SOFTWARE TESTING, VERIFICATION AND RELIABILITY

Softw. Test. Verif. Reliab. 2010; **00**:2–32

Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Automated Generation of Oracles for Testing User-interaction Features of Mobile Apps

Razieh Nokhbeh Zaeem[1,2], Mukul R. Prasad[2] and Sarfraz Khurshid[1]

[1]*The University of Texas at Austin, 1 University Station C5000, Austin, TX 78712*

[2]*Fujitsu Laboratories of America, 1240 E. Arques Ave. M/S 345, Sunnyvale, CA 94085*

SUMMARY

As the use of mobile devices becomes increasingly ubiquitous, the need for systematically testing applications (apps) that run on these devices grows more and more. However, testing mobile apps is particularly expensive and tedious, often requiring substantial manual effort. While researchers have made much progress in automated testing of mobile apps during recent years, a key problem that remains largely untackled is the classic *oracle* problem, i.e., to determine the correctness of test executions. This paper presents a novel approach to automatically generate test cases, that include test oracles, for mobile apps. The foundation for our approach is a comprehensive study that we conducted of real defects in mobile apps. Our key insight, from this study, is that there is a class of features that we term *user-interaction* features, which is implicated in a significant fraction of bugs and for which oracles can be constructed – in an application agnostic manner – based on our common understanding of how apps behave. We present an extensible framework that supports such domain specific, yet application agnostic, test oracles, and allows generation of test sequences that leverage these oracles. Our tool, QUANTUM, embodies our approach for generating test cases that include oracles. Experimental results using 6 Android apps show the effectiveness of QUANTUM in finding potentially serious bugs, while generating compact test suites for user-interaction features.

Received . . .

KEY WORDS: software testing; oracles; mobile apps; Android; user-interaction features

Prepared using stvrauth.cls *[Version: 2010/05/13 v2.00]*

1. INTRODUCTION

Recent years have witnessed an explosive growth in the use of mobile devices and in the number and variety of software applications developed for such devices. Mobile applications, or apps as they are popularly called, are often developed in small, fast-paced projects with scarce testing resources. At the same time, testing mobile apps presents some unique challenges, such as supporting a wide range of devices, platforms and versions, as well as ensuring the integrity of the rich and highly interactive user-interface characteristic of such apps [1]. Thus, there is a growing need to develop automated testing tools to support the development of mobile apps.

Researchers have made significant progress in developing techniques to support automated testing of mobile apps [2, 3, 4, 5, 6]. However, these techniques primarily target the generation of test sequences, leaving the task of adding test oracles [7, 8] into these test sequences to the human tester. This itself can be a manually intensive process and if the oracles are not of a sufficiently high quality, can potentially compromise the efficacy of test cases.

The aim of this paper is to partially address the oracle problem in the context of automated test case generation for mobile applications. To realize this aim we conducted a study where we sampled, studied and categorized the bugs reported for several popular, open-source Android applications. The study revealed that a significant fraction of bugs can be attributed to *user-interaction* features that are supported by the mobile platform and simply implemented by each application. Such features include content presentation or navigation features such as rotating the device or using various gestures to scroll or zoom into screens. A distinguishing characteristic of these features is

[1]Correspondence to: The University of Texas at Austin, Austin, TX 78712. E-mail: nokhbeh@utexas.edu

Contract/grant sponsor: Fujitsu Labs of America and the National Science Foundation; contract/grant number: SRA No. UTA12-001194 and NSF Grant Nos. CCF-0845628 and CNS-1239498.

[2]This author was an Intern at Fujitsu Labs of America for a part of this work.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

that they are largely independent of the core logic of the application. More importantly, there is often a general, common sense expectation of how the application should respond to a given feature. For example, rotating a device and then rotating it back should bring the display precisely back to the initial screen. Such observations motivate our approach.

We present a novel framework for authoring test oracles for checking user-interaction features of mobile applications, in an application agnostic manner. Our framework supports *model-driven test suite generation* [9] where each generated test includes both the test sequence to execute and the corresponding assertions to check (as test oracles). Given a model of the user-interface of the mobile app under test, our framework uses its built-in, extensible library of oracles (for various user-interaction features) and generates a test suite to comprehensively test the app against user-interaction features.

While the basic goal of our framework is to allow generation of test suites that are complete with embedded test oracles for supported features, it includes two more techniques to further enhance its usefulness in practice. Firstly, our framework supports a customizable *cost* function that defines a measure of cost for executing a given test suite and produces an output suite that has likely minimal execution cost, while checking each feature. Secondly, our test generation technique inserts multiple test oracles, for *different* features, within a *single* test case, when possible. This allows checking of multiple properties within the same test execution, by conceptually sharing execution segments common across different tests, thus reducing the overall test execution cost. Our test generation technique produces *feature-adequate* test suites, which for the given model exercise every transition relevant to each supported feature and test its expected functionality.

Our tool, QUANTUM, embodies our framework and provides a fully automated, *push-button* tool-set for test case generation for mobile apps. Our initial experiments with QUANTUM show that it generates valuable test suites and provides the foundation of a promising approach for more effective testing of mobile apps.

This paper makes the following contributions:

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Bug study. We perform a comprehensive study of real mobile application defects and identify a family of mobile application features, which we term *user-interaction features*. We observe that these features are implicated in a significant fraction of the studied defects. Further, they are characteristic of the mobile platform and implemented by many mobile applications but not directly dependent on the application logic.

Feature-driven testing of mobile apps. We introduce a novel form of *test adequacy* [10] in the context of mobile apps where the goal is to cover the given model of the app's user-interface by exercising each transition relevant to any desired feature and checking the expected functionality for the feature.

Automatic oracle generation for testing mobile apps. We present an extensible library of oracles for various user-interaction features. Our framework allows authoring test oracles for features, in an application agnostic manner, for re-use across a number of different apps that are expected to support those features. These oracles are appropriately instantiated by our model-driven test suite generation technique.

Minimal cost test suite generation. We provide a technique for generating a compact test suite by trying to minimize a customizable cost function. The test suite also incorporates the oracles to comprehensively test the supported feature set.

Evaluation. We present our tool, QUANTUM, for automated testing of mobile apps and its evaluation on 6 real Android applications. The evaluation confirms that QUANTUM is able to generate compact test suites, complete with test oracles, for testing the identified features. These test suites are able to reveal a number of bugs in the studied applications: QUANTUM found a total of 22 bugs, a few of them particularly serious, using a total of 60 tests for these 6 apps.

2. BUG STUDY

We conducted a bug study on 106 bugs drawn from 13 open-source Android applications. The aim was to identify opportunities for automatically generating test cases, that include test oracles,

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table I. Subjects for Bug Study

App	Function	Source
Notepad	Note Making Tool	developer.android.com/tools/samples
CMIS	CMIS Browser	gc/android-cmis-browser
Delicious	Social Bookmarking	gc/android-delicious-bookmarks
OpenSudoku	Sudoku Game	gc/opensudoku-android
MonolithAndroid	3D Game	gc/monolithandroid
Wordpress	Blogging Tool	android.trac.wordpress.org
Nexes Manager	File Manager	github.com/nexes/Android-File-Manager
VuDroid	PDF Viewer	gc/vudroid
Kitchen Timer	Timer	gc/kitchentimer
Dolphin Player	Media Player	gc/dolphin-player
AnkiDroid	Flashcard Review	gc/ankidroid
Shuffle	Personal Organizer	gc/android-shuffle
K9Mail	Email Client	gc/k9mail

gc: https://code.google.com/p

by focusing on bugs specific to mobile apps and by exploiting domain knowledge of the mobile platform.

The 13 open-source Android apps we selected included 6 apps studied in previously published work on automated testing for mobile apps [11, 4, 2], a further 6 apps selected from the open source repository Google Code, and the *Notepad* sample app provided for educational purposes by the official Android website (also studied in previous work [2]). Table I lists the name, stated function, and source for each of the 13 subjects.

