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

Ying-Dar Lin, *Fellow, IEEE*, Jose F. Rojas, Edward T.-H. Chu, *Member, IEEE*, and Yuan-Cheng Lai, *Member, IEEE*

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. Traditional record-replay testing tools mainly focus on facilitating the test case writing process but not the replay and verification process. The accuracy of testing tools degrades significantly when the device under test (DUT) is under heavy load. In order to improve the accuracy, our previous work, SPAG, uses event batching and smart wait function to eliminate the uncertainty of the replay process and adopts GUI layout information to verify the testing results. SPAG maintains an accuracy of up to 99.5 percent and outperforms existing methods. In this work, we propose smart phone automated GUI testing tool with camera (SPAG-C), an extension of SPAG, to test an Android hardware device. Our goal is to further 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 writes 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 to allow 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. Moreover, SPAG-C is 50 to 75 percent faster than SPAG in achieving the same test accuracy. With reusability, SPAG-C reduces the testing time from days to hours for heterogeneous devices.

Index Terms—Reusable software, test execution, testing tools, user interfaces

1 Introduction

Automated graphical user interface (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 behaves 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 to 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.

- *Y.-D. Lin and J.F. Rojas are with the Department of Computer Science, National Chiao Tung University, 1001 University Road, Hsinchu, Taiwan. E-mail: ydlin@cs.nctu.edu.tw, josefcorojas@gmail.com.*
- *E.T.-H. Chu is with the Department of Computer Science and Information Engineering, National Yunlin University of Science and Technology, Yunlin, Taiwan. E-mail: edwardchu@yuntech.edu.tw.*
- *Y.-C. Lai is with the Department of Information Management, National Taiwan University of Science and Technology, No. 43, Sec. 4, Keelung Road, Taipei, Taiwan. E-mail: laiyc@cs.ntust.edu.tw.*

Manuscript received 19 Aug. 2013; revised 25 May 2014; accepted 11 June 2014. Date of publication 18 June 2014; date of current version 17 Oct. 2014. Recommended for acceptance by L. Baresi.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TSE.2014.2331982

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 testing 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. According to our previous work SPAG [6], the accuracy of testing tools drops significantly when the device under test (DUT) is heavily loaded, such as running many background processes, transferring data via the Internet and having many concurrently running applications. An application experiencing delay may fail to process an event correctly if the response to the previous event has not been completed. For example, an event may be dropped if the application receives the event ahead of time and is not ready to process it. The dropped event would cause the testing tool to report a false negative.

Improving the accuracy of matching GUI images, however, often conflicts with reusability of test oracles because testing the same application in devices with different screen sizes makes it impossible to use the same images as oracles in both devices even when testing the same functionality. Thus, in order to use the same test cases repeatedly, we must check for similarity in the output, but not exact matches. However, checking for similarity means that it is hard to detect small errors. Therefore, there is a tradeoff

between the accuracy of matching GUI images and the reusability of test oracles.

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 writing codes, but 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 smart phone automated GUI testing tool (SPAG) [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 improves them by adding some functionality that helps further automate the testing process on Android devices. First, SPAG monitors the CPU usage of target application at runtime. Next, SPAG dynamically changes the timing of the next operation so that all event sequences and verification can be performed on time. Compared to existing methods, SPAG maintains an accuracy of up to 99.5 percent and outperforms existing methods.

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, 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 enables us to quickly verify an application's GUI without having the source code (black box testing) and it is system independent, which allows 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 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.

The rest of this paper is structured as follows. Section 2 presents background and related work. Section 3 describes definitions and the problem statement. Section 4 presents SPAG-C design. Section 5 explains some details about SPAG-C implementation. Section 6 is our performance analysis, and Section 7 gives the conclusion.

2 Background and Related Work

In this section, we first provide background information on some image comparison methods used by SPAG-C. We then present a survey of related works on automated GUI testing.

2.1 Image Comparison

Histogram. 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 unaware 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). 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; perhaps the most important one is its repeatability, which means that it could be reliably computed under different conditions (e.g., changes in size, rotation, etc.). After an interest point of an image has been identified, the interest point descriptor 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. 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].

2.2 Automated GUI Testing Techniques

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 a great amount of possible test cases can be automatically generated. However, many of them might be irrelevant. Furthermore, in order to create a model, it is required to have detailed documentation of the SUT, and to keep and update that documentation and the model; 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, thereby verifying 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 the event and perform some operations like gathering the properties of the elements currently being displayed and verifying GUI state of an application. 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. Is the most popular approach for GUI test automation [20] because it allows creating test cases for an application without the need of writing codes. 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. While a tester interacts with the AUT, a tool automatically records all input events and writes them into a test case. Later, testers can modify the recorded test cases if required, and can 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, testers only have to record the test cases.

There is a large body of literature that addressed the problem of testing the correctness of mobile application software. Hu and Neamtiu [29] designed a testing process where random and deterministic events are executed in the AUT and the results are logged in files that are analyzed later in search of errors. Anand et al. [30] designed an algorithm and a system for generating input events to exercise smartphone applications and find software bugs. Based on concolic testing technique, their approach generates sequences of events automatically and systematically. Choi et al. [31] developed an automated technique, called SwiftHand, for generating sequences of test inputs for Android apps. The SwiftHand avoids restarts and aggressively merges states in order to quickly prune the state space. Their experiments show that for complex apps, their method could outperform both random and existing methods. Mirzaei et al. [32] adopted several symbolic execution tools, such as symbolic pathfinder, to generate test cases for Android apps. They first developed a model of Android libraries in Java pathfinder to enable execution of Android applications. They then leveraged program analysis techniques to correlate events with their handlers. Mahmood et al. [33] presented a framework for automated security testing of Android applications on the cloud. They developed a fully automated test case generator and a feedback loop to ensure code coverage. However, all the above works require the source files or application package files of the applications. In addition, all of them did not consider the problem of the synchronization of events. If the testing process introduces overhead to the device and the application takes longer to respond, it is not clear whether all the input events can be executed at the right time or not. Unlike these works [29], [30], [31], [32], [33], we address a different and unique problem. We assume that the application logic itself is correct. Given an application that's working correctly, we aim to find out if a given device (possibly a prototype) is capable of displaying such application as expected. We focus on the accuracy, efficiency, and reusability of automated test oracles for Android devices. In addition, we verify an application's GUI without having the source code (black box testing).

