國立交通大學

網路工程研究所

碩士論文

嵌入式裝置自動測試機制之準確度、效率與重複使用度研究

On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices

研究生：Jose Francisco Rojas Soto

指導教授：林盈達　教授

中華民國一百零二年六月

嵌入式裝置自動測試機制之準確度、負擔與重複使用度研究

On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices

研 究 生：羅荷西　　Student: Jose Francisco Rojas Soto

指導教授：林盈達　　Advisor: Dr. Ying-Dar Lin

國立交通大學

網路工程研究所

碩士論文

A Thesis

Submitted to Institute of Network Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

In

Computer Science

June 2013

HsinChu, Taiwan

中華民國一百零二年六月

嵌入式裝置自動測試機制之準確度、負擔與重複使用度研究

學生: 羅荷西 **指導教授: 林盈達**

國立交通大學網路工程研究所

摘要

自動化圖形使用者界面測試透過模擬使用者事件以及自動驗證圖形用戶界面的變化以確認 Android 應用程式是否符合規格。傳統的 record-replay 測試工具能自動產生測試項目，但不會自動驗證測試的結果。本篇論文實作的工具 SPAG-C (Smart Phone Automated GUI testing tool with Camera) 是延續以前論文所做的 SPAG，除了能節省測試所需的時間外，能在不損失測試準確性的前提下，提高 test oracle 的重複使用性。在 record 階段，SPAG 檢查從手機 frame buffer 獲得的畫面以及將驗證命令加入測試項目。與 SPAG 不同的地方在於，SPAG-C 使用外部的視訊鏡頭捕捉手機畫面。在 replay 階段，SPAG-C 驗證測試結果的方式為自動比較圖片，但 SPAG 只會比較字串。而且為了使 SPAG-C 能重複使用於不同的設備以及保有更好的圖片捕捉同步性，我們開發了一個使用外部的視訊鏡頭和 Web 服務的新架構去搭配 test oracle。實驗結果顯示，用 SPAG-C 記錄測試項目的速度跟 SPAG 一樣快，但測試項目卻更加準確。而在保有相同的測試項目準確性下，SPAG-C 記錄測試項目的速度比 SPAG 快 50%至 75%。因為 SPAG-C 針對不同手機的重複使用性，SPAG-C 能將測試時間從幾天縮短到幾小時。

關鍵字: 自動化測試, 圖形用戶界面, Android, test oracle

On the Accuracy, Efficiency, and Reusability of Automated Test Oracles for Android Devices

Student: Jose Francisco Rojas Soto **Advisor: Dr. Ying-Dar Lin**

Department of Computer Science
National Chiao Tung University

Abstract

Automated GUI testing consists of simulating user events and validating the changes in the GUI in order to determine if an Android application meets specifications. Traditionally, record-replay testing tools facilitate the test case writing process but not the verification process. In this work, we develop SPAG-C (Smart Phone Automated GUI testing tool with Camera), an extension of our previous work called SPAG, to reduce the time required to record test cases and increase reusability of the test oracle without compromising test accuracy. In the record stage, SPAG captures screenshots from device's frame buffer and write verification commands into the test case. Unlike SPAG, SPAG-C captures the screenshots from an external camera instead of frame buffer. In the replay stage, SPAG-C automatically performs image comparison while SPAG simply performs a string comparison to verify the test results. In order to make SPAG-C reusable for different devices and allowing better synchronization at the time of capturing images, we develop a new architecture that uses an external camera and Web services to decouple the test oracle. Our experiments show that recording a test case using SPAG-C's automatic verification is as fast as SPAG's but more accurate. In particular, SPAG-C is 50% to 75% faster than SPAG in achieving the same test accuracy. With reusability, SPAG-C reduces the testing time from days to hours for heterogeneous devices.

Keywords: Automatic testing, GUI, Android, test oracle

Acknowledgement

I would like to thank my family for all the love and support they have always given me: to my mother for her love, patience and kindness; to my father for inculcating in me the habit of reading and expanding my mind with new ideas; and to my sister for sharing her love, happiness and enthusiasm with me.

I would also like to thank my good friend Alina and her family for welcoming me into their home and treating me like family.

Last but not least, I would like to thank my advisor, Dr. Ying-Dar Lin, for his patience and guidance during these past 2 years, especially for all his efforts to inculcate in us the habits of time management and work efficiency; Professor Edward Chu for his feedback and help in revising my writing; and Professor Yuan-Cheng Lai for his valuable suggestions on how to improve my research.

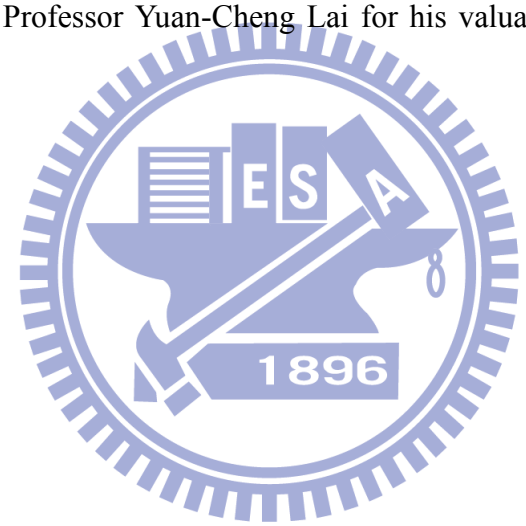


Table of Contents

List of Tables

List of Figures

Chapter 1 Introduction

Automated GUI testing tools aim to test graphical user interfaces while reducing as much as possible the manual work done by testers.

There are two fundamental tasks in automated GUI testing. First, simulating user events, and second, verifying that the application is behaving as expected. More specifically, an automated testing tool executes a given set of tests on an application under test (AUT) and verifies its behavior using a test oracle [3]. The test cases have all the information required to simulate user events on the AUT, while test oracles have the mechanisms to capture the *current state* of the GUI during the testing process and compare it with the corresponding *expected state,* which is usually given before the execution of the test case. In order to make GUI testing tools publicly or commercially available, three major issues must be addressed: reusability, efficiency, and accuracy.

For reusability, most of the time GUI testers would want to run a test case several times under different conditions and devices to see how the AUT responds; thus, the degree of reusability of test cases and test tools is crucial. In addition, reducing testing time is an important goal of automated GUI testing tools. This may be accomplished by automatically running test cases and automatically verifying GUI states. Finally, an automated GUI testing tool must be able to accurately tell whether the AUT is behaving as expected or not. It means that a low percentage of *false positives* and *false negatives* is desirable. Improving accuracy, however, often conflicts with reusability because the GUI states captured while executing the tests may present expected changes, thus in order to use the same test several times we must check for similarity, not exact matches. However, checking for similarity means that it is hard to detect small errors. Therefore, there is a tradeoff between accuracy and reusability.

Related works and SPAG

Several methods have been proposed to address the issues of accuracy, efficiency, and reusability. Different automation approaches address these issues in different ways. For example, “model-based” testing [4, 5] aims to automate as much work as possible by automatically generating test cases and verifying the GUI state. However, the great amount of *possible combinations* regarding which actions can be performed on a GUI means that this process may take days, weeks, even months to fully test an application depending on how complex its GUI is. Using “accessibility technologies” to get *programming access* to *GUI objects* of the AUT [7] is another approach that has been suggested recently. Although, this method also works for black box testing, it is limited by many factors, such as the system’s

API, security restrictions and the information made available by developers of the application through accessibility technologies. Finally, another commonly-used automation technique is "record-replay" [6, 8]. This technique allows testers to "record" test cases without the need of writing code, only by performing *GUI actions* directly on the application, while a tool automatically creates the test case. Afterwards, testers can "replay" these test cases any number of times. Nevertheless, the record-replay technique still requires testers to record the test cases and provide the expected states to verify the GUI.

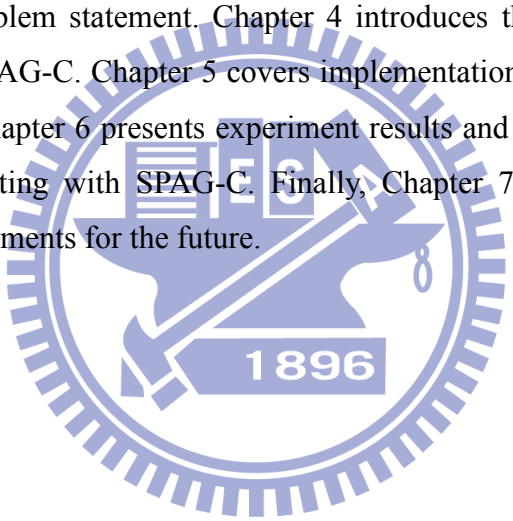
One related record-replay work is our previous work SPAG (Smart Phone Automated GUI testing tool) [6]. SPAG combines Sikuli [9] (an automation tool for computers that makes extensive use of computer vision techniques) and Android Screencast [10] (a remote control tool for Android devices) to perform automated GUI testing on Android devices. Since both tools, Sikuli and Android Screencast, are open-source, SPAG also improves them by adding some functionality that helps automate the testing process even further on Android devices.

SPAG-C

This work (SPAG-C) is an extension of SPAG. It aims to improve SPAG by enhancing testing efficiency and increasing reusability without compromising accuracy. In order to achieve our goals, we develop a different architecture. Usually, testing tools are *platform-dependent* even though the verification process is the same regardless of the device under test (DUT), which means the test oracle might *not* be reusable. Traditionally, there have been two ways to verify the state of an application's GUI: image comparison and object identification. This work uses image comparison because it allows quickly verifying an application's GUI without having the source code (black box testing) and its system independent which allow us to design an approach that can be used in a wider range of devices. But, Instead of capturing the required screenshots from within the device as is normally done, we use an external camera. Using an external camera makes the verification component *platform-independent* and *offloads* some processing from the DUT. Furthermore, we use Web service technologies [26] to expose the verification component to the record-replay component. It means that the test oracle is not only platform-independent but also *independent* from the record-replay component because now it is accessed via Web services, which means it can be accessed by different testing tools thanks to the interoperability provided by Web service standards [21]. Finally, we also propose a method to automatically *derive* the expected states of an application's GUI during the record process, which reduces the time required to record test cases.

In order to evaluate the performance of the proposed method, we conducted several experiments on two different devices: Acer Liquid and LG-P920. Five popular mobile applications, Alarm Clock, Android Market, Calculator, Contacts and Google Maps, were used as benchmarks. By using heterogeneous devices and applications, we demonstrate the reusability of our test oracle; moreover, we show how our Web service API may allow other tools to reuse our test oracle. We also show how automatically deriving the expected states during the record process can reduce test case writing time, and why our approach is better than SPAG's automatic verification. Finally, in order to demonstrate the accuracy of our approach we conducted a series of experiments to measure the percentage of false positives and false negatives.