Our aim was to choose test subjects from a diverse set of application categories and functions. The 6 apps CMIS, Delicious, OpenSudoku, MonolithAndroid, Wordpress, and Nexes Manager, chosen from previously published work, reflect this intention. Further, we applied the following five additional criteria to choose the 6 apps from open source repositories: (1) popularity: a minimum ranking of 3.5 out of 5 on Google Play, (2) high number of active installations: a minimum of 50,000, (3) having active development communities: the latest version of the source of the application should have been downloaded at least 1000 times, (4) rich database of reported issues: at least 25 reported issues, and (5) reproducible defects: the app should have at least some defects reproducible on a standard Android emulator. Similar criteria have been used in previous studies

Prepared using stvrauth.cls
DOI: 10.1002/stvr

of Android apps [11], albeit for somewhat different purposes. By manually browsing Google Code with the above selection protocol we selected the 6 apps VuDroid, Kitchen Timer, Dolphin Player, AnkiDroid, Shuffle, and K9Mail which have on average a ranking 4.3 out of 5, 500,000 active installations, 10,700 downloads, and 1,400 reported issues.

Generally, only a small fraction of issues logged in the bug repository of an app are true, reproducible bugs. Many of them cannot be reproduced and still others are merely feature requests. To select bugs for further investigation, for each test subject we manually examined each issue logged in its repository till we had 10 reproducible bugs (except for Delicious, MonolithAndroid, and Nexes, selected from previous work, that have small bug repositories where we could find only 8, 6, and 2 reproducible defects respectively). No bug reports exist for Notepad. This gave us a total of 106 bugs.

We manually investigated and categorized each of the 106 bugs, from the viewpoint of the test oracles needed to detect them. We identified 20 categories besides the core application logic. Table II shows this categorization. We observed that almost 75% of the studied bugs are not directly tied to the application logic (only 27 bugs are categorized under *Application Logic*), which inspired us to explore avenues for automatically generating test oracles tailored for mobile apps.

We further aggregated the categories based on the automatability of the underlying oracles. Oracles enforcing the application logic are very application specific and notoriously difficult to generate fully automatically. Other than this category we identified 3 groups of oracles: Basic Oracles, App Specific Oracles, and App Agnostic Oracles.

2.1. Application Logic Oracles

A total of 27 bugs were directly related to the application logic. For example, K9Mail has a logic bug (issue 2931) in which "under a certain setting (e.g., an IMAP account with an email server that reuses UIDs) K9Mail shows the header of an old email when it receives a new email". The post-condition for an oracle that checks and finds this inconsistency is largely dependent on the logic of email client applications. As another example, Shuffle has a logic bug (issue 65) where "it

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table II. Categorization of Bugs.

Group→	Basic Oracles										App Agnostic Oracles				App Specific Oracles						
Category →	Loading Lib.	Third Party Lib.	Uncaught Exception	Key Signing	Incompatibility	Memory	Busy Resource	SQL	Infinite Loop		Rotation	Activity Life-Cycle	Gestures	Time Zone	Input Handling	Settings	Showing Progress	Visual Appearance	Foreign Languages	Widget	Website Connection	Application Logic	
Notepad			4				1	1													2	2	
CMIS			1														1	1	1			4	
Delicious		1									1				1	2			3			2	
OpenSudoku				1	1	1					1					1		1			1		
MonolithAndroid			3						1			1		1						1	1	2	
WordPress											2		1			2						2	
Nexes Manager																						2	
VuDroid	1		3	1							2		1					1				1	
Kitchen Timer			1								2	1	1			2						3	
Dolphin Player	1				1	1	1											2	1			3	
AnkiDroid			2					1			1		1			1		2				2	
Shuffle			1				1		1					1						1	2	3	
K9Mail		1											1		3	1	1					3	
Total	**2**	**2**	**15**	**2**	**2**	**2**	**3**	**2**	**2**		**5**	**4**	**4**	**2**		**4**	**7**	**2**	**7**	**5**	**2**	**5**	**27**

Prepared using stvrauth.cls
DOI: 10.1002/stvr

duplicates events on rescheduling, i.e., it fails to delete the old event after rescheduling it". In order to find such bugs, one requires application specific post-conditions for oracles. Therefore, we do not consider this group of bugs in this work.

2.2. Basic Oracles

Basic Oracles encompass general instances of aberrant program behavior such as crashes, hangs, or illegal terminations which are not application specific, or even specific to mobile apps. As an example, *Uncaught Exceptions* belongs to this group. CMIS has a bug (issue 25) in which "if the URL field is left empty, a null pointer exception is thrown". Another example of this group of bugs is *Loading Library* where unsatisfied link errors cause the application to crash, e.g., "Vudroid crashes because of an unsatisfiable link error to the libc library on Archos devices" (issue 3). Such basic oracles are already widely used in automated software testing and hence not particularly interesting for the current investigation.

2.3. App Specific Oracles

Another group, named *App Specific Oracles*, are not directly related to the application logic but can still be very application specific. For example, oracles to validate the *Visual Appearance* of an app belong to this group. The MonolithAndroid app has a defect (issue 10) in which "holes exist in the background texture" of the app rendering. One more example of this group is the *Foreign Languages* category. Delicious has a bug from this category where "bookmarks in the Russian language are garbled" (issue 16). We feel it would be very hard to generate precise automated oracles to distinguish between intended and faulty behavior in such cases. Therefore, our work does not target this category either.

2.4. App Agnostic Oracles

However, the group of *App Agnostic Oracles* contains bugs for which the oracles are significantly more complicated than the basic oracles but sufficiently app agnostic that they could potentially

Prepared using stvrauth.cls
DOI: 10.1002/stvr

be automatically generated. We found relatively well populated categories like *Rotation*, *Activity Life-cycle*, and *Gesture* Bugs in this group.

Rotation bugs manifest as the mobile device is rotated from landscape to portrait orientation or vice versa. There is a common understanding of how applications usually respond to rotation: the same content should stay on the screen, in a possibly different arrangement, and should support the same actions as before. In addition, user data entries should be preserved after rotation. VuDroid contains an example of a rotation bug where "the tab selection resets after rotating the phone". Our bug study found five rotation bugs in three apps. Gesture bugs form another category, similar to rotation bugs, where common gestures such as zooming in and out, scrolling, and selecting text produce a response contradicting common sense expectation. K9Mail has a gesture bug where "it is not possible to select text more than one line" in a particular version of Android (issue 3435). The study found four gesture bugs in four apps.

The graphical user-interface of Android apps is composed of components called *Activities*, each corresponding to a core function of the app. An Activity's behavior should conform to an *activity life-cycle*, a finite state machine where each state represents one coarse level state (such as active or paused)[2]. Activity life-cycle bugs correspond to aberrant behavior exhibited as the app's Activity components transition through different life-cycle states. This happens, for example, as an app is sent to the background, killed, resumed or re-started. Similar to rotation bugs, there is a common sense understanding of how apps should behave when they are paused or killed. For example, when an app is paused and subsequently resumed, it should preserve the user's data entries on the current screen. Our study found four activity life-cycle bugs in four apps. For example, Wordpress has one such bug where "the content disappears if the app goes to the background". Note that rotation and gesture bugs *might* share root causes with activity life-cycle bugs. However, these bug categories are not causally linked. Further, they represent different bugs from the end user's perspective and hence merit being tested independently.

[2] http://developer.android.com/guide/components/activities.html#Lifecycle.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Finally, time zone bugs occur when handling different time zones. We found an example of a time zone bug in WordPress (issue 190) in which "publishing a post from a device in an earlier time zone than the WordPress server works, but editing it thereafter will schedule the post in the future". The study found two time zone bugs in two apps. However, while app agnostic, these bugs do not directly arise from user-interactions, and hence are different from the other three categories in this group.

