one of the two abstract states: an empty list and a non-empty list. Further, it randomly chooses *only one* of the child components to crawl further, by firing actions defined in the container.

System-generated GUI components typically have system-defined IDs and predefined actions. For example, the system-generated *context menu* is a ListView object, with ID *select_dialog_listview* and different menu options as child components, each with different behaviors. The crawler identifies such components at runtime and systematically crawls each child, rather than treating it as a generic container.

7 Evaluation

To assess the efficacy of our automated model extraction, we conducted a study addressing the following research questions:

RQ1: Is the proposed GUI crawling algorithm more efficient than a standard depth-first state-traversal algorithm?

RQ2: Are the widget and screen actions inferred by static analysis effective in enhancing the behavior covered by the generated model?

Subject	#LOCs	#Activities	Category	Purpose
TippyTipper	2238	5	Tool	Dining tip calculator
OpenManager	1595	6	Business	File manager for Android
Notepad	332	3	Productivity	Note creation and management
TomDroid	3711	3	Business	Online note reading
Aarddict	4518	4	Books & Reference	Aard Dictionary for Android
HelloAUT	234	1	Entertainment	Shape drawing & coloring
ContactManager	497	2	Productivity	Contacts manager
ToDoManager	323	2	Productivity	Task-list creation and management

Table 1. Test subjects used in the evaluation

RQ3: Can our tool generate a higher-quality model, more efficiently, compared to other state-of-the-art techniques?

Subjects. For our study, we use eight open-source Android apps that have also been used by other work on automated testing of mobile apps [10, 11, 21]. They are mostly small to medium-sized apps spanning a variety of application categories and are listed in Table 1.

Results. To address the three research questions, we carry out a corresponding experiment for each of the questions on all subjects. Among the subjects, *Notepad* can be started with multiple notes (Notepad2) or no note (Notepad0), which will substantially change the initial state of the crawling. To eliminate bias, we carry out every experiment on Notepad for both scenarios.

To address R1, we record the time spent, the coverage as well as the counts of forward actions (any actions other than back) and back actions exercised during both DFS traversing and our crawling (FwdCrawl) in Table 2. As shown in the Table, although both DFS and FwdCrawl can cover most of an app's behavior, DFS takes 70% more time to traverse all 9 subjects together.

Subject	FwdCrawl				DFS			
	Time(sec)	Coverage(%)	#Fwd	#Back	Time(sec)	Coverage (%)	#Fwd	#Back
TippyTipper	198	78	61	15	512	82	134	52
OpenManager	480	63	92	18	822	56	209	29
Notepad2	102	82	25	4	147	83	39	12
Notepad0	80	78	18	2	75	71	15	2
TomDroid	340	70	78	23	459	58	61	8
AardDict	173	65	15	2	397	60	20	8
HelloAUT	156	86	46	0	278	85	61	0
ContactManager	125	91	20	1	137	92	22	2
ToDoManager	178	75	60	2	294	74	84	4

Table 2. Comparison of standard DFS-based crawling vs. proposed forward crawling

The second experiment is to run our traversal algorithm with click actions only instead of inferred actions. To address R2, we record the coverage and counts of *clicks*, *longClicks*, and *menu*, the three most common actions fired during crawling. The results show that non-click actions constitute only 22% of the total actions but firing these actions during crawling increases the coverage by 34% on average. The low proportion of these actions also supports the argument made in Section 5 that blindly firing all supported actions will produce a large number of invalid edges in our model. Table 3 also shows that our crawling produces fairly compact models with a few states.