Machiry et al. [34] designed an input generation system, named Dynodriod, for Android Apps. For this aim, they used a novel "observe-select- execute" principle to efficiently generate a sequence of inputs to an app. On the contrary, our work addresses a completely different problem. We focus on the accuracy, efficiency, and reusability of automated test oracles for Android devices.

2.3 Android-Based Testing Tools

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, 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 [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 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 [8]. Is an Android testing platform that uses the Robotium framework to define test cases. Testdroid records user interactions 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 testers only need to record the tests.

GUITAR. Android graphical user interface testing framework (GUITAR) [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. The ripper 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. A main problem with GUITAR is that it may not be entirely practical on production-ready devices because it uses Hierarchy Viewer [27], a tool that can only connect to devices running a developer version of the Android system. In addition, GUITAR is platform-dependent even though the verification process is the same regardless of the device under test, which means the test oracle might not be reusable. On the other hand, SPAG-C can be used on a great variety of real devices. We use Web service technologies to expose the verification component to the record-replay component. Our test oracle is not only platform-independent but also independent from the record-replay component. Amalfitano et al. [35] discussed a similar problem as GUITAR did. However, no results were shown about the precision of the system when verifying the GUI. In addition, it may take a considerable amount of time to gather the information required to begin testing, and to perform the verification because the crawler needs to go throughout all possible event sequences and all windows. Further, they did not address the problem of event synchronization. If the testing process introduces overhead to the devise and it takes longer for the application to respond, it is not clear whether all the input events can be executed at the right time or not. Finally, their method cannot be used to perform black-box testing, because they instrumented the source code of the application under test to detect runtime crashes.

2.4 SPAG

This work, SPAG-C, is an extension of a previous work called 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 enable using 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: (1) Batch event, which accurately reproduces the recorded event sequences; (2) Smart wait, which automatically establishes a delay between events to ensure that the DUT has enough time to process previous events; and (3) 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 to 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 such as limited support for devices, slow response time that affects 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.

3 Definitions and Problem Statement

3.1 Definitions

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. 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, key input, etc. An event sequence, therefore, is one or more user events executed in succession. A test case represents a scenario used by testers to evaluate if an application behaves as expected. In our case, a test case is a script file that contains instructions to simulate user events on an application and verification commands to verify the GUI states.

We define $T = \{t_j \mid j = 1 \dots M\}$ as a set of M test cases associated with an application A, where $\mathrm{t_j}$ is the jth test case. $\mathrm{E} = \{e_{j_p} \mid p = 1, 2, \dots, L_j\}$ represents a sequence of $\mathrm{L_j}$ user events, where $\mathrm{e_{j_p}}$ is the p-th user event of the jth test case.

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. In this work, a GUI state is represented by a screenshot of the whole screen of the device. Since an application has many GUI states, $S = \{s_{j_p}\}$ is a set of screenshots that represent the expected GUI states of an application, where s_{j_p} is a screenshot taken after execution of the user event e_{j_p} during the record phase. Similarly, $S' = \{s'_{j_p}\}$ is a set of screenshots that represent the current GUI states of an application, where s'_{j_p} is a screenshot taken after execution of the user event e_{j_p} during the replay phase. However, the expected final and initial states of the application are represented as s_{j_i}, s_{j_f} respectively, and the current final and initial states of the application are represented as s'_{j_i}, s'_{j_f}, respectively.

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, represented as O is the mechanism in charge of capturing both states and performing such assertions. Finally, we define c_{j_p} as a screenshot taken after execution of the user event e_{j_p} during the record phase, $\mathrm{c_{j_p}}$ may or may not become s_{j_p}. It works as a temporary variable.

3.2 Problem Statement

We now formally describe the problem using the variables previously defined. 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 the test results. In other words, given an application that's working correctly, we aim to find out if a given device (possibly a prototype) is capable of displaying such application as expected. 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 will 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. Fig. 1 exhibits part of a SPAG-C test case. This particular example has three events and four expected states (with legend 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 phase.

4 SPAG-C Design

4.1 Architecture Overview

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

4.1.1 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.

4.1.2 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 Fig. 2, the oracle client is coupled with the record-replay component, in this case SPAG, and it is in charge of automatically adding checkpoints 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 Fig. 2, and comparing the GUI states using the image comparison techniques previously discussed. Fig. 2 also demonstrates 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.

Before moving forward to explain how SPAG-C's oracle works it is important to explain the differences between the way SPAG and SPAG-C verify the GUI states.