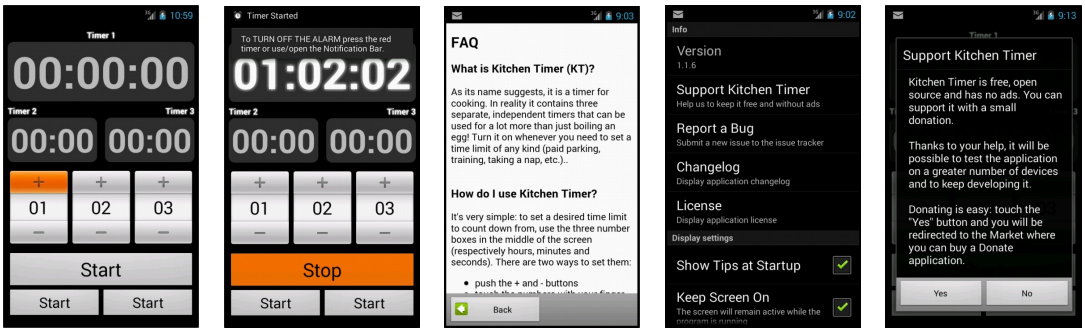
The bug study revealed that many bugs in mobile apps have shared roots beyond the application logic. It further helped us identify three categories of oracles to find such bugs: (1) the basic oracles, which mainly catch exceptions and error messages, (2) the app specific oracles, which require the human tester to detail the intended behavior in each case, and (3) the app agnostic oracles. The app agnostic oracles mostly (with the exception of time zone bugs) correspond to ways of *interacting* with mobile devices, common not only between different apps but also among various mobile platforms. The study found that these *user-interaction features* of mobile apps are the cause of bugs in more than half of the apps studied. Inspired by these bugs, we introduce our test and oracle generation techniques for *user-interaction features* of mobile apps.

3. EXAMPLE

In this section, we present an example app to motivate our technique for automatically testing user-interaction features of mobile apps. For illustration, we use a real bug from Kitchen Timer, an open source Android app. As the name suggests, Kitchen Timer is a timer for cooking. It contains three timers which can be set independently and go off by sounding an alarm after counting down to zero. In addition, Kitchen Timer provides other functionalities to change Preferences (e.g., the alarm sound, the LED color, timer names), save preset timers, and many more.

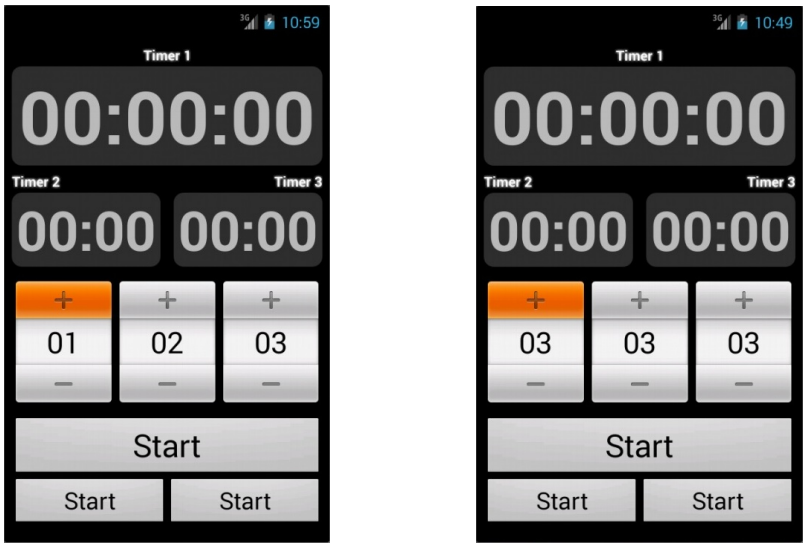
While we used the complete Kitchen Timer for the bug study and evaluation (Sections 2 and 5), we describe a simplified version for the sake of this example. Figure 1 shows snapshots of Kitchen Timer. In this simplified Kitchen Timer, the user can set one timer by using the plus and minus

Prepared using stvrauth.cls
DOI: 10.1002/stvr



(a) Main Timers (b) Timer Running (c) Info (d) Preferences (e) Donation

Figure 1. Snapshots of Simplified Kitchen Timer.



(a) The user sets a timer. (b) Values overwritten after two rotations.

Figure 2. Snapshots of a Bug in Kitchen Timer.

signs on the main screen (Figure 1a). The numbers from left to right show the hours, minutes, and seconds for the timer to be set. Then the user can start the timer by hitting the Start button (to get to Figure 1b) or stop it by hitting the Stop button. Furthermore, he can select Info, Preferences, or Donation from the menu (to get to Figures 1c, 1d and 1e). All of these screens support rotation. For simplicity, we excluded further actions from Info, Preferences, and Donation from the simplified app. The user can, however, go back to the main screen from them.

As Figure 2 depicts, Kitchen Timer has a bug that is manifested when the device is rotated twice. If the user sets a timer (Figure 2a), and then rotates the mobile device twice before starting the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

timer (Figure 2b), the value of seconds overwrites the values of minutes and hours, here changing the timer from 1 hour, 2 minutes, and 3 seconds to 3 hours, 3 minutes, and 3 seconds. Our tool automatically finds this bug.

4. FEATURE-BASED TESTING OF MOBILE APPS

The findings of our bug study motivated us to develop an approach for automatically testing user-interaction features (interaction features, or simply features for short throughout the paper) of mobile apps. Our proposed approach is described in this section. We start by defining some terminology.

Definition 1 (Interaction feature)

An interaction feature is an action supported by the mobile platform, which enables a human user to interact with a mobile app, using the mobile device and the graphical user-interface (GUI) of the app. Further, an interaction feature is associated with a common sense expectation of how the mobile app should respond to that action.

Interaction features include actions like rotating a mobile device, general purpose gestures like zooming in/out or scrolling, and actions which start, pause, kill or resume operation of an app, taking its Activity GUI components through various states in their life-cycle. These features were discussed in our bug study. In addition, features like the *Back* or *Up* buttons of the Android platform[3] are also valid interaction features. Note that the above definition *excludes* a number of common gestures such as `click` or `longClick`, or other custom gestures, for which there is no standard expected response from apps; it is completely context and application specific. Since a given interaction feature will have, in general, a standard expected behavior, across apps and different mobile platforms[4], this provides a general, app agnostic *oracle* for validating an app's response to exercising that feature. Thus, a key component of our approach is authoring such reusable oracles and employing them in interaction feature testing.

[3]See `http://developer.android.com/design/patterns/navigation.html`.

[4]Specific apps may of course choose to modify this standard response.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

We follow a model-driven approach to generating test-cases for testing interaction features of a given mobile app. The starting point for our technique is a finite state model of the GUI behavior of the app, which is defined as follows.

Definition 2 (GUI model)

A GUI model of an app is a finite state machine $\mathcal{M}$, denoted by the 4-tuple $\mathcal{M} = (S, s_0, A, R)$, where S is a finite set of abstract states representing different GUI screens, $s_0 \in S$ is the initial state denoting the app's opening screen, A is a finite set of application specific actions the user may perform in executing the logic of the app, and $R \subseteq S \times A \times S$ is a transition relation describing transitions between the states in S in response to user actions from A.

Two GUI screens are represented by the same abstract state in $\mathcal{M}$ if and only if they contain the same set of actions on the same widgets. The only exception to this is screens showing collections of items, such as books, files, songs, transactions, *etc.*, where each item supports some set of actions. In this case two screens with different (non-zero) numbers of items are interpreted as the same state. Thus, the contents of a collection are abstracted as empty or non-empty. Similar notions of GUI states have also been used in previous work [12, 6]. The set A includes application specific actions such as clicks or longClicks, *etc.,* on specific widgets but *does not* include platform supported interaction features (e.g., device rotation, *etc.*). We believe this is typical of GUI models as well [13].

Note that although the visible part of a GUI screen of a mobile app, as viewed on a mobile device, may change by performing an action such as a device rotation, a zoom, or a scrolling action, these apparently different screens still correspond to the same abstract state in the GUI model. We define the notion of a *view*, denoted by the symbols w, to represent the visible portion of abstract GUI model states s. Thus, a state s can have several views, generated by exercising different available interaction features on s. Specifically, we use the notation $\Phi(s, \mathbf{u}, -)$ and $\Phi(s, \mathbf{u}, +)$ to denote respectively, the two different views of state s before and after action (or action sequence) $\mathbf{u}$ was fired, where $\mathbf{u}$ corresponds to an instance of exercising an interaction feature. The view notation provides a relative notion of *time* of sampling states (for their current view), before and after exercising interaction features.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

A GUI model $\mathcal{M}(S, s_0, A, R)$ can also be represented as a rooted, labeled directed graph $G = \langle V, E, r, A, \mathcal{L} \rangle$, in a straightforward manner. Here, the nodes V represent the states S, the root node r represents initial state s_0, edges E represent transitions between states, consistent with transition relation R, and the labeling function $\mathcal{L}\colon E \rightarrow A$ labels each edge with the action $a \in A$ responsible for the transition. GUI models can either be constructed manually or generated automatically using one of the techniques from a growing body of work on GUI model generation for mobile apps [4, 5, 13].