We also compare ORBIT with other existing Android GUI ripping tools to address R3. In Table 4, we compare ORBIT with *Android GUITAR* [21], *Android GUI Ripper* [11] and Android's *Monkey* tool. As Android GUI Ripper

Subject	**#clicks**		**#longClicks**		**#menu**		**#States**		**Coverage(%)**	
	C	C+I	C	C+I	C	C+I	C	C+I	C	C+I
TippyTipper	21	55	–	2	–	4	3	9	47	78
OpenManager	50	67	–	19	–	4	10	20	39	63
Notepad2	2	13	–	3	–	9	2	7	39	82
Notepad0	0	8	–	1	–	9	0	7	14	78
TomDroid	3	52	–	0	–	26	2	9	36	70
AardDict	4	15	–	0	–	7	3	7	43	64
HelloAUT	15	34	–	0	–	12	4	8	53	86
ContactManager	20	20	–	0	–	0	5	5	92	92
ToDoManager	60	60	–	0	–	0	7	7	76	76

Table 3. Comparison of Crawling with and without Action Inference

takes substantially long time to run, we use the runs of its generated test cases to do the comparison. The time of each run was recorded from the start of the AUT (App Under Testing) to the generation of coverage report. The time along with coverage shows that our crawler is 32%-75% faster while constructing a 5%-140% more complete model than Android GUITAR and Android GUI Ripper.

For illustration purposes, we also compare our tool against the Android Monkey tool. Monkey fires a pseudo-randomly-generated action sequence, of a specified length, on the app. For our subjects, we found that the maximum coverage achieved by Monkey tended to saturate at around 1200 events. For our experiment, we ran Monkey 10 times with a 1500 event count on each app, and report the median of the coverage achieved in these 10 runs in Table 4. Indeed, for the given event count, Monkey is much faster than ORBIT but achieves substantially lower coverage. This result underscores the benefit of the systematic crawling performed by ORBIT.

Subject	**Monkey**		**Android GUITAR**		**Android GUI Ripper**		**ORBIT**	
	Time(sec)	Cov.(%)	Time(sec)	Cov.(%)	Time(sec)	Cov.(%)	Time(sec)	Cov.(%)
TippyTipper	83	41	322	47	-	-	198	78
OpenManager	90	29	-	-	-	-	480	63
Notepad2	127	60	-	-	-	-	102	82
Notepad0	122	59	-	-	-	-	80	78
TomDroid	69	46	-	-	529	40	340	70
AardDict	124	51	-	-	694	27	173	65
HelloAUT	98	71	117	51	-	-	79	98
ContactManager	90	53	247	61	-	-	125	91
ToDoManager	115	71	194	71	-	-	121	75

Table 4. Comparison of ORBIT with other tools

8 Discussion

Crawling algorithm. Our crawling algorithm is faster than DFS for every subject except *Notepad0*. By examining the execution log, we found that our algorithm had traversed two more states than DFS, accounting for the difference. Such result is due to the randomness in the choice of the next action to explore and to the side effects of execution. When crawling by DFS, the crawler happened to click the *delete note* button first before clicking the *open note* button. Since there are no notes left after deletion, the crawler cannot visit the edit note screen. Our crawling happened to click the edit button before delete, so we are able to traverse the editing screen. The randomness can be mitigated by carrying out multiple runs. We also plan to consider controlling the order of event sequences as part of future work.

Selection of subjects. Our evaluation is based on subjects drawn from existing related tools, and we try to avoid bias by including all subjects used to evaluate Android GUITAR and GUI Ripper in previous work. However, we do see a preference in the choice of subjects made by these tools. Both of the tools seemed to select subjects with few non-click actions. For GUI Ripper, both the subjects do not have longClick actions, although we did find a later version of TomDroid that has longClicks. For GUITAR, because GUITAR does not support non-click actions, two of its subjects, ContactManager and ToDoManager, do not support non-click actions at all. In general, Android apps have a wide variety of actions, and we apply our methodology against on apps with multiple actions and those with only one or two kinds of actions. The both results show that our methodology is effective on both of the cases.

ORBIT vs Android GUITAR. As Android GUITAR can fire only click actions, it seems unfair to use our results with action inference for comparison. If we compare our click-only runs with GUITAR, we observe that for most of the subjects, Android GUITAR's coverage rate in Table 4 is comparable to our click-only coverage in Table 3. So we infer that our advantage in model completeness is largely attributable to the action-detection technique. Another difference between the two tools is that GUITAR was initially created for desktop applications and its event-flow model typically contains many invalid paths, while ORBIT is designed specifically for mobile apps, and uses a more precise state-based model, which would also integrate well with other state-based testing techniques.