Each of these two tools provides both semi-automatic and automatic verification methods. Both SPAG's and

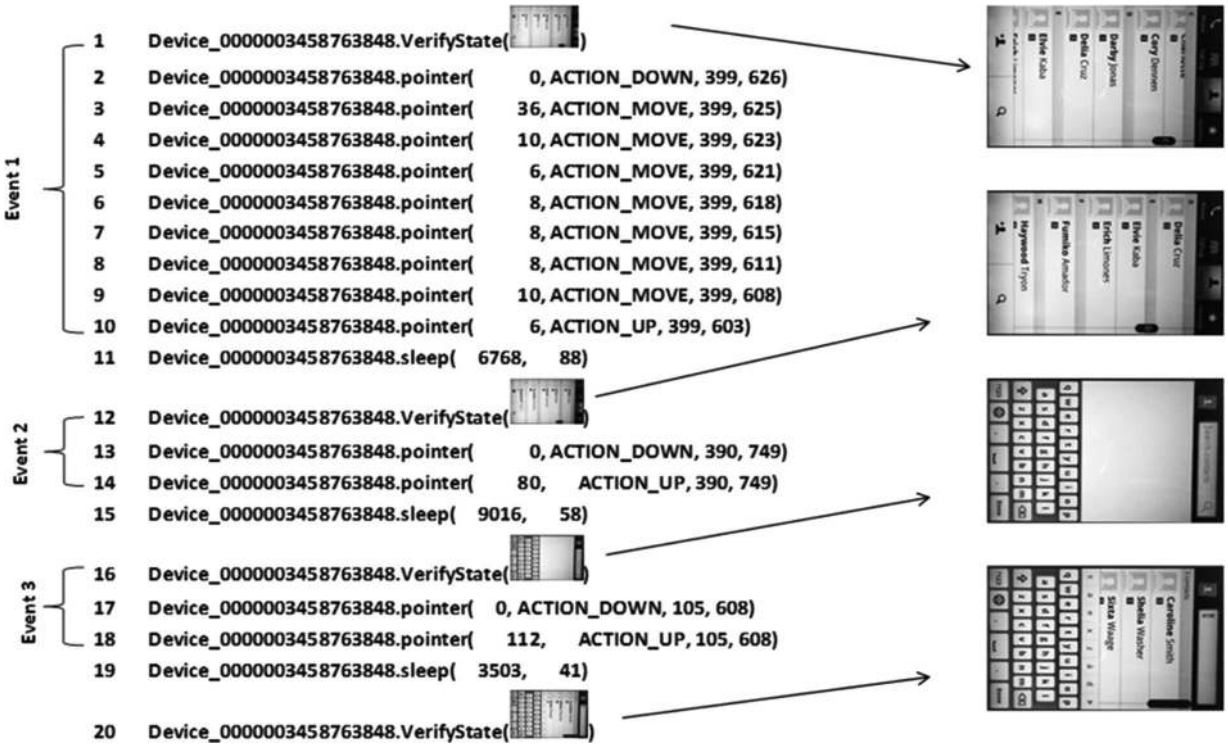

Fig. 1. Example of a SPAG-C test case consisting of three user events, four checkpoints, and four expected GUI states.

SPAG-C's semi-automatic verifications require testers to provide screenshots and write checkpoints 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 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 recording also happens during replaying, SPAG-C automatically performs image comparison based on the impact the user events have on the GUI during recording. 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.

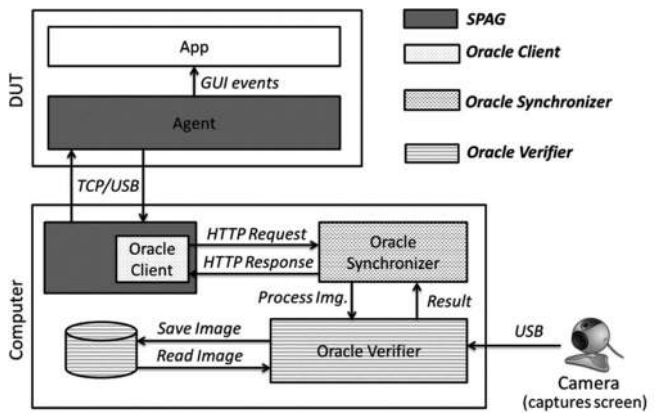

Fig. 2. SPAG-C architecture consists of hardware components: DUT, host computer, and camera; softare components: SPAG, oracle client, oracle synchronizer, and oracle verifier.

4.2 Oracle Client

As Fig. 2 shows, the oracle client performs two main functions: 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 by automatically adding a verification command if the change goes beyond a given threshold.

Fig. 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 the oracle client listens to every user event. Recall 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 image c_{j_p}, and measure the difference between c_{j_p} and $c_{j_{p-1}}$.

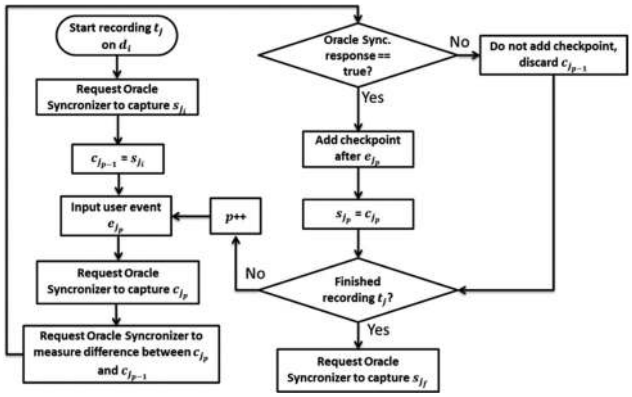

Fig. 3. The procedure used by the oracle client during record phase in order to automatically capture the expected GUI states and add a checkpoint to the test case.