This work is organized as follows. Chapter 2 covers common techniques and tools used in automated GUI testing. Chapter 3 provides some definitions that will be used throughout this work and defines our problem statement. Chapter 4 introduces the architecture, algorithms and techniques used by SPAG-C. Chapter 5 covers implementation details of the components described in Chapter 4. Chapter 6 presents experiment results and important observations we obtained while experimenting with SPAG-C. Finally, Chapter 7 concludes this work and proposes potential improvements for the future.



Chapter 2 Background and Related Works

In this chapter we first provide background knowledge of image comparison methods used in automated GUI testing. We then present a survey of related work on automated GUI testing techniques and Android-based automated GUI testing tools.

2.1 Image comparison techniques

Histogram comparison

A color histogram is a representation of the color distribution of an image (i.e., the number of pixels of an image with a given color). Color histogram comparison extracts and compares the color histogram of two images. If both histograms are similar, then the images are considered to be similar. Although, this is an efficient technique to compare images, it is sensitive to changes in lighting conditions and not aware of the contents of an image. In other words, two completely different images will be considered as having similar contents if they have similar histograms [11].

SURF: Speeded Up Robust Features

SURF is a "scale- and rotation-invariant *interest point* detector and descriptor" [12]. In computer vision, an interest point detector is used to detect parts of an image that can be used to uniquely describe it. An interest point, also called *feature* or *key point*, has many properties but the most important one is its *repeatability*, which means that it must be reliably computed under different conditions (e.g., changes in size, rotation, etc.). After an interest point of an image has been detected, the interest point *descriptor*, a feature vector that represents this neighborhood, uses the neighborhood information of the interest point to characterize it. By adopting the characteristics of interest points, SURF-based image comparison method first extracts the interest points of the two images being compared. It then *matches* descriptors of both images. Finally, image similarity is measured according to the *amount* of matches.

Template matching

Template matching is a method used to find a small image (template) in a larger image (source). This is done by taking the template and *sliding* it on the original image pixel by pixel; at every point a metric is calculated to determine how good the match is. After all metrics are calculated, the best match can be selected. Depending on the method used, the best match may be the highest or the lowest calculated value [13].

The image comparison techniques described above have proved to be accurate and efficient but not perfect, each one of them have some weaknesses. The Table 1 compares these

three techniques in terms of invariance to size and rotation, sensitivity to noise, and performance.

Table 1: Image comparison techniques

Technique	Size-invariant	Rotation-invariant	Noise sensitivity	Performance
Histogram[11]	yes	Yes	high	high
SURF[12]	yes	Yes	low	low
Template Match[13]	no	No	high	high

We have chosen to use these techniques because they allow some degree of flexibility while still being accurate. The reason we use all three techniques and not just one is because according to our experiments they *complement* each other; that is, errors undetected by one technique are likely to be detected by any of the other two.

2.2 Automated GUI testing techniques

A lot of effort has been dedicated on the area of automated testing. In this section, we introduce several commonly-used automated GUI testing techniques.

Model-based testing

In recent years an increasing amount of research effort has been dedicated to model-based testing. Model-based testing represents the system under test (SUT) as a model. A model is a detailed abstract description of how the SUT is supposed to work. The behavior of the SUT can be represented in many ways but it is usually represented as a *state machine*, and graph algorithms are used to automatically *derive* the test cases [14].

Time consumption is the major problem of model-based testing because the great amount of possible test cases can be automatically generated. However, many of them are *irrelevant*. Furthermore, in order to create a model, it is required to have detailed documentation of the SUT, and keep that documentation and the model updated; otherwise, invalid tests will be generated. By using the record-replay technique, SPAG-C takes advantage of the knowledge testers have about how the application is supposed to work and the context in which it is used. Therefore, testers can always create *relevant* test cases and SPAG-C verifies *only* what is necessary.

Accessibility technologies

Although the real purpose of accessibility technologies is to facilitate the use of technology for disabled people, many researchers are making use of it to access GUI

information, and thus, verify the GUI state of an application. For example, Grechanik *et al*. [7] made use of Windows accessibility technologies to create *hooks* that listen to events generated by the GAP (GUI-based application). Whenever an event of interest is triggered these hooks can react to it and perform some operations like *gathering* the properties of the elements currently being displayed and *verifying* its state. Although, this is a valid approach for black box testing, it is still limited by not only the system's API but also the way developers make use of it. Android automatically makes applications more accessible but there are some steps that developers can take to provide extra information about the application. SPAG-C, on the other hand, does not depend on accessibility services to verify an application's GUI.

Record-replay

Record-replay is the most popular approach for GUI test automation [20] because it allows creating test cases for an application without the need of writing code. The testing process basically consists of two phases: the record phase and the replay phase. During the record phase, testers interact with the AUT in exactly the same way *end users* would do it. While a tester is interacting with the AUT, a tool is automatically recording all input events and writing them into a test case. Later, testers can modify the recorded test cases if required and replay them at any time. In order to improve over the traditional record-replay approach we propose a method to automatically *capture* the required images and add the *verification* commands during the record process. This way all testers have to do is record the test cases.

2.3 Android-based automated GUI testing tools

Monkeyrunner

Monkeyrunner [15] is a testing tool provided by Google. It provides an API that developers can use to control Android devices without the need of any source code. To use Monkeyrunner, developers write Python programs to simulate user interaction. If they want to corroborate the state of the GUI, they can also write commands to capture screenshots from within the devices using Android's *frame buffer* which is the part of video memory containing the current video frame. There are three main issues with Monkeyrunner apart from the fact that in order to use it testers need *programming* skills: first, the naïve form in which it simulates events on the AUT [6]; second, its verification approach; third, capturing screenshots from Android's frame buffer is *time-sensitive*, which means that testers need to adequately *synchronize* the simulation of events with the time of the capture or otherwise invalid images will be taken for verification. On the contrary, SPAG-C takes advantage of the

method used by SPAG to *accurately* simulate events on the DUT, and uses a *non-intrusive* method to capture images which is automatically *synchronized* with the simulated events at all times.

Robotium framework

Robotium [16] is a framework used to perform black box testing on Android devices. It uses Android Instrumentation [18] to interact with an application's GUI and gather some information. In order to check the state of an application, screenshots can be taken or object identification can be performed using Robotium's API and JUnit's assertions. Robotium is widely used but just like Monkeyrunner it requires testers to *manually program* test cases. SPAG-C automatically creates test cases by listening to user events and recording them in the test case which reduces the test writing time considerably.

Testdroid

Testdroid [8] is an Android testing platform that uses the Robotium framework to define test cases. Testdroid records user interaction and automatically generates Java code with calls to Robotium API. These test cases can be later replayed at any time in the same way that Robotium tests are executed. With Testdroid, testers can execute their tests either locally, on their own devices, or remotely, using Testdroid's cloud services. Testdroid's cloud services provide log files and statistics about test execution; additionally, it takes screenshots during the testing process so developers can verify the GUI. Testdroid services, however, are quite expensive, and GUI verification has to be done *manually* by the testers since Testdroid does not perform any comparison against expected states. On the contrary, SPAG-C completely automates the verification process so that all testers need to do is to record the tests.

GUITAR

Android GUITAR (Graphical User Interface Testing frAmewoRk) [17] was an effort of Xie and Memon to migrate their previous work [4] on model-based testing to the Android platform. GUITAR consists of two modules: ripper and replayer. The ripper is in charge of automatically generating event-flow graphs for their later conversion into test cases. It does this by automatically interacting with an application and gathering all relevant information about its GUI. Since the GUI ripper cannot be guaranteed to have access to all different windows and widgets of an application, a capture/replay tool was created for testers to complement the ripper. The replayer is in charge of the execution of the generated test cases. The main problem with GUITAR is that only works on Android *emulator* because it uses *Hierarchy Viewer* [27] a tool that only works on development builds, not on real devices. On

the other hand, SPAG-C can be used on a great variety of real devices, and its test oracle can be reused by different testing tools.

2.4 SPAG

This work, SPAG-C, is an extension of a previous work called SPAG (Smart Phone Automated GUI testing tool) [6]. SPAG combines and extends two open source tools: Sikuli [9] and Android Screencast [10]. SPAG merges these two tools together to be able to use Sikuli's API for testing Android devices. SPAG *intercepts* user interactions with Android Screencast, saves these interactions in a Sikuli *test file* and replays them later as required by the tester.

SPAG provides three contributions: *Batch event*, which is used to accurately reproduce the recorded event sequences; *Smart wait*, which is used to automatically establish a delay between events to ensure that the DUT has enough time to process previous events; and an *automatic verification* method, which makes use of Android accessibility services to record transition between activities after an event is executed.

Since SPAG is integrated with Sikuli, it can also take advantage of Sikuli's API to perform image verification in a *semi-automatic* way, which means that the verification is done by Sikuli but the tester still needs to provide the images and write the commands into the test case. SPAG also provides an *automatic* verification that uses Android Accessibility services to gather the name of the activities, and performs a string comparison to verify that the same activity transition that occurred after the input of a specific event during record also happens during replay. This, however, does not ensure that applications are being displayed as expected. SPAG-C also provides two verification approaches: *semi-automatic* and *automatic.* In both approaches SPAG-C performs image verification with images captured from a camera, the only difference is that the *semi-automatic* approach requires testers to capture the images, while *automatic* approach does not.

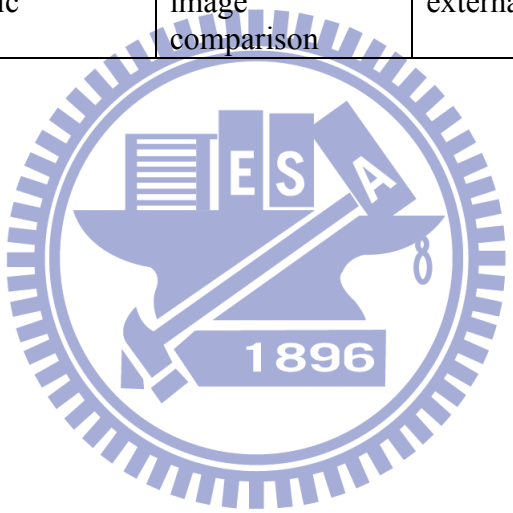
SPAG depends on Android Screencast to interact with the DUT; therefore, it inherits its limitations like limited support for devices, slow response time, which may affect the image verification process, and the inability to reproduce multi-touch events. Since SPAG-C is based on SPAG it also inherits some of SPAG's limitations but we improve the verification process by making it more reusable, automated, better synchronized and platform-independent.

In Table 2 we compare the previously discussed tools by looking at: how often the GUI is verified during test execution (verification frequency), what technique is used to verify the GUI (verification approach), how screenshots are taken (capture screen method), and what

testing technique is used.