Overall Approach: Our technique generates compact test suites, complete with test oracles, to comprehensively test interaction features of a given mobile app. The approach uses an extensible library $\mathcal{F}$ of reusable and application agnostic feature definitions, described in Section 4.1. Given a user provided GUI model of the app we automatically augment this model with feature instances, using the feature definitions in $\mathcal{F}$ (Section 4.2). Then, based on the cost and test adequacy criteria defined in Section 4.3, we automatically traverse the augmented model to create compact test sequences (Section 4.4). Finally, we automatically instantiate test oracles in the test sequences to obtain a compact and complete test suite.

4.1. Authoring Oracles for Interaction Feature Testing

We introduce an extensible framework in which interaction features can be defined in an application agnostic manner and stored in a library. When testing a given app our technique appropriately instantiates features from the library, using these feature definitions, and generates tests, complete with test oracles, to comprehensively test each feature.

Definition 3 (Feature definition)

The feature definition of a given interaction feature f is a triple: $\langle \mathbf{u}_f, D_f(s), O_f(w_1, w_2) \rangle$. $\mathbf{u}_f = \langle u_1, u_2, \ldots, u_n \rangle$ is a sequence of actions that exercises the feature. $D_f(s)$ is the destination function, that maps a given state s at which the feature can be exercised to a set of states $S_f \subseteq S$ that could potentially result from exercising f at state s. $O_f(w_1, w_2)$ is the oracle for feature f, where $w_1 = \Phi(s_1, \mathbf{u}_f, -)$ is a view of some state s_1 before firing actions $\mathbf{u}_f$ and $w_2 = \Phi(s_2, \mathbf{u}_f, +)$ is a

Prepared using stvrauth.cls
DOI: 10.1002/stvr

view of a state s_2 reached after firing actions $\mathbf{u}_f$ on some previous state s_1, possibly the same state as s_2.

A crucial aspect of the above feature definition is to express $\mathbf{u}_f$, D_f, and O_f in an application agnostic manner. We demonstrate how to do this below, through example feature definitions of several common interaction features. Another important restriction implied by Definition 3 is that the set of abstract states in the GUI model should be closed under the application of the interaction feature, *i.e.,* exercising the feature in one of the states should not take the application to a fundamentally new abstract state outside the GUI model. This common sense restriction is also valid for all interaction features in our knowledge. In the following examples we use $\Phi^-(s)$ and $\Phi^+(s)$ as shorthand for $\Phi(s, \mathbf{u}_f, -)$ and $\Phi(s, \mathbf{u}_f, +)$ respectively, since $\mathbf{u}_f$ is clear from the context.

Double rotation (DR): We incorporate the mobile device rotation feature in a *double rotation* feature definition, which expresses the act of rotating a mobile device and then rotating it back to the original orientation. With this action the application should stay in the same state. Further, the view of that state before an after double rotation should be identical. This is expressed in the feature definition: $DR = \langle \mathbf{u}_f = \langle rotate, rotate \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) = \Phi^+(s)) \rangle$.

Killing and restarting (KR): The operating system might choose to kill and then restart an app for various reasons (*e.g.,* low memory). Similar to double rotation, the app should retrieve its original state and view. Thus, $KR = \langle \mathbf{u}_f = \langle kill, restart \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) = \Phi^+(s)) \rangle$.

Pausing and resuming (PR): The app can be paused (*e.g.,* by hitting the Android Home button) and then resumed. $PR = \langle \mathbf{u}_f = \langle pause, resume \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) = \Phi^+(s)) \rangle$. Killing and then restarting, and pausing and then resuming are both instances of activity life-cycle transitions which all apps should support.

Back button functionality (Back): The Back button is a hardware button on Android devices which takes the app to the previous screen. $Back = \langle \mathbf{u}_f = \langle back \rangle, D_f(s) = \{s_p : s_p \in parent(s)\}, O_f = (\Phi^-(s_1) = \Phi^+(s_1)) \rangle$, where $s_1 \in D_f(s)$. In this case, the destination function produces a set of destinations $D(s)$ corresponding to each of the parent (using the standard graph theoretic notion of parent and child) nodes of the current state s in the GUI model.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Opening and closing menus (Menu): The hardware Menu button on Android devices opens and closes custom menus that each app defines. For this feature definition $Menu = \langle \mathbf{u}_f = \langle menu, menu \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) = \Phi^+(s)) \rangle$.

In the above instances the oracle was always an assertion of equality between two appropriate state views. In general, however, the oracle predicate can include arbitrary relational or logical operators. For example:

Zooming in (ZI): Zooming into a screen should bring up a subset of what was originally on the screen. $ZI = \langle \mathbf{u}_f = \langle zoomIn \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) \supset \Phi^+(s)) \rangle$.

Zooming out (ZO): Zooming out from a screen should result in a superset of the original screen. $ZO = \langle \mathbf{u}_f = \langle zoomOut \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) \subset \Phi^+(s)) \rangle$.

Scrolling (SCR): Scrolling down (or up) should display a screen that shares parts of the previous screen. $SCR = \langle \mathbf{u}_f = \langle scrollDown \rangle, D_f(s) = \{s\}, O_f = (\Phi^-(s) \cap \Phi^+(s) \neq \emptyset) \rangle$.

Note that the feature definition itself includes an implementation of the oracle, albeit an app independent one, that can be re-used across different apps. Thus, the semantics of operators used in the oracles are defined there.

4.2. Augmenting GUI Models with Feature Instantiations

Given a GUI model $G = \langle V, E, r, A, \mathcal{L} \rangle$ of the target app and a library $\mathcal{F}$ of interaction features, specified as discussed in Section 4.1, the next step in our approach is to annotate G with all possible instantiations of each feature in $\mathcal{F}$ to produce an augmented GUI model $G^+ = \langle V, E^+, r, A^+, \mathcal{L}^+ \rangle$. Specifically, this involves adding a set of special labeled edges, called *golden edges*, to G. Each golden edge, $e_f(v_1, v_2)$ denotes that feature f when exercised at the state of vertex v_1 takes the application to the state of vertex v_2. Further, e_f is labeled with $\mathbf{u}_f$, the action sequence of feature f. Thus, the augmented model G^+ includes the augmented set of edges $E^+ = E \cup E_{golden}$, augmented action set $A^+ = A \cup \bigcup_{f \in \mathcal{F}} \mathbf{u}_f$, and appropriately modified labeling function $\mathcal{L}^+ : A^+ \rightarrow E^+$, where E_{golden} are the golden edges and $\bigcup_{f \in \mathcal{F}} \mathbf{u}_f$ are the actions for features $\mathcal{F}$ labeling the golden edges.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 1: GUI Model Augmentation Algorithm

Input : $G = \langle V, E, r, A, \mathcal{L} \rangle$: Original GUI model of target app
$\mathcal{F}$: Feature library
Output: $G^+ = \langle V, E^+, r, A^+, \mathcal{L}^+ \rangle$: Augmented GUI model

```
1  begin
2      G+ ← G
3      foreach v ∈ V do
           // Iterate over each vertex (state) of G
4          foreach f ∈ F do
               // Iterate over each feature in F
5              dSet = destinationSet(v, G+, f)
6              foreach v1 ∈ dSet do
7                  e ← createEdge(v, v1);
8                  setEdgeLabel(e, getAction(f))
9                  markGolden(e)
10                 addEdge(e, G+)
11             end
12         end
13     end
14     return G+
15 end
```

Algorithm 1 shows the procedure to perform GUI model augmentation. The algorithm iterates over each state v in the GUI model (lines $3-13$) and each feature f in library $\mathcal{F}$, instantiating f at v, as per the feature definition. It computes the set of possible destination vertices $dSet$, by evaluating function D_f in the feature definition (function $destinationSet()$ on line 5). It then iterates over each possible destination vertex v_1 (lines $6-11$) creating and adding a golden edge, labeled by the feature's action sequence $\mathbf{u}_f$ (line 8), to the augmented model G^+. Figure 3 shows the GUI model of our simplified version of Kitchen Timer from Section 3, and Figure 4 shows the model augmented with golden edges for the Double Rotation and Back button features.