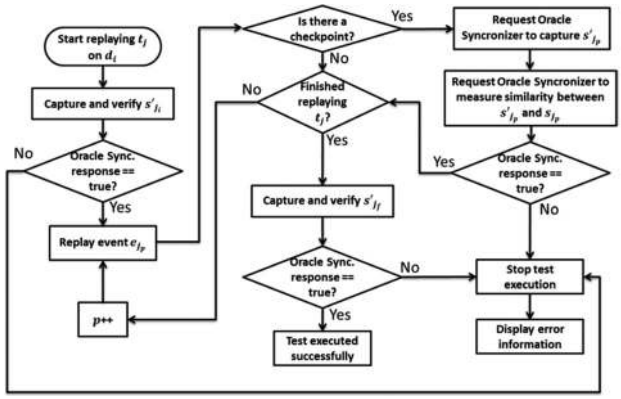

Fig. 4. The procedure used by the oracle client during replay phase in order to compare each *current* state against its respective *expected* state.

If the difference is more than 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.

The reason we use c_{j_p} in the record phase in Fig. 3 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.

Similar to recording 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 available on the Internet [28].

Fig. 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}.

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. The oracle synchronizer is a web service that listens to HTTP requests from the oracle client, de-serializes the messages, and asks the oracle verifier to perform the requested operation. After the oracle verifier finishes 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.

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.

Fig. 5a 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 first measure their similarity in order to measure the difference between both images. Image similarity is calculated as

$$similarity = \frac{100 \times \alpha}{\beta}, \tag{1}$$

where α presents matched features and β presents average features. 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 the percentage of the average features represented by the amount of matched features. The average features are simply the sum of the features detected in both images divided by two. Averaging the features of both images

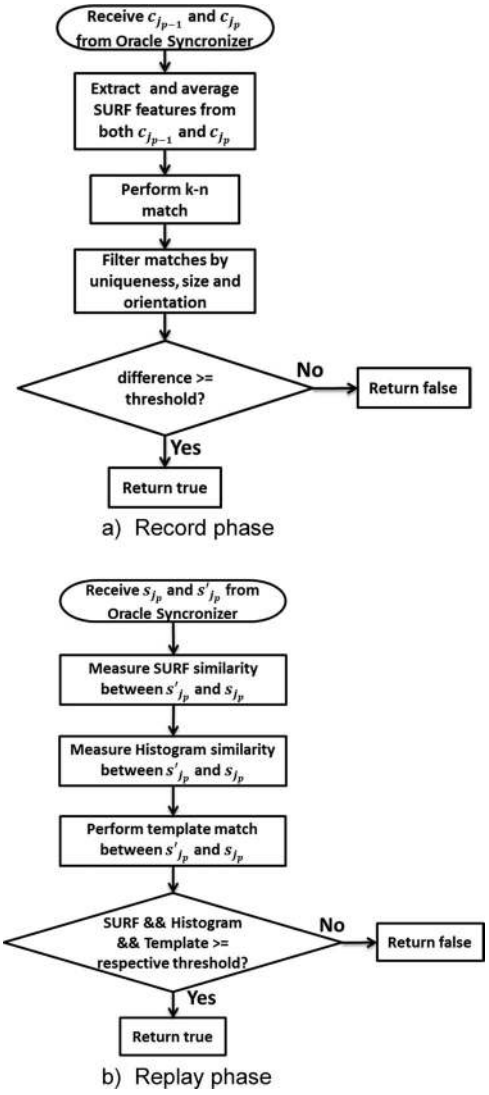

Fig. 5. The procedure used by the oracle verifier during (a) record and (b) replay phases in order to compare the current states against their respective expected states.

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 use the lessor or more of the two numbers of features detected, but our experiments show that using average is more robust. For example, if c_{j_p} has 1,872 features and $c_{j_{p-1}}$ has 1,132 features, the average number of features is 1,502. If after the match there are 165 features in common between both images, we say they are 10.2 percent similar. We let the image difference be the complement of the image similarity:

$$difference = 100 - similarity. \quad (2)$$

Following the example above, the difference percentage would be 89.8 percent. 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 by oracle client.

During the replay phase, shown in Fig. 5b, the oracle verifier only compares images for similarity. In order to ensure the accuracy of matching GUI images, the oracle verifier performs three different kinds of image comparison: SURF, Histogram matching, and Template matching. For SURF, the process is the same as described before but we only care about similarity and not difference. As for Histogram and Template matchings, we adopt open source computer vision's (OpenCVs) implementations. In fact, we also use OpenCV to extract, match, and filter SURF features. As Fig. 5b shows, the oracle verifier returns true only if each of SURF, Histogram matching, and Template matching returns true.

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 phase and synchronization during replay phase are not the same. During the record stage, interactions from the tester cannot be predicted; but during the replay phase, we do know when an event will be replayed.

During the record stage, intuition suggests capturing c_{j_p} right after the tester inputs e_{j_p}. However, doing so would be incorrect, because we would be assuming that the DUT takes virtually zero time to process the requested operation, whereas 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.

5 SPAG-C Implementation

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 [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 events in order to perform the automatic verification process described in the previous chapter. During the replay phase, when the test case execution reaches a checkpoint, 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 open source computer vision and .NET framework functionality to compare, crop, and rotate images.

In particular, we use EmguCV, a cross-platform .NET wrapper for the OpenCV image processing library [24] that allows us to call OpenCV functions from any of the .NET compatible languages (in this case C#). OpenCV 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.

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, based on 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.

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, and blue. After reducing the colors, we calculate the color histogram by calling OpenCV's DenseHistogram.Calculate method on each of the channels and compare the histograms by calling the cvCompareHist method.

As described in Section 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.

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, but simply captures a screenshot and returns its path.

Additionally, the oracle synchronizer is in charge of logging all relevant information each time there is a call to the service. For example, during the verification process, the oracle synchronizer will log the images that have been compared, the similarity thresholds we have, the length of time the image comparison process took, and the success or failture of the verification process.