Table 2: Common automation tools in Android

Tool	Verification frequency	Verification approach	Capture screen method	Testing Technique
Monkey runner[15]	up to tester	image comparison	frame buffer	test script
Robotium[16]	up to tester	code assertions	no need	test script
GUITAR[17]	--	code assertions	no need	model-based
Testdroid[8]	--	--	frame buffer	record-replay
SPAG semi[6]	up to tester	image comparison	frame buffer	record-replay
SPAG auto	activity transition	string comparison	no need	record-replay
SPAG-C semi	up to tester	image comparison	external camera	record-replay
SPAG-C auto	dynamic	image comparison	external camera	record-replay



Chapter 3 Definitions and Problem Statement

In this chapter we define some terminologies that will be used throughout the rest of this work and formally define our problem.

3.1 Definitions

Since this work focuses on GUI testing, the term "application" implies that an application has a graphical user interface. Android applications are formed by one or more of the following components: Activities, Services, Content providers and Broadcast receivers [19]. However, since only the "Activity" component is GUI-related we shall not cover the other three.

An *activity* is the basic container of an application's GUI. It represents a single screen where *GUI elements* are drawn. GUI elements are the basic components of an application's GUI, including button, checkbox, textbox, etc. GUI elements have properties like color, size and coordinates. Technically, in Android, a GUI element is called a *widget;* here however, we use the term GUI element to avoid confusion with *application widgets* which are small "previews" of applications that are displayed in the home page of an Android device. In addition to properties, GUI elements also have behaviors. They respond to different user events. A *user event* or *GUI event* is an event triggered by the user's interaction with an application's GUI. Examples of user events are click, long click, and key input, etc. An *event sequence,* therefore, is one or more user events executed in succession.

User interaction usually changes an application's appearance by making a transition between different activities or by modifying the appearance of the current activity. We define a *GUI state* of an application as the set of all GUI elements being displayed on the screen at a specific time. Transitions from one GUI state to another are triggered by user events. In this work, a GUI state is represented by a screenshot of the whole screen of the device.

In order to verify if an application is behaving correctly, assertions have to be performed during the replay phase to compare the *current states* (captured during the replay phase) against their corresponding *expected states* (captured during the record phase). A *test oracle* is the mechanism in charge of capturing both states and performing such assertions.

As mentioned before, test cases contain all the required information to simulate user events and verify the correctness of an application. A *test case* represents a scenario used by testers to evaluate if an application behaves as expected. A test case is a script file that contains instructions to simulate user events on an AUT and verification commands to verify

the GUI states. There are two kinds of test cases: positive and negative. A positive test case expects an AUT to allow the execution of a *normal* sequence of events. A negative test case expects an AUT to prohibit the execution of an *illegal* sequence of events or to trigger an error when such a sequence is executed. For example, an application should allow submitting a form after filling it with the right information, and display an error if some required information is not provided. A positive test case would test the former condition while a negative test case would test the latter.

Table 3: Notations and definitions used in this work

Notations	**Definitions**
$T = \{ t_j \mid j = 1 \ldots M \}$	a set of M test cases associated with an application A, where t_j is the *j-th* test case which has a sequence of L_j user events
$E = \{e_{j_p} \mid p = 1, 2, \ldots, L_j\}$	a sequence of L_j user events, where e_{j_p} is the *p-th* user event of the *j-th* test case
$D = \{ d_i \mid i = 1 \ldots N \}$	a set of N heterogeneous DUTs, where d_i is the *i-th* device
$S = \{s_{j_p}\}$	a set of screenshots that represent the expected GUI states of A, where s_{j_p} is a screenshot taken after execution of the user event e_{j_p} during the **record** phase
$S' = \{s'_{j_p}\}$	a set of screenshots that represent the current GUI states of A, where s'_{j_p} is a screenshot taken after execution of the user event e_{j_p} during the **replay** phase
s_{j_i} , s_{j_f}	initial and final **expected** states respectively, they belong to set S
s'_{j_i} , s'_{j_f}	initial and final **current** states respectively, they belong to set S'
c_{j_p}	a screenshot taken after execution of the user event e_{j_p} during the **record** phase, c_{j_p} may or may not become s_{j_p}. It works as a temporal variable.
O	a test oracle, a mechanism that determines whether a test passed or failed by comparing the current state s'_{j_p} with its corresponding expected state s_{j_p} after replaying e_{j_p}

3.2 Problem Statement

We now describe the problem formally with variables defined in Table 3. Given a set of test cases T associated with an application A, a set of DUTs D, and a set of expected GUI states S, we aim to design a test oracle O to verify test results. The test oracle O will capture the current GUI states S' of A on a DUT d_i while replaying a sequence of user events e_{j_p} in E

for the test case t_j, and compare every s'_{j_p} in S' with its corresponding expected state s_{j_p} in S to determine whether t_j passed or failed. It is worth noting that even though s_{j_i} and s_{j_f} are valid expected states, and s'_{j_i} and s'_{j_f} are valid current states, they are not captured after the input of any user event, rather they are captured before the very first event, in the case of $s_{j_i}$and s'_{j_i}, and after the very last event, in the case of s_{j_f} and s'_{j_f}. D is a set of heterogeneous devices which means that the DUTs may have different screen sizes, different screen configurations and even different versions of Android OS.

Figure 1 exhibits part of a SPAG-C test case. This particular example has three events and four expected states (amplified to the right). During replay, each time the test reaches a *checkpoint,* lines 1, 12, 16 and 20, the test oracle O will capture the *current states* and *verify* they match the *expected states* that were derived during record.

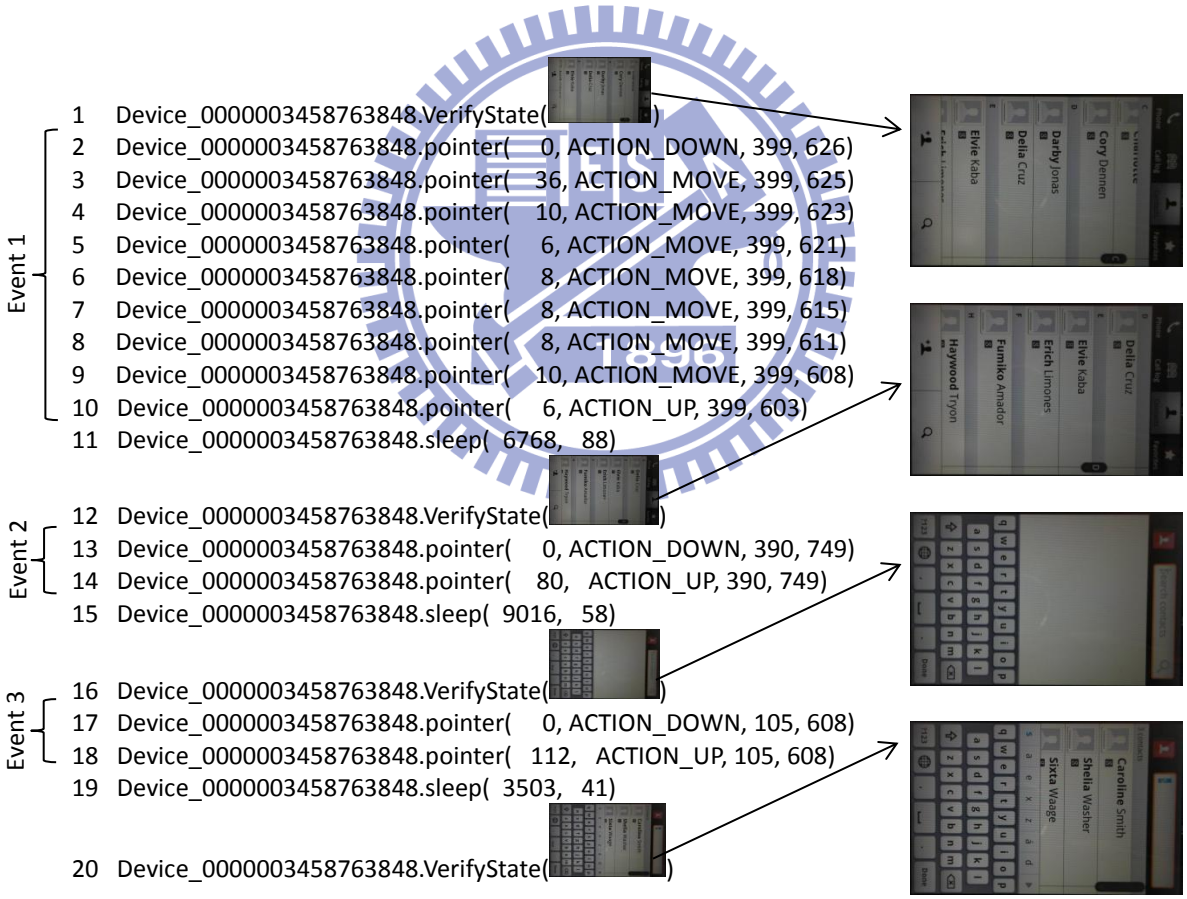


Figure 1: Example of SPAG-C test case

Chapter 4 SPAG-C Design

This chapter discusses our solution approach for improving the three issues of test automation mentioned before: accuracy, efficiency, and reusability. By observing that the test verification process is the same regardless of the DUT, we propose a new architecture for SPAG, called SPAG-C, which makes use of an external camera and Web services technology to decouple the test oracle from the record-replay component. Therefore, allowing us to address efficiency and reusability without compromising accuracy.

4.1 Architecture overview

As illustrated in Figure 2, we have two sets of components: hardware components and software components.

Hardware components

The DUT is the Android device that runs the application for which the test cases are written. It is worth noting that even though the test cases are written for a specific application, the DUT is what is being tested. The camera is used to capture the required GUI states during both record and replay phases. To avoid any interference with the process of capturing the required images, test cases are recorded by controlling the DUT remotely from a computer.

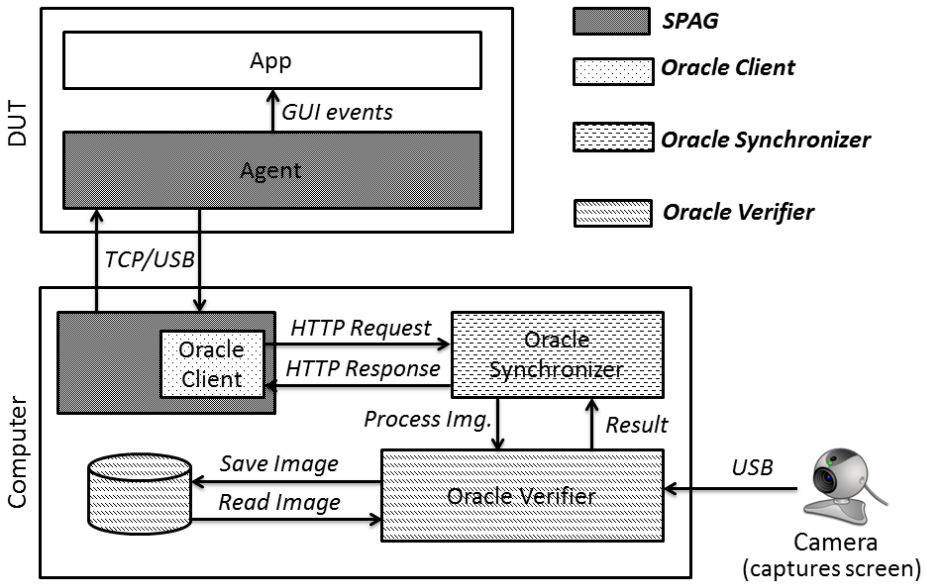


Figure 2: Architecture of SPAG-C

Software components

SPAG-C records and replays test cases remotely. We divide the test oracle into three major

components: *oracle client*, *oracle synchronizer* and *oracle verifier*. As shown in Figure 2, the oracle client is coupled with the record-replay component, in this case SPAG, and it is in charge of automatically adding verification commands to SPAG's test cases during the record phase and sending requests to the oracle synchronizer to verify a GUI state during the replay phase. The oracle synchronizer uses Web service technology to expose the oracle verifier to the oracle client. Oracle synchronizer also handles requests from the oracle client, passes them to the oracle verifier and sends the response back to the oracle client. The oracle verifier validates the testing results by capturing images from a camera, as shown in Figure 2, and comparing the GUI states.