4.3. Test Suite Definition

Given an augmented GUI model G^+ the final phase of our technique generates a test suite with the following goal.

Test objective: *A **compact** suite of tests to **comprehensively** test the interaction features of the given app under test.*

A test is a sequence of actions (*i.e.,* a sequence of edges or a path in G^+) starting at the initial state r, with each action possibly followed by an oracle check. Therefore, a test can be represented as $\langle a_1, o_1, \ldots, a_n, o_n \rangle$. Each $a_i \in A^+$ is any action (including those that exercise interaction features)

Prepared using stvrauth.cls
DOI: 10.1002/stvr

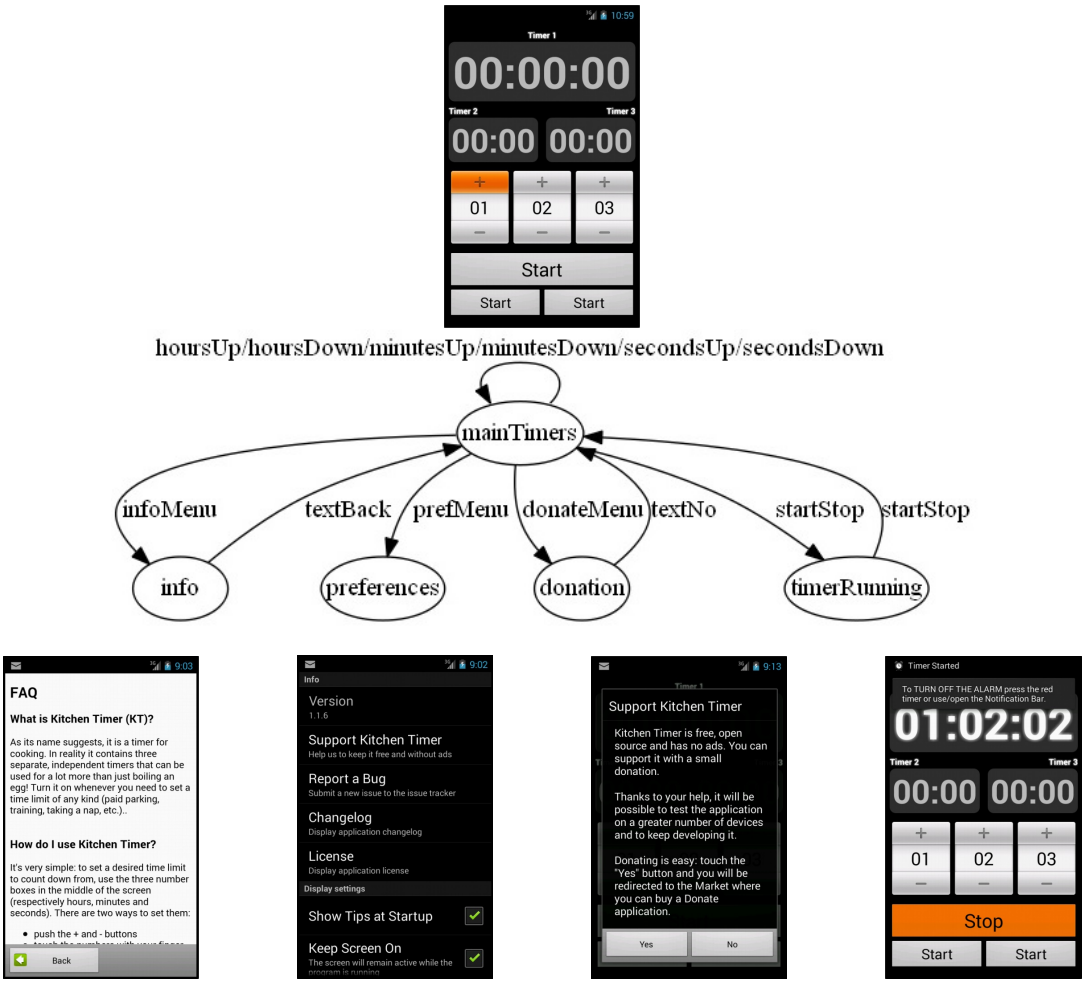


Figure 3. Simplified Model of Kitchen Timer.

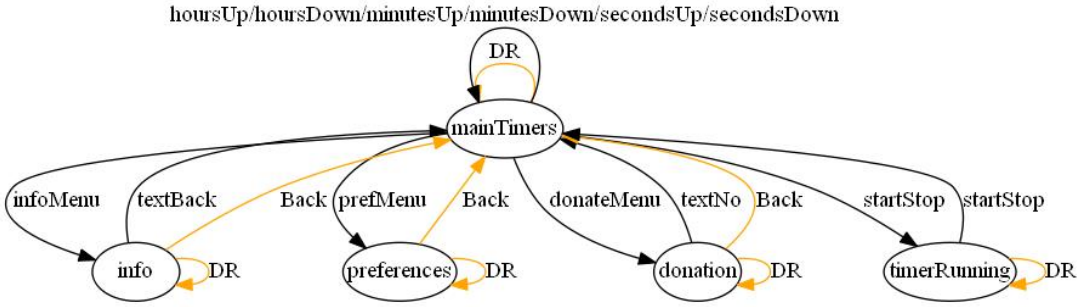


Figure 4. Augmented Model of Kitchen Timer.

allowed by the app's GUI. Each o_i is either an oracle check or no operation. We assume that oracle checks are side effect free, *i.e.,* they do not change the state of the app.

We define a test adequacy criterion to concretize the notion of a "comprehensive" test suite stated in the test objective. Since mobile apps are event-driven systems and interaction features

Prepared using stvrauth.cls
DOI: 10.1002/stvr

are elements of the app's GUI, we develop a criterion that is motivated by the notions of path coverage and event-flow coverage used by previous work on GUI testing [14]. This is in contrast to code coverage-based criteria such as line or branch coverage, which would be more appropriate for functional testing of the software implementation rather than testing its high level platform features, as in our case. Intuitively, we say that a test suite covers a feature, if it contains tests to exercise and validate *each possible instance* of exercising that feature on that app. Simply put, this implies exercising the feature in each GUI state. Given a test suite T, an interaction feature f from a feature library $\mathcal{F}$ and an augmented GUI model $G^+ = \langle V, E^+, r, A^+, \mathcal{L}^+ \rangle$, as defined in Section 4.2, we define adequacy of T in testing f with respect to G^+ as follows.

Definition 4 (Interaction feature coverage)

A test suite T covers a feature f iff $\forall s \in S : \exists t \in T, t = \langle a_1, o_1, \ldots, a_n, o_n \rangle, \exists j, k, 0 \leq j < k \leq n$ such that $\langle a_1, \ldots, a_j \rangle$ takes the app from the initial state s_0 to state s, $\langle a_{j+1}, \ldots, a_k \rangle = \mathbf{u}_f$, and $o_k = O_f$.

Since there are no standard or widely accepted cost functions to optimize test suites we quantify the "compactness" of our test suite using the common sense observation that large test suites are hard to set up, execute, and maintain. The size of a test suite can be measured by the number of tests it contains as well as the cumulative number of operations (actions a_i) in the test suite as a whole. We propose a customizable cost function that captures this.

Definition 5 (Cost of a test suite)

The cost of a test suite T is $cost(T) = \alpha * |T| + \beta * \Sigma_{t \in T} |t|$, where α and β are positive coefficients.