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.

6 Experiment Results

6.1 Testbed and Test Scenarios

We compared SPAG-C with our previous work SPAG [6] in both testing efficiency and testing accuracy. Comparing against SPAG implies a comparison with Sikuli [2] since SPAG depends on Sikuli for GUI verification purposes. It also implies a comparison with Monkeyrunner [15] because a detailed comparison has already shown that SPAG ourperforms Monkeyrunner [6]. We did not compare SPAG-C with related work [8], [16], [17]. This is because some tools are not publicly available, and some present a very different testing approach. For example, Testdroid [8] is not available because it is a commercial cloud service and not a tool we can use. GUITAR´s [17] purpose was to automatically generate test cases for a given UI. In order to perform verification, they adopt object identification, which may not be useful for device testing. Robotium [16] also uses object identification to perform verification which may not be useful for device testing. That is, all these tools are designed to test applications not devices. Let´s consider the case where an application is correct but there is an issue with the device´s screen driver and that issue caused the application to be displayed incorrectly. Since the application is working properly performing object identification will not detect the problem.

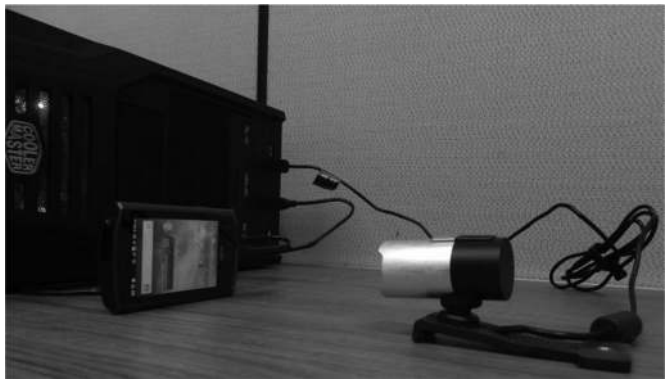

Fig. 6. Positioning of the devices during the testing process.

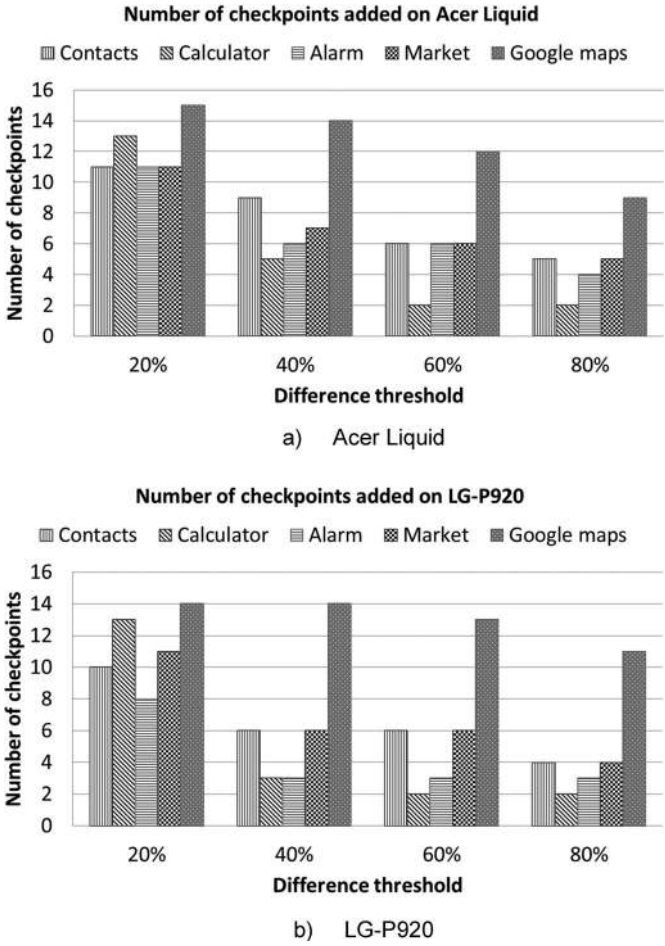

Fig. 7. Checkpoints added by automatic verification in (a) acer liquid and (b) LG-P920.

As mentioned before, SPAG-C uses three hardware components: the DUT, a host computer, and an external camera. In order to make a fairer comparison with SPAG, we run our experiments on two different DUTs: Acer Liquid and LG-P920. Acer Liquid has a 3.5 in TFT capacitive touchscreen with 256 K colors and a resolution of 480×800 pixels. Acer Liquid runs Android 2.2 with Acer UI 3.0. On the other hand, LG-P920 has a 4.3 in 3D LCD capacitive touchscreen with 16 M colors and a resolution of 480×800 pixels; it runs Android 2.3 with LG 3D UI. Note that any device supported by Android screencast can be tested using SPAG-C. The external camera used is a 1,080 p Microsoft Lifecam Studio with autofocus functionality. We use a normal desktop computer as the host computer with a 3.2 GHz Intel Core i5 processor, 4 GB of RAM, and 32-bit Windows 7.

We perform experiments in both record and replay phases. Test cases are created for five different applications: 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 difference 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.

During the record phase, we record every test case 10 times for each of the following difference thresholds: 20, 40, 60 and 80 percent. 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). A false negative is when a test is run and the SUT is working as expected, but the testing tool reports a failure. A false positive, on the other hand, is when there is a failure in the testing process but the testing tool does not detect it. 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 minor and significant changes to see how small an error could 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, thereby making the expected and the 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.