Figure 2, also exhibits the original architecture of SPAG which consists of two modules: one runs on the DUT and the other on the host computer. The agent, which is installed on the DUT, is in charge of capturing the required information to perform the verification process, while Sikuli IDE (integrated with Android Screencast) runs on the host computer, and is in charge of recording and replaying user events.

SPAG-C *semi* and SPAG-C *auto*

Recall from Table 2 that each of these two tools provides both semi-automatic and automatic verification methods. Both SPAG's and SPAG-C's semi-automatic verification requires testers to provide screenshots and write assertions into the test case. The difference is that SPAG captures these screenshots from the frame buffer, while SPAG-C does it from the camera. The automatic method differs a lot in these two tools. While SPAG simply performs a string comparison to verify that the same activity transition that occurred after the input of a specific event during record also happens during replay, SPAG-C automatically performs image comparison based on the impact the input events have on the GUI during record. For example, if during the record phase the tester performs an event that causes the application to go from activity "com.android.contacts" to activity "com.android.contacts.twelvekeydialer" then SPAG will corroborate that the same transition happens after replaying that event. This, however, does not ensure it is displayed correctly. SPAG-C, on the other hand, automatically takes the required screenshots based on the difference threshold set by the tester and performs image comparison, which provides a better evaluation of the GUI.

4.2 Oracle Client

As Figure 2 shows, the oracle client is divided in two parts: handling the communication with the oracle synchronizer and integrating the test oracle with the record-replay component. In order to automatically *derive* the expected states during record phase, we propose an

automatic verification method. Traditionally, record-replay tools help testers to automatically write most of the code of the test cases. However, every time testers need to add a verification command, also known as checkpoint, to check the state of the GUI, they have to *manually* take the screenshots and write the verification command into the test case, which requires a considerable amount of time. Our method on the other hand, automates this process by analyzing the impact the user events have on the GUI, i.e., how much the GUI *changes* after the user events are executed, and automatically adding a verification command if the change goes beyond a given *threshold*.

Figure 3 shows how the oracle client captures images and adds verification commands to the test case by using our automatic verification method during the record phase. When the tester starts recording, the oracle client first asks the oracle synchronizer to capture the *initial* state s_{j_i} of the application and add a checkpoint to verify it on replay, then it listens to every user event. Recall from Chapter 3 that a GUI state is represented by a screenshot, and the verification process consists in an image comparison between the *expected* and *current* states. After event e_{j_p} is executed, the oracle client asks the oracle synchronizer to capture the *new* imagec_{j_p}, and measure the *difference* between c_{j_p} and $c_{j_{p-1}}$. If the difference goes beyond the threshold provided by the tester, the oracle synchronizer will return true, which means that a checkpoint must be added after e_{j_p} and c_{j_p} becomes s_{j_p}, the expected state. If the oracle synchronizer returns false, no checkpoint is added and $c_{j_{p-1}}$ is discarded.

Similarly to the initial state, when the tester finishes recording the test case, the oracle client asks the oracle synchronizer to capture the *final* state s_{j_f} and add a checkpoint to verify it on replay. This way both the initial state and the final state are *always* verified. A demo of the recording process using SPAG-C is accessible at [28].

Figure 4 describes the behavior of the oracle client during the replay phase. Similar to the record phase, before any event is replayed, the oracle client first asks the oracle synchronizer to capture the current initial state s'_{j_i} and compare it against its corresponding expected state s_{j_i}. If the states match then the replay process continues. Otherwise, the testing process is stopped and relevant error information is provided to the tester. During the replay process, the oracle client asks the oracle synchronizer to verify the current state s'_{j_p} every time it meets a checkpoint. If the verification passes, then the replay process continues. Otherwise, the replay process is stopped and relevant error information is provided to the tester. After the last event has been replayed, the current final state s'_{j_f} is also captured and compared against its

corresponding expected state s_{j_f}.

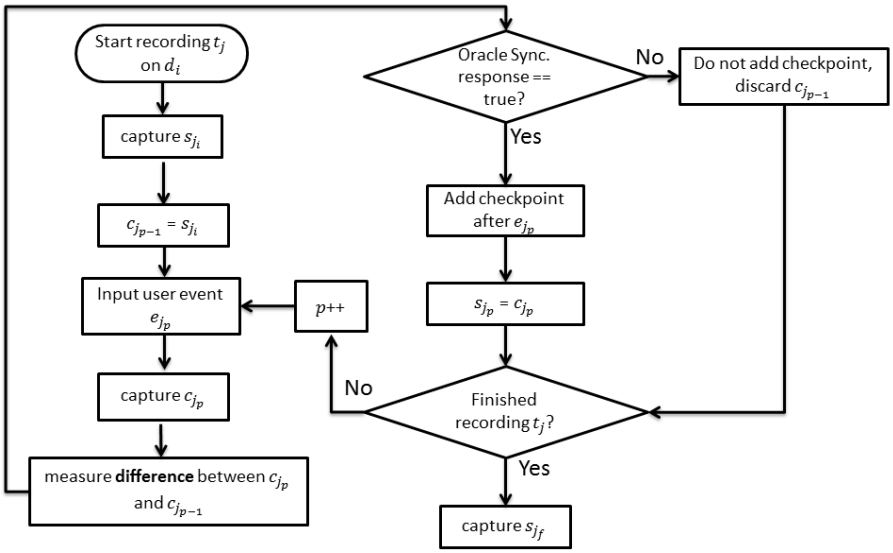


Figure 3: Oracle client (on record)

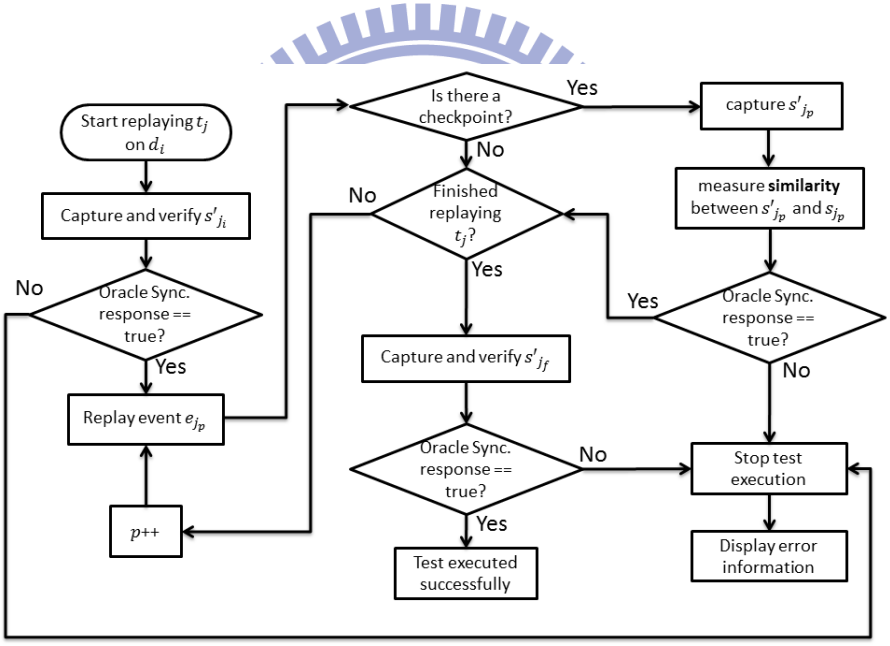
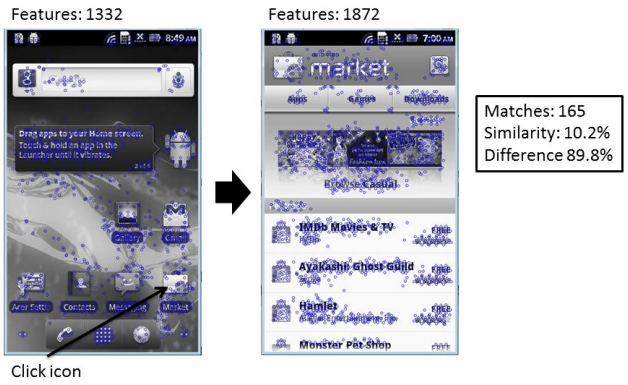


Figure 4: Oracle client (on replay)

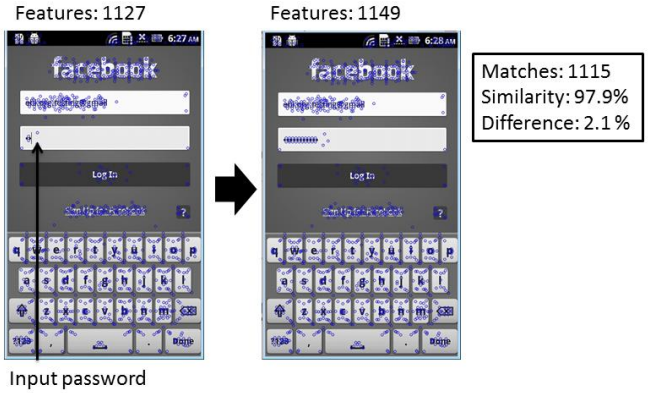
The reason we use c_{j_p} in Figure 3, record phase, is because we are trying to derive s_{j_p} by measuring the *difference* between the images captured before and after the execution of an event. The purpose is to give testers the ability to use the *difference* threshold to automatically determine how often the GUI should be verified. For example, in Figure 5(b), by setting the difference threshold between 10% and 20% testers can avoid adding a checkpoint after an event that causes no changes or only causes trivial changes in the GUI. In the case of Figure 5(a) since the click event causes a big change on the GUI, a checkpoint would still be added

to the test case. The little circles drawn on the images are the *features* detected by SURF.