Manual effort. The only manual work in our approach is to manually select attributes of executable components to compose the visual observable states for the GUI. This effort is a one-time effort for a mobile platform. As we have already performed this exercise for Android apps, additional effort will be required only when applying our technique on other mobile platforms to make minor adjustments or revisions for Android.

9 Conclusion

In this paper, we have proposed an approach for automatically reverse-engineering GUI models of mobile apps. We described our tool called ORBIT that implements our approach for Android, and presented the results of our empirical evaluation of this tool on several Android apps. The results showed that for these apps, ORBIT efficiently extracted high-quality models fully automatically.

References

1. comScore Inc.: State of the Internet: Q1 2012. `http://tiny.cc/c9gkqw`
2. netimperative: Twitters mobile ad revenue overtakes desktop PCs. `http://tiny.cc/74gkqw` (2012)

3. engadget.com: Google Play hits 600,000 apps, 20 billion total installs. http://www.engadget.com/2012/06/27/google-play-hits-600000-apps/ (2012)
4. Android Developers Site. http://developer.Android.com/
5. Google Project Hosting: Robotium. http://code.google.com/p/robotium/
6. Pivotal Labs: Robolectric. http://pivotal.github.com/robolectric/
7. Bitbar: Testdroid. http://testdroid.com/
8. Contus: Mobile App Testing. http://mobileappstesting.contussupport.com/
9. Anand, S., Naik, M., Harrold, M.J., Yang, H.: Automated concolic testing of smartphone apps. In: Proc. 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering. FSE '12 (2012) 59:1–59:11
10. Nguyen, C.D., Marchetto, A., Tonella, P.: Combining model-based and combinatorial testing for effective test case generation. In: Proc. International Symposium on Software Testing and Analysis. ISSTA 2012 (2012) 100–110
11. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI ripping for automated testing of Android applications. In: Proc. 27th IEEE/ACM International Conference on Automated Software Engineering. ASE 2012 (2012) 258–261
12. Mirzaei, N., Malek, S., Păsăreanu, C.S., Esfahani, N., Mahmood, R.: Testing Android apps through symbolic execution. Volume 37. (November 2012) 1–5
13. Joorabchi, M.E., Mesbah, A.: Reverse engineering iOS mobile applications. In: Proc. 19th Working Conference on Reverse Engineering. WCRE '12 (2012) 177–186
14. Syer, M.D., Adams, B., Zou, Y., Hassan, A.E.: Exploring the development of micro-apps: A case study on the Blackberry and Android platforms. In: Proc. IEEE 11th International Working Conference on Source Code Analysis and Manipulation. SCAM '11 (2011) 55–64
15. Pezzè, M., Young, M.: Software testing and analysis - process, principles and techniques. Wiley (2007)
16. Memon, A., Banerjee, I., Nagarajan, A.: GUI ripping: Reverse engineering of graphical user interfaces for testing. In: Proc. 10th Working Conference on Reverse Engineering. WCRE '03 (2003) 260–269
17. Mesbah, A., van Deursen, A., Lenselink, S.: Crawling ajax-based Web applications through dynamic analysis of user interface state changes. ACM Trans. Web **6**(1) (2012) 3:1–3:30
18. 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, Verification and Validation. ICST '11 (2011) 377–386
19. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proc. 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies. WEASELTech '07 (2007) 31–36
20. Shafique, M., Labiche, Y.: A systematic review of model based testing tool support. Technical Report SCE-10-04, Carleton University, Canada (2010)
21. Sourceforge: Android GUITAR. http://tiny.cc/8eapqw
22. Dallmeier, V., Burger, M., Orth, T., Zeller, A.: WebMate: a tool for testing Web 2.0 applications. In: Proc. Workshop on JavaScript Tools. JSTools '12 (2012) 11–15
23. Hu, C., Neamtiu, I.: Automating GUI testing for Android applications. In: Proc. 6th International Workshop on Automation of Software Test. AST '11 (2011) 77–83
24. Sourceforge: WALA. http://wala.sourceforge.net/wiki/index.php