Fig. 6 shows how the DUT and the camera were placed during experimentation. The DUT presented in 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 set both the devices and the camera in landscape mode.

6.2 Testing Efficiency

Fig. 7 shows the number of checkpoints added to a test case during automatic verification during the record phase on both DUTs. As described before, automatic verification always 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

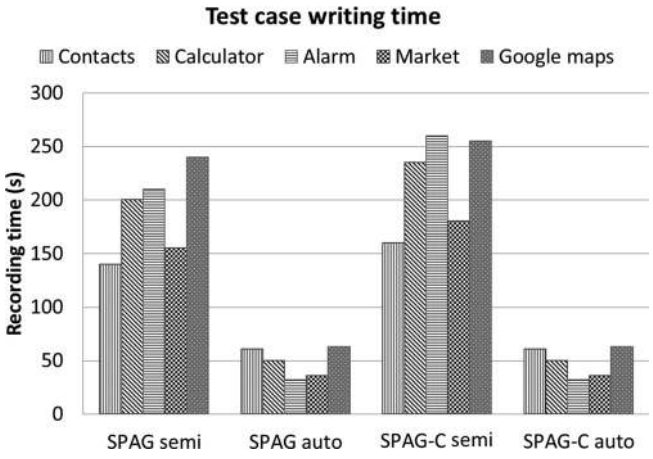

Fig. 8. Time required for recording a test case using semi-automatic and automatic approaches of both SPAG and SPAG-C.

20 percent will add a checkpoint after every user event, since the number of checkpoints added equals the number of events in the test case plus two (initial and final states). Furthermore, using a difference threshold lower than 20 percent would rarely make a difference because even if an event incurs 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 percent different (80 percent 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 percent is actually close to a perfect match.

As expected, the number of checkpoints added to the test case decreases as the difference threshold increases. Fig. 7 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 where some events will introduce small changes and 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. Fig. 7b exhibits no difference in the number of checkpoints added for Google maps for difference thresholds between 20 and 40 percent. 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 percent different. This behavior, however, is not seen in Fig. 7a. 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 Fig. 7, we can also conclude that 80 percent 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.

Fig. 8 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 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.

6.3 Testing Accuracy

In order to evaluate the accuracy of SPAG-C, we tested every difference threshold from 10 to 100 in increments of 5 to find out the different results. Table 1 displays the accuracy achieved with the specified thresholds, which are optimal as these are the ones that produce the least amount of false positives/negatives. Table 1 suggests that different

TABLE 1
Accuracy Results

App	DUT	h. t.	s. t.	t. t.	f. p.	f. n.
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%

Table 1 shows the percentage of false positives (F. P.) and false negatives (F. N.) when using the optimal thresholds for Histogram (H. T.), SURF (S. T.) and Template match (T. T.).

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 1 also 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. However, setting lower thresholds makes it more difficult to detect small errors.

According to our results, the similarity threshold should be set as high as possible to avoid false positives but low enough to avoid false negatives. The higher the thresholds the more false negatives we get, and the lower the thresholds the more false positives we get. An easy way to determine the thresholds is to run some test cases a few times without performing any kind of assertion, and check the log files for the similarity calculated between the current and expected states during different stages. Based on that value, an optimal similarity threshold can be chosen. This process should take a few minutes depending on the test case.

Fig. 9 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 yellow circles represent the optimal thresholds (the threshold that allows the lowest percentage of false positives and false negatives) as presented in Table 1. The values to the left of the yellow 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 sensitive than others. Also, it can be observed that the three techniques working together compensate, to some extent, the flaws of each other. 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 1 could be used, or the color reduction process before the Histogram comparison could be modified to allow for more sensitivity.

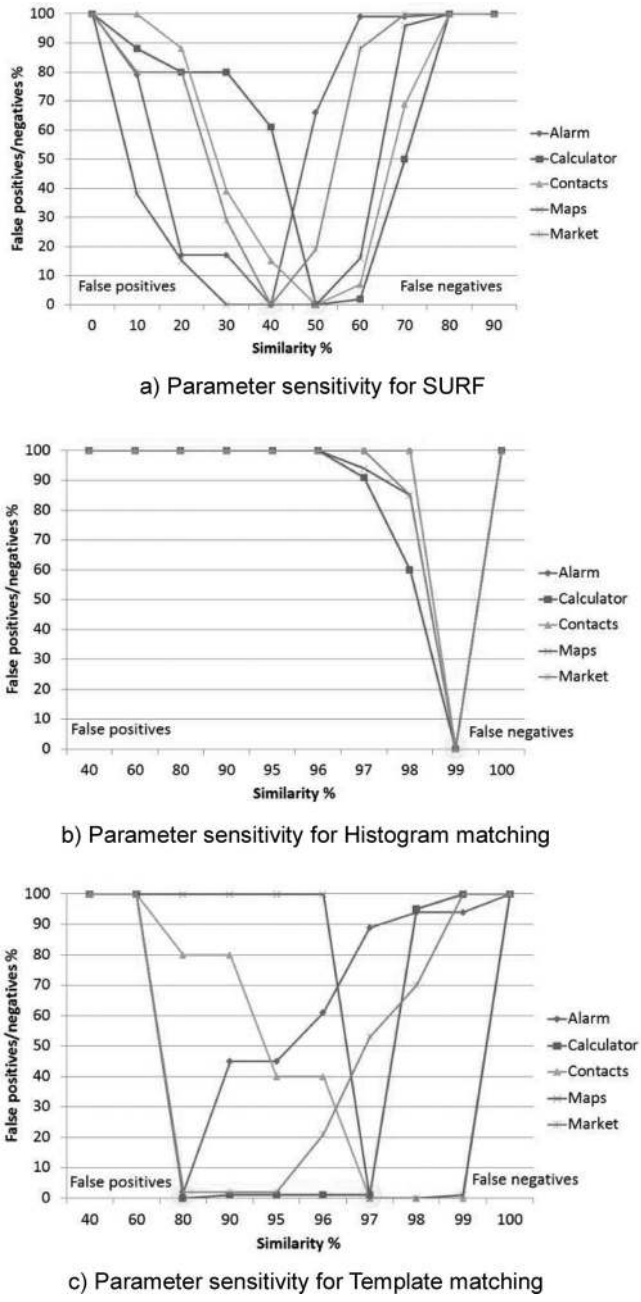

Fig. 9. Time required for recording a test case using semi-automatic and automatic approaches of both SPAG and SPAG-C.