In Figure 4, replay phase, we measure *similarity* between s_{j_p} and s'_{j_p} because they are expected to be the same, if they are not the same it means the GUI is not properly displayed.



a) Two different states



b) Two similar states

Figure 5: Example of the process described in Figure 3

4.3 Oracle Synchronizer

As its name suggests, the oracle synchronizer is the component in charge of synchronizing the oracle client with the oracle verifier. As shown in Figure 6, the oracle synchronizer is a Web service that listens to HTTP requests from the oracle client, de-serializes the message and asks the oracle verifier to perform the requested operation. After the oracle verifier is done performing the requested operation it sends the response back to the oracle synchronizer. The oracle synchronizer then serializes the response and sends it back to the oracle client. Since Web services are designed to support interoperable machine-to-machine interaction over a network [22], the oracle synchronizer allows our test oracle to be used by different tools, not just SPAG-C.

For every request, the oracle synchronizer may receive zero or more parameters from the

oracle client, but it always sends back a response, even if it is only to confirm that the operation has been executed successfully. The oracle synchronizer performs the same process during record and replay phases. The only difference is the parameters it receives from the oracle client. During the record stage, it receives c_{j_p}, $c_{j_{p-1}}$ and the threshold on difference. During the replay stage, it receives s_{j_p}, s'_{j_p} and the threshold on similarity.

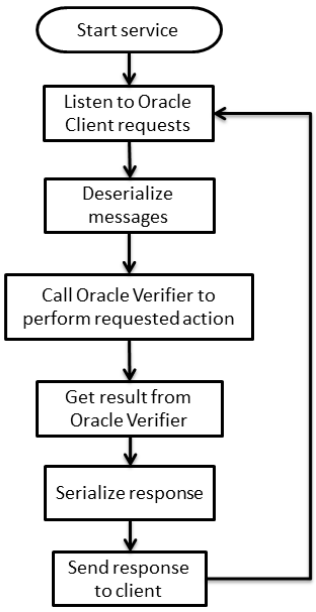


Figure 6: Oracle synchronizer

4.4 Oracle Verifier

The oracle verifier is where the verification process takes place. It is in charge of capturing the required screenshots using an external camera, performing image comparison and providing error information when a current state does not match its expected state.

Figure 7(a) shows the steps of the automatic verification process performed by the oracle verifier during the record phase. After the oracle client calls the oracle synchronizer asking it to measure the difference between c_{j_p} and $c_{j_{p-1}}$, the oracle synchronizer passes those images to the oracle verifier. The oracle verifier then performs a SURF [12] comparison between both images. Since SURF does not measure image difference, we need to measure their similarity first in order to measure the difference between both images. Image similarity is calculated as

$$similarity = \frac{100 * matched\ features}{Avg.\ features},$$

where the *matched features* are the result of performing a *nearest-neighbor* match between the features of both images, and some filtering to remove false matches. However, SURF only provides the matched features; we still need to develop a metric to measure similarity. We

opted to calculate what percentage of the *average features* represents the amount of *matched features*. The *average features* is simply the sum of the features detected in both images divided by two. Averaging the features of both images provides some flexibility in case two "identical" images have different amount of features detected, which may happen due to the *non-deterministic* characteristics of the external camera, or in case the DUT's position has changed. Instead of using the *average* we could have used the number of features detected in the image with the *least*, or *most*, features but our experiments showed that using average is more robust. Following the example in Figure 5(a), c_{j_p} has 1872 features and $c_{j_{p-1}}$ has 1132 features, the average features is 1502. After the match there are 165 features in common between both images, it means that they are 10.2% similar. Since the difference percentage is just the opposite of the similarity percentage, in order to calculate the difference between both images, we subtract the similarity percentage from 100 as

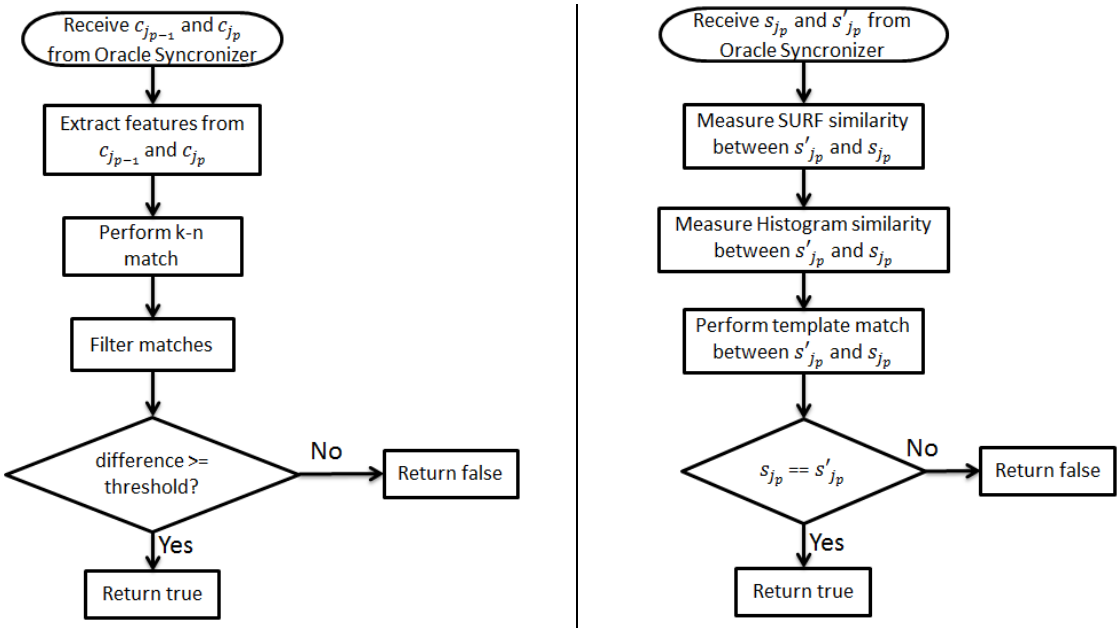
$$difference = 100 - \frac{100 * matched\ features}{Avg.\ features}.$$

Following the example above, the difference percentage would be 89.8%. Finally, if the difference percentage is *greater* than the threshold provided by the tester, the oracle verifier will return *true*, meaning that a checkpoint should be added.

During the replay phase, shown in Figure 7(b), the oracle verifier only compares images for similarity. Because in this case we are more concerned about accuracy, the oracle verifier performs three different kinds of image comparison: SURF, Histogram and Template match. For SURF, the process is the same as described before but we only care about similarity but not difference. As for Histogram and Template match, we simply use OpenCV's implementation of these techniques. In fact, we use OpenCV to extract, match and filter SURF features.

4.5 Synchronization

It is crucial for SPAG-C to capture images at the right time. Otherwise, too many errors would be introduced. Synchronization during record and replay phases is not the same. During the record stage, interaction from the tester cannot be predicted; but during the replay phase we do know when an event will be replayed.



a) record phase

b) replay phase

Figure 7: Oracle Verifier during record and replay phases

During the record stage, intuition suggests capturing c_{j_p} right after the tester inputs e_{j_p} but doing so would be wrong, because we would be assuming that the DUT takes virtually no time to process the requested operation, but in reality it does. In addition, the time required depends on some factors like CPU utilization and network connection. For example, a tester wants to install an application using Android Market, so he opens Android Market by clicking its icon. Since Android Market requires establishing an Internet connection, the device will display a white screen with a message informing that the application is loading. If we capture the image right after clicking the icon, all we would get is this white screen, which is not the desired state. Our solution to that problem is capturing c_{j_p} *after* the tester inputs $e_{j_{p+1}}$ remotely but *before* sending it to the DUT for it to take effect. The assumption here is that if the tester is able to input $e_{j_{p+1}}$ it means that e_{j_p} has *already* been processed and the GUI has already been updated accordingly.

Capturing GUI states during the replay phase is simpler, thanks to SPAG's *Smart Wait* function that measures, during the record phase, the elapsed time between e_{j_p} and $e_{j_{p+1}}$ along with the CPU utilization to predict, during the replay phase, how long to wait before replaying $e_{j_{p+1}}$. Therefore, during the replay stage, we simply capture the GUI states *after* Smart Wait's timer expires and *before* $e_{j_{p+1}}$ is replayed.

Chapter 5 SPAG-C Implementation

In the previous chapter we described how each component of SPAG-C works, now we proceed to explain how each component was implemented.

5.1 Oracle Client

As stated before the oracle client is coupled with the record-replay component. In our case that component is SPAG. Since SPAG is implemented in Java, the oracle client is also implemented in Java.

Most of the code of the oracle client is automatically created by the tool "wsdl2java" which is part of the Apache CXF framework (an open source services framework) [22]. "wsdl2java" simply takes the WSDL file exposed by a Web service (in this case the oracle synchronizer), and automatically generates Java code from which to call the service. With that code in place, we added event listeners to SPAG to automatically call the oracle synchronizer after any mouse event in order to perform the automatic verification process described in the previous chapter. During the replay phase, when the test case execution reaches the verification command, the client will make a call to the oracle synchronizer asking it to perform the requested action.

5.2 Oracle Verifier

The oracle verifier makes use of OpenCV (via EmguCV) and .NET framework functionality to compare, crop and rotate images.

EmguCV is as cross-platform .NET wrapper for the OpenCV image processing library [24] that allows calling OpenCV functions from any of the .NET compatible languages (in this case C#). OpenCV (Open Source Computer Vision) is a library of programming functions for real time computer vision [25]. In this work we only use OpenCV's functions to extract and compare SURF features, to calculate and compare image Histograms, to perform template matching, and to detect Canny Edges.

Detecting ROI (Region of Interest)

In order improve accuracy, we remove the undesired content of the images by detecting the ROI on the device's screen in two ways: manually and automatically. Since the tester can see in the computer the input of the camera, he can simply drag a rectangle around the desired ROI. After that, the oracle verifier will automatically crop any image captured. The automatic method, illustrated in Figure 8, makes use of an image processing technique called Canny

Edges to detect the borders of the screen. However, because Canny Edges will detect all edges in the image, it is important to make the device's screen as white as possible. After edges are detected, as shown in Figure 8 above "Canny edges", we find contours on the image but filter them in order to keep only quadrilateral figures. Then, the quadrilateral figures are filtered by the angle of their vertices; because we are looking for the screen of the device (i.e., a rectangle), we only keep those figures with angles between 80 and 100 degrees. Finally, we filter the rest of the figures by the size of their area, since the desired ROI should be the biggest rectangle detected. The result after the filter process is shown in Figure 8 above "Area detected".