Coefficient α measures the relative cost of developing and maintaining a suite, which scales with the number of tests in a suite. Coefficient β quantifies the cost of executing actions and asserting oracles which is proportional to the number of operations.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

4.4. Feature-based Test Sequence Generation

Recall that interaction features are orthogonal to the core logic of the app and their function is typically to help the user navigate or access content on the app by mutating the state of the app's GUI. Further, exercising an interaction feature on a given state has no side effects in terms of the GUI model, *i.e.,* the effect of exercising that feature is limited to a single GUI screen and has no impact on the downstream actions. This observation is very important as it allows us to arbitrarily mix and match instances of several features (and their test oracles) in a single test case, as long as it lowers the cost of the test suite, per Definition 5. Since each instance of every feature is already recorded in our augmented GUI model G^+ (servicing the test adequacy criterion of Definition 4), our test generation problem can be stated as follows.

Test suite generation problem: *Given an augmented GUI model G^+ generate a minimum cost test suite such that each golden edge in G^+ is covered by at least one test in the suite.*

It can be shown that the above problem is NP-hard, by reducing the *minimum path cover problem* [15] to this problem. We omit the detailed proof here for lack of space.

We propose a greedy algorithm for this NP-hard problem. In addition, we introduce two optimizations to further reduce the cost associated with covering features. Algorithm 2 shows a pseudo-code of the traversal algorithm we propose. The input to this algorithm is the augmented graph model. We use a set to keep track of *covered edges* CE and a stack to record the test sequence. First, on Line 4, we sort the nodes based on their increasing distance from the root using a Breadth First Search (BFS) and keep the sorted list in L. For example, we can sort the nodes of Figure 4 as $\langle$*mainTimers*, *info*, *preferences*, *donation*, *timerRunning*$\rangle$. Then, working through the list L on Line 5, we select the next node s that has uncovered outgoing edges (Line 6). In our example, the first node in the list with uncovered outgoing edges is *mainTimers* (as we have not yet covered any edges). We use the shortest path from the root to this node (saved through previously performed BFS) as the prefix of all sequences to be generated starting from it. Lines 7 to 10 iterate through the shortest path and (1) mark edges as visited by adding them to CE, and (2) push them onto $stack$.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Algorithm 2: Traversal Algorithm

Input : $G^+ = \langle V, E^+, r, A^+, \mathcal{L}^+ \rangle$: Augmented GUI model of app
Output: T: Test Suite

```
1  begin
2      CE ← ∅
3      stack ← ∅
4      L ← sortWithBFS(G+)
5      foreach s ∈ L do
6          while ∃(s, y) ∈ outGoing(s), s.t.(s, y) ∈ E − CE do
7              foreach e ∈ shortestPathBFS(r, s) do
8                  stack.push(e)
9                  CE ← CE ∪ {e}
10             end
11             c ← s
12             stop ← false
13             while !stop do
14                 if ∃(c, v) ∈ outGoing(c), s.t.(c, v) ∈ E − CE then
15                     CE ← CE ∪ {(c, v)}
16                     stack.push((c, v))
17                     c ← v
18                 end
19                 else stop ← true
20
21             end
22             T ← T ∪ stack
23             stack.clear()
24         end
25     end
26     return T
27 end
```

The rationale behind using such a prefix is to minimize the cost associated with taking edges to get to a given node, where the exploration for uncovered golden edges begins.

Then, using c as a pointer to the *current* node, which is initially set to s, on Line 14 we pick an uncovered edge going out of c. We take this edge, mark it as covered (Line 15), push it onto $stack$ (Line 16), and update c to the destination of this edge accordingly (Line 17). Once we get to a node that has no uncovered outgoing edge, the current test sequence is complete and we set $stop$ to `True` on Line 19. The current $stack$ makes one test sequence and we continue by generating more sequences and adding them to T which is the test suite and is the output of this algorithm. For instance, the first test sequence that is generated is shown as T_0 under *No Optimization* in Table III. This Table displays the test suite our greedy algorithm generates for the simplified model of Kitchen Timer. The test suite has 7 tests at a total cost of 34, with α and β both set to 1 in the cost function.

We introduce two optimizations to augment our basic traversal algorithm. The first optimization called *prioritization*, prioritizes golden edges whenever there are both golden and regular (non-golden) uncovered edges going out of a node, since the goal of the traversal algorithm is to cover

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Test Sequences for Figure 4.

No Optimization	
T_0 = ⟨hoursUp, hoursDown, minutesUp, minutesDown, secondsUp, secondsDown, infoMenu, textBack, prefMenu, Back, donateMenu, textNo, startStop, startStop, DR⟩	
T_1 = ⟨infoMenu, Back⟩	T_2 = ⟨infoMenu, DR⟩
T_3 = ⟨prefMenu, DR⟩	T_4 = ⟨donateMenu, Back⟩
T_5 = ⟨donateMenu, DR⟩	T_6 = ⟨startStop, DR⟩
#Tests = 7, Cost(T) = 34	
Prioritization Optimization On	
T_0 = ⟨DR, hoursUp, hoursDown, minutesUp, minutesDown, secondsUp, secondsDown, infoMenu, Back, prefMenu, Back, donateMenu, Back, startStop, DR, startStop⟩	
T_1 = ⟨infoMenu, DR, textBack⟩	T_2 = ⟨prefMenu, DR⟩
T_3 = ⟨donateMenu, DR, textNo⟩	
#Tests = 4, Cost(T) = 28	
Prioritization and Truncation Optimizations On	
T_0 = ⟨DR, hoursUp, hoursDown, minutesUp, minutesDown, secondsUp, secondsDown, infoMenu, Back, prefMenu, Back, donateMenu, Back, startStop, DR⟩	
T_1 = ⟨infoMenu, DR⟩	T_2 = ⟨prefMenu, DR⟩
T_3 = ⟨donateMenu, DR⟩	
#Tests = 4, Cost(T) = 25	

golden edges. To implement this optimization, the method $outGoing()$ in Algorithm 2 returns golden edges first. Table III displays the output of the traversal algorithm with this optimization incorporated. For example, at the beginning of T_0 under *Prioritization Optimization On*, when the golden edge *DR* is available, it is taken before any other edge. This optimization makes the test suite smaller and decreases the cost from 34 to 28.

The second optimization, called *truncation*, uses the observation that a test can be truncated after the last golden edge it covers, and deleted if it covers no golden edges. Truncation is applicable in a post-processing phase on any test suite. Table III shows the result of combining both optimizations (applying truncation on the result of prioritization optimization) which makes the cost of the test suite go down to 25.

Once test sequences are generated, we insert oracles by augmenting test sequences in two ways. Firstly, we automatically add appropriate instrumentation before and after relevant actions in test sequences, to dynamically record the current view of each GUI state, as the test is being run.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Secondly, we automatically instantiate oracles O_f from the feature definitions to assert checks on the state views recorded by the instrumentation.

4.5. Implementation

The QUANTUM tool embodies our approach. QUANTUM currently supports testing of the following features[5]: rotation, killing and restarting, pausing and resuming, and Back button. There are four key steps in using QUANTUM.

Step 1: QUANTUM receives a (manually or automatically generated) model of the application's GUI as an XML file. QUANTUM automatically adds golden edges for the currently supported set of features. Then, QUANTUM generates a graphical representation of the GUI model using the dot program[6] so that the user can visually validate the model. Figure 4 is a sample graphical representation that QUANTUM generated.

Step 2: Once the model is validated, QUANTUM traverses the model using traversal algorithms to generate test suites. QUANTUM provides the following options for traversing the model: (1) our algorithm described in Section 4.4 and (2) a basic Depth First Search algorithm (DFS) that covers all edges to serve as a baseline for comparison. On top of our traversal algorithm, each of the optimizations can be turned on or off independently. By traversing the model, QUANTUM generates a suite of JUnit[7] tests. The tests use a combination of Robotium[8] and JUnit to interact with Android apps.

Step 3: In the generated test suite, QUANTUM automatically inserts (1) instrumentation to record views of states, and (2) oracles after exercising each golden edge. Recording views of states can be done through various user-interfaces provided by a mobile platform. We experimented with two interfaces from the Android platform: Hierarchy Viewer[9] and taking graphical snapshots.

[5]Zooming in and out functionality is currently unavailable in JUnit and Robotium frameworks, hence we did not include them in our tool.
[6]http://www.graphviz.org
[7]http://junit.org
[8]https://code.google.com/p/robotium
[9]http://developer.android.com/tools/help/hierarchy-viewer.html

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Hierarchy Viewer is a tool for debugging user-interfaces of apps that displays the hierarchy and properties of items on the screen. A programmatic interface is not available for Hierarchy Viewer to be used by tests, so we implemented one using Java reflection. The hierarchy and properties of items on the screen, provided by the state view, are then compared by oracles.