6.4 Oracle Reusability

Android is an open platform; as such, it is hard for a testing tool to provide support for all the devices available. Our test oracle 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 in this paper we mention that the reason why we decouple the test oracle from the record-replay tool is that we can reuse the test oracle. Furthermore, we propose using Web services as an interface between both. We now show how our test oracle could be reused by other tools via Web services, and how much time can be saved by doing so. It is also 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.

Please note that when we say SPAG-C is reusable it means that the tool can be coupled with different testing frameworks, like Sikuli, SPAG and Robotium. Once SPAG-C is coupled with a testing framework it can be used to test devices supported by that framework. Therefore, SPAG-C is reused not by the applications being tested, but by the testing frameworks making use of it. For example, if one wants to use SPAG-C to test devices that are not supported by SPAG but are supported by Robotium, he or she could do that just by writing a new client for Robotium.

TABLE 2
Estimated Time to Reuse or Reimplement the Test Oracle

Task	Time	No reuse	Reuse
Setup Dev. Env.	~5 hours	*	
Oracle Client	~5 hours	*	*
Oracle Synchronizer	~1 day	*	
Oracle Verifier	~2 days	*	
Total time		~4 days	~5 hours

Table 2 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. The estimated time listed in Table 2 is based on not only our experience in implementing the entire system but also our experience in working with students who have a background in programming and computer vision.

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 Section 4, to call the test oracle.

For example, if one wants to use SPAG-C to test devices that are not supported by SPAG but are supported by Robotium, he or she could do that just by writing a new client for Robotium. According to our experience, it took us five hours to implement a client for Robotium. On the contrary, if SPAG-C was not reusable and one would like to use SPAG-C with Robotium, he or she would need to implement the entire system again, which would take much longer, around four days, assuming the programmer has knowledge of OpenCV and Web Service technologies, and he or she would have to do this every time he or she needs to couple the SPAG-C to another tool. 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.

6.5 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, reflections on the screen of the DUT, and objects getting in between the camera and the DUT. In order to avoid reflections we suggest placing the device facing towards a uniformly dim black background. Controlling lighting in the room and avoiding objects from getting in the way of the camera are trivial.

In this work, we define a GUI state of an application as the set of all GUI elements being displayed on the screen at a specific time. However, for some applications, only a small portion of the screen is changed when the applications respond to an input event. For these cases, the tester can select the region of interest (ROI) in the screen of the device. Then, SPAG-C will verify the ROI during testing.

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. Thus, SPAG-C is not able to test any kind of applications. 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, often allow the positioning of other GUI elements.

7 Conclusion

This work, SPAG-C, is the continuation of a previous work called SPAG [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. The accuracy of testing tools drops significantly when the DUT is under heavy loads. SPAG outperforms existing methods by using event batch and smart wait functions to eliminate the uncertainty of the replay process, and by adopting GUI layout information to verify the testing results. SPAG can maintain an accuracy of up to 99.5 percent. SPAG-C aims to further reduce the time required to record test cases by automating the verification process and increase reusability of the test oracle without compromising accuracy.

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 also make sure the application is being properly displayed. On the other hand, achieving the same accuracy with SPAG requires testers to use the semi-automatic approach, in which case, our method would be between 50 and 75 percent 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 nontrivial, simply by adjusting the difference threshold. We also demonstrate 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 show that despite using an external camera, our solution remains accurate, yielding less than 2 percent false positives/negatives.

Acknowledgments

This work was supported in part by National Science Council (NSC) and Institute of Information Industry (III) in Taiwan.

References

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

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

[3] L. Baresi and M. Young, "Test oracles," University of oregon, Dept. of Computer and Information Science, Tech. Rep. CISTR-01-02," Eugene, OR, U.S.A., [Online]. Available: http://www.cs.uoregon.edu/michal/pubs/oracles.html, Aug. 2001.