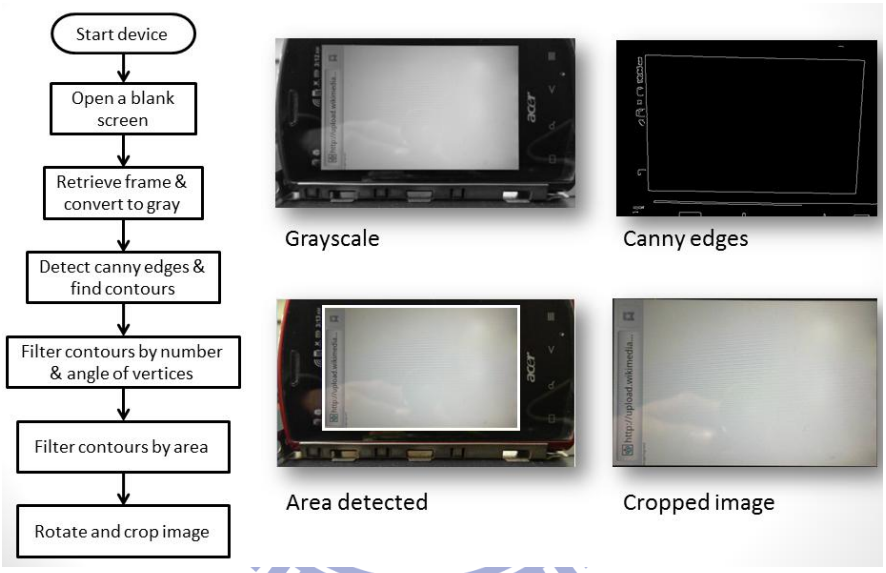


Figure 8: Process used to detect the device's screen (ROI)

SURF

In the previous chapter, we described most of the SURF comparison process; however, there are some implementation details that are worth mentioning. There are many ways to match SURF features. In this work we use OpenCV's *BruteForceMatcher* to perform *KnnMatch* (k-nearest neighbor match). *KnnMatch* returns the K nearest neighbors, $K = 2$ in our case, using a Euclidean Distance (L2 Distance). After the match is performed, we filter the matched features by using OpenCV's function *VoteForUniqueness*, which discards non-unique matches, with a uniqueness threshold of 0.9; and *VoteForSizeAndOrientation,* which discards those features whose size and orientation does not match the majority's size and orientation, using a scale increment of 1.5 and 20 rotation bins.

Histogram

In order to perform color histogram comparison, we need to first extract the color

histogram of each image. However, we first reduce the colors of the images to get a reliable similarity measure and improve computation efficiency [25] on each one of the three channels of the image: red, green, blue (RGB). After reducing the colors, we calculate the color histogram by calling OpenCV's *DenseHistogram.Calculate* method on each one of channels and compare the histograms by calling the *cvCompareHist* method.

Template match

As described in Chapter 2, template matching finds a given small image in a larger image. However, since our automatic verification automatically decides what images will be used as the expected states the tester cannot provide the templates. Therefore, we automatically split the expected state into smaller images, and match each of those small images against the current state. This means that several template matches are performed; if at least one of the matches has a value smaller than the provided similarity threshold then the overall match is considered a failure.

OpenCV implements different methods to match templates. In this work, we use the *MatchTemplate* function using *CV_TM_CCORR_NORMED*, which returns a value between -1 and 1, where the higher the value the better the match, and vice versa.

5.3 Oracle Synchronizer

We use Microsoft WCF and C# to implement the oracle synchronizer. Microsoft WCF is a framework for building service-oriented applications [23] that facilitates building interoperable Web services using different standards.

The oracle synchronizer exposes three methods: *AddCheckpoint, VerifyState* and *CaptureScreen.* Each of the methods receives a different set of parameters. The *AddCheckpoint* method is only called during the record phase, it receives the difference threshold as well as the previously captured screenshots c_{j_p} and $c_{j_{p-1}}$, and performs the automatic verification process. The *VerifyState* method is only called during the replay phase, it receives the expected state s_{j_p}, captured during the record phase, and the similarity threshold, and performs the verification process. Finally, the *CaptureScreen* method is called during both phases, it does not receive any parameters, it simply captures a screenshot and returns its path.

Additionally, the oracle synchronizer is in charge of logging all relevant information each time a call to the service is done. For example, during the verification process, it will log what images have been compared, what similarity thresholds we had, how long the image

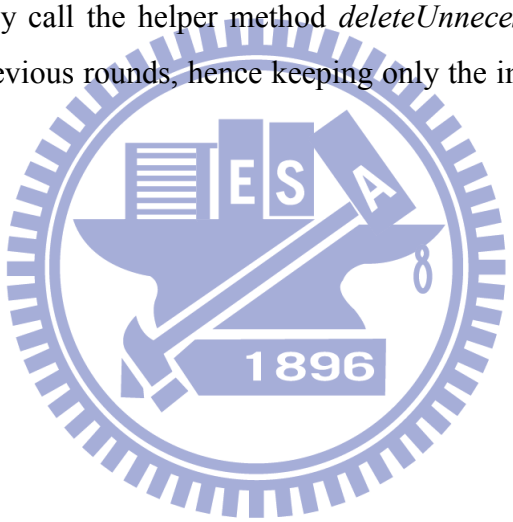
comparison process took and whether the verification process failed or passed.

5.4 Image maintenance

SPAG-C captures a lot of images but not all of them are required in order to replay a test case. In order to automate image maintenance, we name images randomly by using .NET's *Path.GetRandomFileName* method, and we prefix image names with the word "record", in case of s_{j_p}, and "replay", in case of s'_{j_p}.

Of all the images captured during the record phase, only the expected states are kept, and the rest are all discarded automatically. Expected states are required in order to replay a test case; therefore, these images are not deleted.

Each time a test case is replayed, a new set of current states will be captured. This means that a lot of duplicated images will be stored on the host computer. In order to alleviate this problem, testers can simply call the helper method *deleteUnnecessaryImages* to remove all images captured during previous rounds, hence keeping only the images that will be captured during the current round.



Chapter 6 Experiment Results

In this chapter we first introduce the environment used to evaluate our approach and show how SPAG-C improves over SPAG on the issues being addressed.

6.1 Testbed and test scenarios

As described in Chapter 4, SPAG-C uses three hardware components: the DUT, a host computer, and an external camera. We run our experiments on two different DUTs: Acer Liquid and LG-P920. Acer Liquid has a 3.5in TFT capacitive touchscreen with 256K colors and a resolution of 480 x 800 pixels. Acer Liquid runs Android 2.2 with Acer UI 3.0. On the other hand, LG-P920 has a 4.3in 3D LCD capacitive touchscreen with 16M colors and a resolution of 480 x 800 pixels; it runs Android 2.3 with LG 3D UI. The external camera used is a 1080p Microsoft Lifecam Studio with autofocus functionality. We use a normal desktop computer as the host computer with a 3.2GHz Intel Core i5 processor, 4GB of RAM and Windows 7 32-bits.

We perform experiments in both record and replay phases. Test cases are created for five different applications, as shown in Table 4: Contacts, Calculator, Google Maps, Android Market and Alarm Clock. During the record phase we are interested in measuring how much *time* we save using our automatic verification approach and how the differece threshold affects the number of *checkpoints* added to the testcase. During the replay phase we are interested in measuring how *accurate* the verification process is and how it is affected by external factors.

Table 4 shows the testing scenarios used to evaluate our approach. On each DUT, we run five popular applications. It‘s worth noting that some application’s GUI may be considerably different from one device to another as well as from one Android version to another, that is the reason why a *different* number of user events is required to perform the *same* operation on *different* devices.

During the record phase, we record every test case 10 times using each of the following difference thresholds: 20%, 40%, 60% and 80%. During the replay phase, we replay each test case 200 times: 100 times to measure *false negatives* (i.e., expecting tests to pass) and 100 times to measure *false positives* (i.e., expecting tests to fail). When measuring false positives we change the state of the application so that it will eventually display an *unexpected* GUI state; moreover, we introduce both trivial and significant changes to see how *small* should an error be to pass *unnoticed*. Errors are introduced in three different ways. First, in order to

simulate misplacement of GUI elements we edit the current states of the application (i.e., edit the images that represent the current states) so that they do not match the expected state. An alternative to this process (that would produce similar results) would be to use mutation testing to modify the layout of the application. However, since we are working with black box applications, mutation testing is not an option. Second, we add data to or delete data from the application. For example, if the test case is about finding the contact information of a person we delete that entry so that when replaying the test case such entry cannot be found, thus, making the expected and current states of the application different. Third, we add extra (erroneous) events to the test case after the recording process; this also causes the application to display different GUI states from those that are expected.

Table 4: Testing scenarios

Application	Test scenario	DUT	# of events	Android Version
Contacts	Contact lookup	Acer Liquid	9	2.2
		LG-P920	8	2.3
Calculator	Perform an operation	Acer Liquid	11	2.2
		LG-P920	11	2.3
Google Maps	Browse maps	Acer Liquid	13	2.2
		LG-P920	12	2.3
Android Market	Install Skype	Acer Liquid	9	2.2
		LG-P920	9	2.3
Alarm	Set alarm	Acer Liquid	11	2.2
		LG-P920	16	2.3

Figure 9 shows how the DUT and the camera were placed during experimentation. The DUT present on the image is Acer Liquid but the same positioning was used for LG-P920. The distance between the device and the camera is about 12 cm. During experimentation the auto-rotate screen functionality of both devices was disabled and the brightness fixed to the lowest possible. We placed the devices in landscape position because it allows bringing them closer to the camera (the camera has more horizontal resolution), thus capturing more detailed images.

6.2 Experiment Results

Traditionally, record-replay tools only automate the process of writing and executing test cases but not the process of verifying the state of an application. This means that testers need

to manually capture screenshots and add verification commands, which slows down the testing process. As mentioned in Chapter 4, the purpose of automating the verification process is to decrease the time required for recording a test case.