Graphical snapshots are taken from inside JUnit tests and are then compared using image processing. In the current implementation of QUANTUM, snapshots of the states are automatically recorded and the comparison is based on a basic image differencing algorithm that uses the Red-Green-Blue coloring system to compare images pixel by pixel and allows for an adjustable threshold of difference. Since the states are rendered on the same device and the same screen, it is conceivable that basic image comparison might be good enough. Indeed, taking graphical snapshots proved to be easier to use than Hierarchy Viewer for the currently implemented set of features, gave less false positives, and was faster.

Step 4: Now the test suite is complete and can be run to test the app running on an Android device or emulator. Each test case traverses and checks multiple golden edges. After executing each test, a log is provided which contains the result of checking each golden edge as Pass or Fail. In addition, QUANTUM takes snapshots of the app and provides them along with the expected snapshot for each failure. These snapshots facilitate identifying false positives, evaluating the severity of bugs, and debugging.

5. EVALUATION

We evaluated QUANTUM on 6 Android apps (3 apps from previously studied apps and 3 apps from the apps we selected, as discussed in Section 2) to answer the following research questions: (1) Can QUANTUM find bugs in real applications? (2) How effective is QUANTUM in terms of the ratio of real bugs to false positives (FP's)? (3) How compact are the test suites generated by QUANTUM?

DOI: 10.1002/stvr

Table IV. Bugs Automatically Found with QUANTUM.

Application	#Tests	#Assertions	#Failures	#FP's	#Bugs	#Distinct FP's	#Distinct Bugs
Notepad (Version N/A)	8	22	7	4	3	2	2
OpenSudoku (Version 1.1.5)	7	22	9	4	5	2	3
Nexes Manager (Version 2.1.8)	15	67	11	3	8	2	7
VuDroid (Version 1.4)	6	16	3	0	3	0	2
Kitchen Timer (Version 1.1.6)	8	37	13	5	8	2	4
K9Mail (Version 4.317)	16	53	8	1	7	1	4
Total	**60**	**217**	**51**	**17**	**34**	**9**	**22**

5.1. RQ 1 and 2: Finding Real Bugs

Given manually created GUI models, we used QUANTUM to automatically generate test suites. We first used our traversal algorithm with both optimizations, along with the image processing oracle. Then, we automatically executed the test suites on a rooted Android emulator (running Android 4.3 API level 18 with an Intel Atom (x86) CPU, 512 MB of SD card, and resolution WVGA800).

Table IV summarizes the results of finding bugs. *#Tests* is the size of the test suite generated for each app using our traversal algorithm with both truncation and prioritization optimizations. Each test covers several golden edges and tests multiple features, thereby generating compact test suites. *#Assertions* shows the total number of assertions in the test suite, which is equal to the number of golden edges. *#Failures* is the number of assertions that failed. We manually investigated the failures and identified real bugs and false positives. Some of these bugs or false positives were revealed more than once. Therefore, we show the number of distinct false positives and bugs in the last two columns.

QUANTUM found a total of 22 bugs in 6 apps. These bugs included 12 rotation bugs, 1 killing and restarting bug, 5 pausing and resuming bugs, and 4 Back button bugs. Examples of the bugs are as follows.

Pausing and resuming bug in K9Mail: The user finally finds an email after searching the inbox for some time, but while reading the email he receives a phone call (which pauses K9Mail). After

Prepared using stvrauth.cls
DOI: 10.1002/stvr

the phone call is over, K9Mail resumes, but back to the inbox, requiring to perform the search again.

Killing and restarting bug in K9Mail: The operating system decides to kill K9Mail because of low memory while the user is composing an email. K9Mail fails to save the email as a draft, deleting the contents of the email.

Rotation bug in Kitchen Timer: Explained in Section 3.

Rotation bug in OpenSudoku: Rotating the device closes the custom pop up for entering numbers and discards them.

Rotation bug in VuDroid: Rotation clears tab selection.

Rotation bug in Nexes Manager: If there is an empty folder which has no permission (read, write, etc.), rotating the device makes the folder icon disappear.

Back button bug in Kitchen Timer: Going to sub-menus and coming back makes buttons go out of focus.

We found two of these bugs already reported and accepted in the bug repositories of the corresponding apps. All the other bugs were new. We reported these to the respective developers and are awaiting confirmation of the bugs from them.

QUANTUM reported a total of 9 distinct false positives. However, 4 of these false positives were because of an inconsistency in the Android testing instrumentation, which caused it to act differently when paused programmatically (by tests) or through the emulator GUI (when manually confirming bugs). Another 2 false positives were because of time sensitivity of some app states. For instance, when a timer is running in Kitchen Timer, rotation changes timer values, not because there is a bug, but rather because the state of a running timer changes with time. Such time sensitivity is usually abstracted out from the app's GUI model to achieve conciseness. Another 2 false positives manifested because if the app GUI provides a visual back button on the screen, hitting this visual button and then the hardware Back button does not take the app to the original state. The remaining 1 false positive could be considered a bug, depending on the intent of the app designer.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table V. Compactness of Generated Test Suites.

	Our Algorithm								DFS	
	Basic		+ Trunc.		+ Prior.		+ Both			
Application	**#Tests**	**Cost**	**#Tests**	**Cost**	**#Tests**	**Cost**	**#Tests**	**Cost**	**#Tests**	**Cost**
Notepad	15	73	11	59	12	65	8	44	35	114
OpenSudoku	13	85	10	67	9	67	7	51	30	138
Nexes Manager	38	200	24	149	26	174	15	127	97	354
VuDroid	8	41	7	38	7	40	6	36	17	57
Kitchen Timer	14	113	14	113	11	105	8	75	70	310
K9Mail	30	228	25	196	21	187	16	148	76	490

5.2. RQ 3: Compactness of Generated Test Suites

We compared our test generation algorithm to a baseline DFS, in generating test cases that cover all golden edges. For the set of implemented features (rotation, killing and restarting, pausing and resuming, and Back button) we measured the compactness of generated test suites in terms of the number of tests and the cost of the test suite when generated by each of the following algorithms: (1) Our basic algorithm; (2) Our algorithm plus the truncation optimization, which truncates test cases after the last golden edge and ignores test cases that do not cover any golden edge; (3) Our algorithm plus the prioritization optimization, which prioritizes golden edges while traversing the model; (4) Our algorithm plus both truncation and prioritization optimizations; and (5) A DFS algorithm starting at the root.

Table V shows the experimental results. The cost function is calculated with both α and β set to 1. As this table shows, our algorithm shows a clear improvement over DFS, in terms of the number of tests as well as the cost of executing test suites. Furthermore, truncation and prioritization improve the results when applied separately (except for Kitchen Timer, for which truncation does not improve the number of tests or cost). In some cases (Notepad, Nexes Manager, and VuDroid), truncation yields better results compared to prioritization, while in the other cases prioritization is more effective. Fortunately, the optimizations are compatible and combinable and using both of them produces even better results for all of the studied apps.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Threats to Validity: To minimize threats to internal validity, we automated the entire test generation and execution process and manually identified real bugs from false positives. To address external validity, we experimented with 7 previously studied apps and set forth a criterion to choose 6 other popular apps from open source repositories as discussed in Section 2. With respect to construct validity, we strictly followed our traversal algorithm and oracle generation techniques, used well-known frameworks Robotium and JUnit, and manually investigated generated tests for some of the apps.

6. RELATED WORK

Our work attempts to address the classic oracle problem [7, 8], in the context of mobile apps. In practice, test oracles are typically specified manually, often at the expense of substantial time and effort. There is a rich body of work that aims to alleviate this long-standing problem by automatically generating oracles. Software specification mining or model inference techniques are often used for this purpose. A comprehensive survey of API property inference techniques by Robillard et al. describes many of these techniques [16]. Automated oracle generation techniques usually generate general purpose oracles for functional testing; they are not specific to any platform or class of software applications or any aspects of software behavior. However, a recent empirical study by Nguyen et al. on the cost and effectiveness of automated oracles concludes that their false positive rate is often prohibitively high for practical use [17].