[4] Q. Xie and A. M. Memon, "Model-based testing of community-driven open-source GUI applications," in *Proc. IEEE 22nd Int. Conf. Softw. Maintenance*, Sep. 2006, pp. 145–154.
[5] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an android application," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 377–386.
[6] Y. D. Lin, T.-H. Chu Edward, S. C. Yu, and Y. C. Lai, "Improving the accuracy of automated GUI testing for embedded systems," *IEEE Softw.*, vol. 31, no. 1, pp. 39–45, Jan. 2014.
[7] M. Grechanik, Q. Xie, and C. Fu, "Creating GUI testing tools using accessibility technologies," in *Proc. IEEE Int. Conf. Softw. Testing, Verification, Validation Workshops*, 2009, pp. 243–250.
[8] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: Automated remote UI testing on android," in *Proc. 11th Int. Conf. Mobile Ubiquitous Multimedia*, Ulm, Germany, 2012, pp. 28:1–28:4.
[9] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, Atlanta, GA, USA, 2010, pp. 1535–1544.
[10] Android screencast, an open-source remote control tool for Android devices. (2013, Mar.) [Online]. Available: http://code.google.com/p/androidscreencast/
[11] G. Pass and R. Zabih, "Comparing images using joint histograms," *Multimedia Systems*, vol. 7, no. 3, pp. 234–240, May 1999.
[12] H. Bay, A. Ess, T. Tuytelaars, and L. Van Gool, "SURF: Speeded up robust features," *Comput. Vis. Image Understanding*, vol. 110, pp. 346–359, 2008.
[13] Opencv, template matching. (2013, Mar.) [Online]. Available: http://docs.opencv.org/doc/tutorials/imgproc/histograms/template_matching/template_matching.html
[14] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based GUI testing of an android application," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation*, 2011, pp. 377–386.
[15] Android MonkeyRunner. (2013, Mar.) [Online]. Available: http://developer.android.com/tools/help/monkeyrunner_concepts.html
[16] Robotium framework homepage. (2013, Mar.) [Online]. Available: http://code.google.com/p/robo tium/
[17] Android GUITAR, a model-based system for automated GUI testing. (2013, Mar.) [Online]. Available: http://sourceforge.net/apps/mediawiki/guitar/index.php?title=GUITAR_Home_Page
[18] Android developer guide, instrumentation. (2013, Jun.) [Online]. Available: http://developer.android.com/reference/android/app/Instrumentation.html
[19] Android developer guide, application fundamentals. (2013, Mar.) [Online]. Available: http://developer.android.com/guide/components/fundamentals.html
[20] A. M. Memon, "GUI testing: Pitfalls and process," *IEEE Comput.*, vol. 35, no. 8, pp. 87–88, Aug. 2002.
[21] W3C working group note 11 February 2004, web services glossary [Online]. Availabe: http://www.w3.org/TR/ws-gloss/, Apr. 2013.
[22] Apache CXF: An open-source services framework. (2013, Apr.) [Online]. Available: http://cxf.apache. org/
[23] Windows communication foundation. (2013, Apr.) [Online]. Available: http://msdn.microsoft.com/en-us/library/ms731082.aspx
[24] EmguCV. (2013, Apr.) [Online]. Available: http://www.emgu.com/wiki/index.php/Main_Page
[25] R. Laganiere, *OpenCV 2 Computer Vision Application Programming Cookbook*. Birmingham, U.K.: Packt Publishing, May 2011.
[26] W3C working group, web services architecture. (2013, Jun.) [Online]. Available:http://www.w3.org/TR/ws-arch/#id2260892
[27] Android, hierarchy viewer. (2013, Jun.) [Online]. Available: http://developer.android.com/tools/help/hierarchy-viewer.html
[28] SPAG-C live demo. (2013, Jun.) [Online]. Available: http://youtu.be/V841LpD4ULo
[29] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proc. 6th Int. Workshop Autom. Softw. Test*, 2011, pp. 77–83.
[30] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 59:1–59:11.
[31] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, New York, NY, USA, 2013, pp. 623—640.
[32] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood, "Testing android apps through symbolic execution," *ACM SIGSOFT Softw. Eng. Notes*, vol. 37, pp. 1–5, 2012.
[33] R. Mahmood, N. Esfahani, T. Kacem, N. Mirzaei, S. Malek, and A. Stavrou, "A whitebox approach for automated security testing of android applications on the cloud," in *Proc. 7th Int. Workshop Autom. Softw. Test*, 2012, pp. 22–28.
[34] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, Saint Petersburg, Russia, 2013, pp. 224–234.
[35] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A GUI crawling-based technique for android mobile application testing," in *Proc. IEEE 4th Int. Conf. Softw. Testing, Verification Validation Workshop*, 2011, pp. 252–261.

Ying-Dar Lin received the PhD degree in computer science from UCLA in 1993. He is currently a professor of computer science at National Chiao Tung University (NCTU) in Taiwan. Since 2002, he has been the founding director of Network Benchmarking Lab (NBL, www.nbl.org.tw), which reviews network products with real traffic. He also cofounded L7 Networks Inc. in 2002, later acquired by D-Link Corp. His research interests include network security, wireless communications, and embedded systems. He is a fellow of the IEEE and serves on the editorial boards of several IEEE journals and magazines. He published a textbook *Computer Networks: An Open Source Approach* (McGraw-Hill, 2011).

Jose F. Rojas received the MS degree in computer science from National Chiao Tung University, Hsinchu, Taiwan. He is currently a software engineer. His research interests include embedded systems and web technologies.

Edward T.-H. Chu received the PhD degree in computer science from the Department of Computer Science at National Tsing Hua University, Hsinchu, Taiwan, in 2010. He has more than four years work experience in the industry, where he worked on embedded software and owns a Chinese patent. He was a visiting scholar at Purdue University in 2009. He joined the Department of Electronic and Computer Science Information Engineering at National Yunlin University of Science and Technology, Taiwan, as an assistant professor in 2010. He received the best paper award in IEEE IS3C 2012, Taiwan. His research interests include embedded systems software and applications design. He is a member of the IEEE.

Yuan-Cheng Lai received the PhD degree from National Chiao Tung University, Hsinchu, Taiwan, in 1997. In August 1998, he joined the faculty of the Department of Computer Science and Information Science, National Cheng Kung University, Tainan, Taiwan. In August 2001, he joined the faculty of the Department of Information Management, National Taiwan University of Science and Technology, Taipei, Taiwan, where he has been a professor since February 2008. His research interests include performance analysis, protocol design, wireless networks, and web-based applications. He is a member of the IEEE.