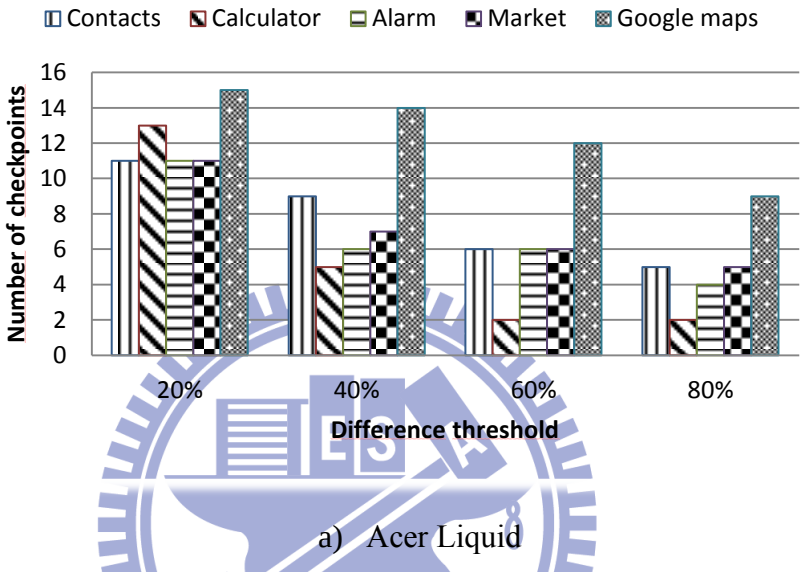
Figure 9: Positioning of the DUT and the camera

Efficiency

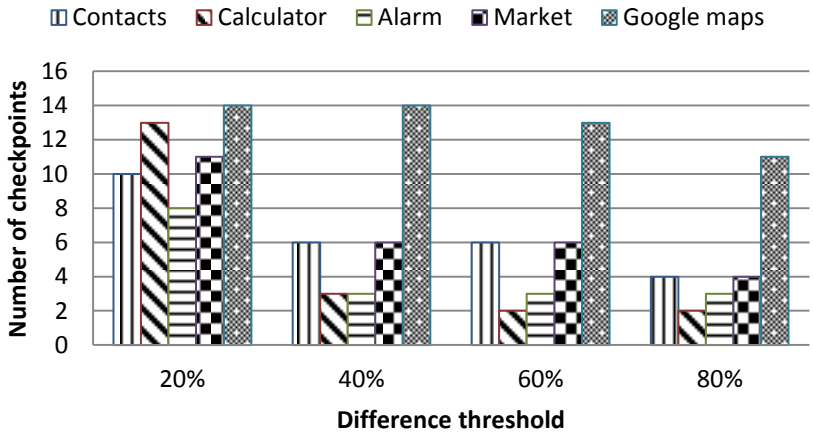
Figure 10 shows the number of checkpoints added to a test case during automatic verification during the record phase on both DUTs. We can see that when using a difference threshold of 20% there are two more checkpoints than the number of user events in each of the scenarios introduced in Table 4. The reason is that, as described before, automatic verification captures the *initial* and *final* states of the application when the record process starts and finishes, respectively. Therefore, we can say that using a difference threshold of 20% will add a checkpoint after *every* user event. Furthermore, using a difference threshold lower than 20% would rarely make a difference because, even if an event causes no change in the GUI state of the application, SURF comparison usually (though not always) considers c_{j_p} and $c_{j_{p-1}}$ at least 20% different (80% similar) for three reasons. First, c_{j_p} and $c_{j_{p-1}}$ may have a different amount of features detected due to the non-deterministic characteristics of the external camera images. Second, not all of the detected features will be matched. Third, the filtering process of the matched features performs well. This means that a SURF similarity match of 80% is actually close to a perfect match.

As expected, the number of checkpoints added to the test case decreases as the difference threshold increases. Figure 10 suggests that automatic verification has a different impact on different types of applications. We deliberately chose these applications because of their GUI characteristics. For example, most user events executed on the Calculator will only cause

small changes to the GUI, while in Google Maps user events usually have a *great* impact on the GUI. The Contacts application behaves somewhat in the middle while some events will introduce small changes others will introduce significant changes. Android Market and Alarm are what we call applications with "dynamic content", content that changes not only with user events but also with *time*, which introduces an interesting problem when using image comparison to determine the GUI state of an application because the current and expected states will *not* always be exactly the same.



a) Acer Liquid



b) LG-P920

Figure 10: Checkpoints added during automatic verification

Figure 10(b) exhibits no difference in the number of checkpoints added for Google maps by difference thresholds of 20% and 40%. The reason is that user events greatly impact the GUI, which means that c_{j_p} and $c_{j_{p-1}}$ generally are more than 40% different. This behavior,

however, is not seen in Figure 10(a). This is because, despite our efforts, it is hard to repeat exactly the same events every round; besides Google maps behaves differently on both devices. From Figure 10, we can also conclude that 80% should be the highest difference threshold used, either because only initial and final states are being captured, like in the case of Calculator, or because c_{j_p} and $c_{j_{p-1}}$ are so different that it is worth checking the new state.

Figure 11 exhibits the average time required to record a test case with: SPAG semi (taking the screenshots manually), SPAG auto (using SPAG's automatic verification), SPAG-C semi (taking the screenshots manually using the external camera) and SPAG-C auto (using SPAG-C's automatic verification). Clearly, recording a test case using both SPAG and SPAG-C takes considerable more time when capturing screenshots manually. SPAG takes slightly less time because with Sikuli's API the tester only needs to select the area of the screen he or she wants to capture, while SPAG-C requires the tester to use the camera to take a picture and save it. When using SPAG's and SPAG-C's automatic verification, the recording time is the same for both SPAG and SPAG-C. However, as mentioned before, SPAG's automatic verification doesn't corroborate that an application is being displayed properly, while SPAG-C does.

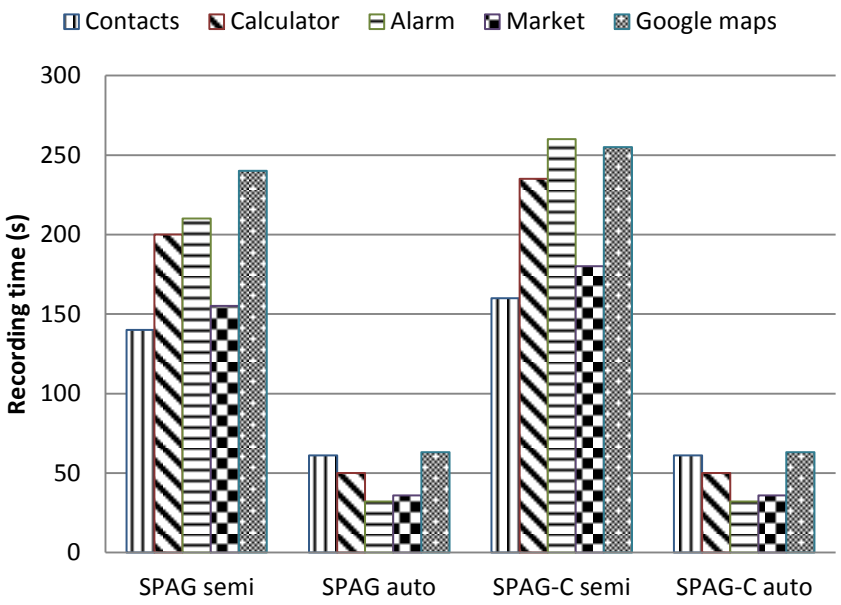


Figure 11: Time required for recording a test case

Accuracy

The verification process relies on the similarity threshold to determine if expected and final states are the same. In the previous two chapters we described what image comparison techniques we use in this work and how we use them. Although, in this work we use all three

of the techniques, it's worth mentioning that, if required, testers could use only one of those techniques without modifying the tool.

Figure 12 exhibits the running time of each image comparison technique, from which we can conclude that the time required to perform image comparison is relatively short. During record, automatic verification only takes a few milliseconds to execute; usually by the time the DUT has finished processing a user event, and Android Screencast has refreshed the screen on the remote host, the process would have completed already. During replay, the verification process only takes a few milliseconds. Therefore, the tester does not perceive any kind of delays from the test oracle.

The running time shown for Template Matching is based on a 6-column 2-row grid, which means that 12 template match operations are performed as described in Chapter 5.

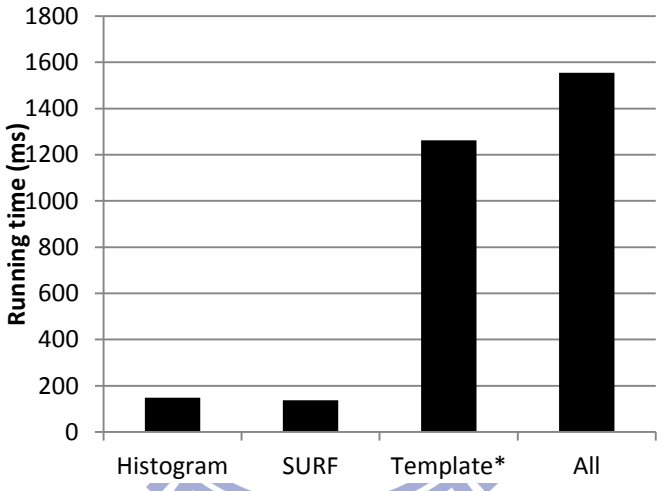


Figure 12: Running time of image comparison techniques

Table 5 suggests that different types of applications require different SURF and Template matching thresholds but not so for Histogram threshold. Reducing the colors of the image before performing Histogram comparison makes it more stable. Table 5 suggests that applications with "dynamic content" require *lower* similarity thresholds. This is because that s_{j_p} and s'_{j_p} may not be exactly the same. Therefore, more flexibility must be allowed to reduce false negatives. Nonetheless, setting lower thresholds makes it more difficult to detect small errors. This is all part of the tradeoff between accuracy and reusability of the test case. Setting higher thresholds increases accuracy but reduces reusability of the test case, and vice versa.

Figure 13 shows the sensitivity of each image comparison technique to the *similarity threshold* and its effect in the number of false positives and false negatives. The small circles

represent the optimal thresholds (the threshold that allows the lowest percentage of false positives and false negatives) as presented in table 5. The values to the left of the small circle represent the amount of false positives while the values to the right represent the amount of false negatives. Clearly, there is an inverse relationship between both values. As one increases the other decreases. The figure also suggests that some image comparison techniques are more sensible than other. Also, it can be observed that the three techniques working together compensate, to some extent, the flaws of the others. It is also clear that most errors are detected by SURF and Template match, and that Histogram could be removed without causing too much change in the accuracy of the tool. In order to get Histogram to contribute more significantly to the accuracy of the verification process, higher thresholds to those suggested in Table 5 could be used or the color reduction process before the Histogram comparison could be modified to allow for more sensitivity.