Our proposed technique does not automatically generate general purpose oracles but rather falls into a related body of work that uses manually created oracles based on domain specific knowledge, that are appropriately *instantiated* during testing and used to test very specific, sometimes non-functional, aspects of software behavior. For example, the TODDLER tool uses a hand-crafted oracle that detects repetitive memory access patterns in loops to identify performance bugs in software [18]. Our previous work used differential testing [19] for oracle automation in the context of testing web browser implementations [20] as well as detecting cross-browser errors in web applications [21],

Prepared using stvrauth.cls
DOI: 10.1002/stvr

i.e., discrepancies in web application behavior across different web browsers, using test oracles specifically designed for these domain-specific applications of differential testing. Our work in this paper exploits characteristics of mobile apps and the mobile platform to design oracles for testing an important class of user-interface features of mobile apps. Hu et al. also employ a specialized oracle for testing Android mobile apps, which implements and checks the Activity life-cycle specification[10] for Android apps [11]. However, this is one single oracle, whereas our approach proposes an extensible framework that spans a whole class of properties – user-interaction features.

There is a growing body of research focused on automated testing of mobile applications. The proposed techniques span the complete gamut of technologies from random testing [11, 12], to symbolic-execution-based test-case generation [3, 22], model-based testing, combinatorial testing, and combinations thereof [2, 6]. However, the emphasis here is on generating test sequences to maximize code coverage, for the purpose of functional testing. The oracle problem is not directly addressed in these papers. It is implicit that the oracles would either be manually specified or use the simple oracle corresponding to catastrophic failure when the application crashes, hangs or otherwise throws an exception. By contrast, the focus of our approach is precisely to address the oracle problem, for a class of non-functional and platform-specific features of mobile apps.

Our approach to test sequence generation falls under the broad area of model-based testing. The model may be manually specified or automatically extracted from the application under test. In fact there is a rich and active body of work on reverse-engineering such models from the user-interface of GUI applications [23], web applications [24], and more recently, mobile applications [4, 5, 13]. However, our approach is independent of the method used to produce the model and is therefore orthogonal to these techniques.

The aim of model-based test sequence generation is to extract a suite of concrete test cases based on the behavior represented in the model. Most techniques in this category do this by heuristically solving some variant of the NP-Hard *minimum path cover problem* [15], typically guided by some supporting analysis and test suite sufficiency criteria. Memon et al. propose several test adequacy

[10] http://developer.android.com/training/basics/activity-lifecycle/index.html

Prepared using stvrauth.cls
DOI: 10.1002/stvr

criteria for GUI testing based on coverage of events and event-sequences in the GUI model [14]. Arlt et al. use a lightweight static analysis to compute data dependencies between event-handlers of a GUI application and use that to guide the choice of test sequences from the GUI model [25]. Ganov et al., on the other hand, focus on the problem of generating suitable values for the input parameters of abstract test sequences extracted from a GUI model and employ symbolic execution to compute these parameter values [26]. Nguyen et al. address the same problem by using combinatorial testing techniques to embed user-specified data values into abstract test sequences [2]. The aim of all the above techniques is functional testing of the application and, more specifically, to extract test cases which maximize coverage of the application code. By contrast, our test sequence generation is intended to exhaustively exercise a set of platform-specific user-interaction features. This leads to different test targets, cost-functions and ultimately a different set of model traversal algorithms than those by pure functional testing approaches.

7. CONCLUSION

In this paper we presented a novel approach for automatically generating test cases, complete with test oracles, for mobile apps. It was motivated by a comprehensive study that we conducted of real defects in mobile apps. Through this study we identified a class of features called user-interaction features, which were implicated in a significant fraction of bugs and for which oracles could be constructed, in an application agnostic manner, based on our common understanding of how apps behave. Our approach, as embodied by our tool QUANTUM, includes an extensible framework that supports such domain specific, yet application agnostic, test oracles, and automatically generates a compact suite of test sequences, including test oracles, to comprehensively test the user-interaction features of a given mobile app. Our initial experimental evaluation of QUANTUM on 6 open-source Android apps was quite promising: QUANTUM found a total of 22 bugs, a few of them particularly serious, using a total of 60 tests for these 6 apps. For future work we would like to augment the set of oracles currently supported by QUANTUM, evaluate it more extensively on a larger set of

Prepared using stvrauth.cls
DOI: 10.1002/stvr

apps, and explore the possibility of extending our basic oracle generation approach beyond the set of user-interaction features reported in this work.

ACKNOWLEDGEMENT

We thank Guowei Yang for detailed discussions and comments on this work.

REFERENCES

1. Williamson L. A mobile application development primer: A guide for enterprise teams working on mobile application projects. IBM Software Thought Leadership White Paper 2013.
2. Nguyen CD, Marchetto A, Tonella P. Combining model-based and combinatorial testing for effective test case generation. *ISSTA*, 2012.
3. Anand S, Naik M, Harrold MJ, Yang H. Automated concolic testing of smartphone apps. *FSE*, 2012.
4. Amalfitano D, Fasolino AR, Tramontana P, Carmine SD, Memon AM. Using GUI ripping for automated testing of Android applications. *ASE*, 2012.
5. Joorabchi ME, Mesbah A. Reverse engineering iOS mobile applications. *WCRE*, 2012.
6. Jensen CS, Prasad MR, Møller A. Automated testing with targeted event sequence generation. *ISSTA*, 2013.
7. Howden WE, Miller E. *Introduction to the Theory of Testing*. 1978.
8. Weyuker E. The oracle assumption of program testing. *ICSS*, 1980.
9. Pezzè M, Young M. *Software testing and analysis - process, principles and techniques*. 2007.
10. Goodenough J, Gerhart S. Toward a theory of test data selection. *TSE* 1975; (2):156–173.
11. Hu C, Neamtiu I. Automating GUI testing for Android applications. *AST*, 2011.
12. Machiry A, Tahiliani R, Naik M. Dynodroid: An input generation system for android apps. *FSE*, 2013.
13. Yang W, Prasad MR, Xie T. A grey-box approach for automated GUI-model generation of mobile applications. *FASE*, 2013.
14. Memon AM, Soffa ML, Pollack ME. Coverage criteria for GUI testing. *FSE*, 2001.
15. Ntafos SC, Hakimi SL. On path cover problems in digraphs and applications to program testing. *TSE* 1979; **5**(5):520–529.
16. Robillard MP, Bodden E, Kawrykow D, Mezini M, Ratchford T. Automated API property inference techniques. *TSE* 2013; **39**(5):613–637.
17. Nguyen CD, Marchetto A, Tonella P. Automated oracles: an empirical study on cost and effectiveness. *FSE*, 2013.
18. Nistor A, Song L, Marinov D, Lu S. Toddler: detecting performance problems via similar memory-access patterns. *ICSE*, 2013.
19. McKeeman W. Differential testing for software. *Digital Technical Journal* 1998; **10**(1):100–107.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

20. Zaeem RN, Khurshid S. Test input generation using dynamic programming. *FSE*, 2012.
21. Choudhary SR, Prasad M, Orso A. X-PERT: Accurate identification of cross-browser issues in web applications. *ICSE*, 2013.
22. Mirzaei N, Malek S, Pasareanu CS, Esfahani N, Mahmood R. Testing Android apps through symbolic execution. *Software Engineering Notes* 2012; **37**(6):1–5.
23. Memon AM, Banerjee I, Nagarajan A. GUI ripping: Reverse engineering of graphical user interfaces for testing. *WCRE*, 2003.
24. Mesbah A, van Deursen A, Lenselink S. Crawling ajax-based web applications through dynamic analysis of user interface state changes. *TWEB* 2012; **6**(1):3:1–3:30.
25. Arlt S, Podelski A, Bertolini C, Schäf M, Banerjee I, Memon AM. Lightweight static analysis for GUI testing. *ISSRE*, 2012.
26. Ganov SR, Killmar C, Khurshid S, Perry DE. Event listener analysis and symbolic execution for testing GUI applications. *ICFEM*, 2009.

DOI: 10.1002/stvr