Table 5: Accuracy results

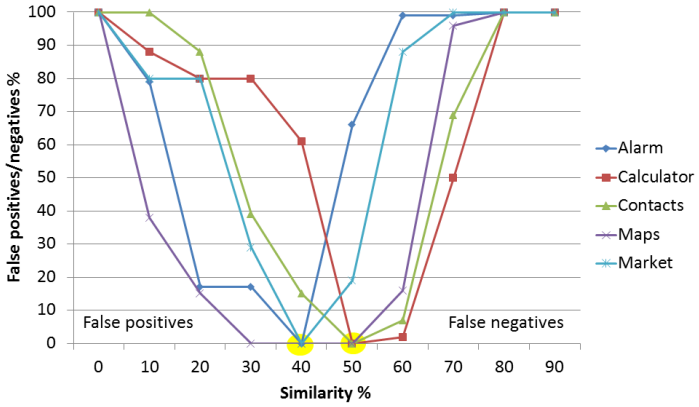
AUT	DUT	Histogram Threshold	SURF Threshold	Template Threshold	False Positives	False Negatives
Contacts	Acer Liquid	90%	50%	97%	2%	none
	LG-P920	90%	50%	97%	none	none
Calculator	Acer Liquid	90%	50%	97%	none	1%
	LG-P920	90%	50%	97%	2%	1%
Google Maps	Acer Liquid	90%	55%	97%	none	2%
	LG-P920	90%	40%	97%	none	2%
Alarm	Acer Liquid	90%	40%	80%	none	2%
	LG-P920	90%	40%	80%	none	none
Android Market	Acer Liquid	90%	40%	80%	none	2%
	LG-P920	90%	40%	80%	none	2%

Reusability

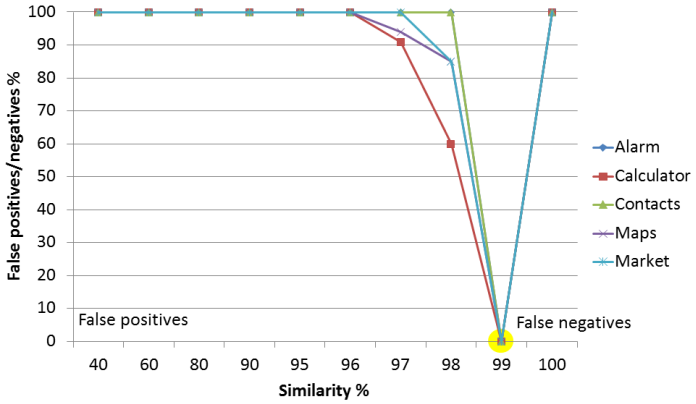
Android is an open platform, and as such, it is hard for a testing tool to provide support for all the devices available. Our test oracle, however, is *non-intrusive*. It does *not* depend on the

DUT to perform the verification process. This means that it can be used to test a great variety of heterogeneous smartphones.

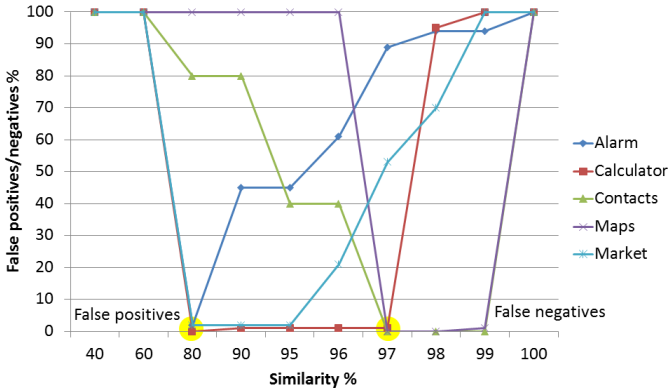
Earlier we mentioned that the reason why we decouple the test oracle from the record-replay tool is that we can reuse the test oracle. Furthermore, we proposed using Web services as an interface between both.



a) Parameter sensitivity for SURF



b) Parameter sensitivity for Histogram



c) Parameter sensitivity for Template

Figure 13: Sensitivity of image comparison to similarity threshold

Now, we proceed to show how our test oracle could be reused by other tools via Web services, and how much time can be saved by doing so. However, it is worth mentioning that *only* record-replay tools can take advantage of our automatic verification approach, since it is triggered by user events during the record phase. *Script-based* testing tools can still use the oracle to compare expected and current states but testers would have to capture the expected states *manually* before executing the test script.

Table 6 exhibits the list of tasks (marked with *) required to either reuse our test oracle or to implement it from scratch, and the estimated time to do so.

Table 6: Actions required for reusing or implementing our test oracle

Task	Estimated time	No reuse	Reuse
Setup development environment	~5 hours	*	
Oracle Client	~5 hours	*	*
Oracle Synchronizer	~1 day	*	
Oracle Verifier	~2 days	*	
Total time		~4 days	~5 hours

Reusing our test oracle is relatively simple thanks to Web service technologies. Since the Web service exposes a WSDL file, all that is required to create a client is to execute Apache CXF command *wsdl2java* (*svchost.exe* if creating a .NET service client). The *wsdl2java* reads the WSDL file and generates most of the client code. Once the client code is in place, developers need to add event listeners to the record-replay tool, as described in Chapter 4, to call the test oracle.

Assuming developers have experience with OpenCV and Web Service technologies, implementing our test oracle from scratch would take approximately 4 days. The bottle neck during the implementation process is the implementation of image processing techniques since it requires several adjustments of the different algorithms to get a robust yet flexible system so that it is tolerant to some changes in the external environment.

6.3 Discussion and Limitation

Since a camera is affected by its surroundings, our approach must be used in a relatively controlled environment or otherwise it would yield inaccurate results. The issues that are more likely to affect our system are: abrupt *changes* in lighting conditions in the room, *reflects* on the screen of the DUT, and *objects* getting in between the camera and the DUT. In order to avoid reflects we suggest placing the device facing towards a uniform dim black

background. Controlling lighting in the room and avoiding objects from getting in the way of the camera are trivial problems.

Using image comparison allows us to quickly verify an application's GUI in a platform independent way, support multiple devices without having to make changes to the tool and in most cases is accurate enough. However there are situations where using image comparison would not yield good results. For example, when testing applications with non-deterministic GUIs like video players and some types of games or applications who's GUI consists of a considerable amount of small text. Also, image comparison does not verify invisible GUI elements, which, though invisible, many times allow the positioning of other GUI elements.

Chapter 7 Conclusions and Future Work

This work, SPAG-C, is the continuation of a previous work called SPAG (Smart Phone Automated GUI testing tool) [6]. Both SPAG and SPAG-C use the record-replay technique to perform GUI testing on Android devices. Traditionally, record-replay tools facilitate the test case writing process but not the verification process. SPAG-C aims to reduce the time required to record test cases by automating the verification process and increase reusability of the test oracle without compromising accuracy.

In order to solve these issues, we developed SPAG-C. We use an external camera to capture screenshots, and decouple the record-replay component from the test oracle to allow the latter be reused by different testing tools via Web services. Furthermore, we proposed a different and more accurate automatic verification method to fully automate the test verification process; thus reducing image maintenance and the time required to record test cases. Finally, we use different robust computer vision techniques to verify that an application's GUI is being displayed as expected. This approach allows us to perform black box testing on real Android devices without the need to apply reverse engineering techniques.

Our experiments show that recording a test case using SPAG-C's automatic verification is as fast as SPAG's but more accurate since we do make sure the application is being properly displayed; on the other hand, achieving the same accuracy with SPAG would require testers to use the semi-automatic approach, in which case our method would be between 50% and 75% faster. Moreover, we explained how our method can be used to verify an application's GUI only when the changes introduced by an event are not trivial, simply by adjusting the difference threshold. We also demonstrated that our test oracle can be reused via Web services, and that doing so only requires a few hours instead of several days, which is what it would take to implement a new one each time an unsupported device needs to be tested. Finally, we showed that besides the fact of using an external camera our solution remains accurate enough, having less than 2% false positives/negatives.

In the future, we plan to provide support for multi-touch events and find an approach to test non-deterministic GUIs. We would also like to experiment with other image processing techniques like LBP (Local Binary Patterns) to see the changes in performance, flexibility and accuracy, and probably use some reverse engineering to complement the verification process.

References

[1] Microsoft MSDN, Guidelines for touch interaction, http://msdn.microsoft.com/en-us/library/cc872774.aspx, Mar. 2013.

[2] A. M. Memon, M. E. Pollack, and M. L. Soffa, "Automated test oracles for GUIs," SIGSOFT Softw. Eng. Notes, vol. 25, pp. 30-39, 2000.

[3] L. Baresi and M. Young. Test oracles. Technical Report CISTR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001. http://www.cs.uoregon.edu/ michal/pubs/oracles.html.

[4] Q. Xie and A. M. Memon, "Model-Based Testing of Community-Driven Open-Source GUI Applications," Software Maintenance, 22nd IEEE International Conference on (ICSM), vol., no., pp.145-154, 24-27, Sept. 2006.

[5] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," Software Testing, Verification and Validation (ICST), IEEE Fourth International Conference on, pp. 377-386, 2011.

[6] Y. D. Lin, Edward T. H. Chu, S. C. Yu, Y. C. Lai, S. Z. Yu, “On the Accuracy of Automated GUI Testing for Embedded Systems”, IEEE Software, in revision.

[7] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI Testing Tools Using Accessibility Technologies," Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops, 2009.

[8] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: automated remote UI testing on Android," Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, Ulm, Germany, 2012.

[9] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Atlanta, Georgia, USA, 2010.

[10] Android Screencast, an open-source remote control tool for Android devices, http://code.google.com/p/androidscreencast/, Mar. 2013.

[11] Greg Pass, Ramin Zabih, "Comparing Images Using Joint Histograms", Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, Atlanta, Georgia, USA, 2010.

[12] A. E. Herbert Bay, Tinne Tuytelaars, Luc Van Gool, "SURF: Speeded Up Robust Features," Computer Vision and Image Understanding (CVIU), 2008.

[13] Opencv, Template matching, http://docs.opencv.org/doc/tutorials/imgproc/histograms

/template_matching/template_matching.html, Mar. 2013.

[14] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," in Software Testing, Verification and Validation (ICST), IEEE Fourth International Conference on, pp. 377-386, 2011.

[15] Android MonkeyRunner, http://developer.android.com/tools/help/monkeyrunner_concepts.html, Mar. 2013.

[16] Robotium Framework homepage, http://code.google.com/p/robotium/, Mar. 2013.

[17] Android GUITAR, a model-based system for automated GUI testing, http://sourceforge.net/apps/mediawiki/guitar/index.php?title=GUITAR_Home_Page, Mar. 2013.

[18] Android Developer Guide, Instrumentation, http://developer.android.com/reference/android/app/ Instrumentation.html, Jun. 2013.

[19] Android Developer Guide, Application Fundamentals, http://developer.android.com/guide/components/fundamentals.html, Mar. 2013.

[20] Atif M. Memon, “GUI Testing: Pitfalls and Process,” in IEEE Computer, vol. 35, no. 8, pp. 87-88, 2002.

[21] W3C Working Group Note 11 February 2004, Web Services Glossary, http://www.w3.org/TR/ws-gloss/, Apr. 2013.

[22] Apache CXF: An Open-Source Services Framework, http://cxf.apache.org/, Apr. 2013.

[23] Windows Communication Foundation, http://msdn.microsoft.com/en-us/library/ms731082.aspx, Apr. 2013.

[24] EmguCV, http://www.emgu.com/wiki/index.php/Main_Page, Apr. 2013.

[25] Rober Laganiere, OpenCV 2 Computer Vision Application Programming Cookbook, Packt Publishing, May 2011.

[26] W3C Working Group, Web Services Architecture, http://www.w3.org/TR/ws-arch/#id2260892 Jun. 2013.

[27] Android, Hierarchy Viewer, http://developer.android.com/tools/help/hierarchy-viewer.html, Jun. 2013.

[28] SPAG-C live demo, http://www.youtube.com/watch?v=V841LpD4ULo&feature=youtu.be Jun. 2013.