

國立交通大學

資訊科學系

碩士論文

提升嵌入式系統圖形化介面測試之精確度

On the Accuracy of Automated GUI Testing for Embedded Systems

研究生：余尚哲

指導教授：林盈達 教授

中華民國 101 年 9 月

提升嵌入式系統圖形化介面自動化測試之精確度

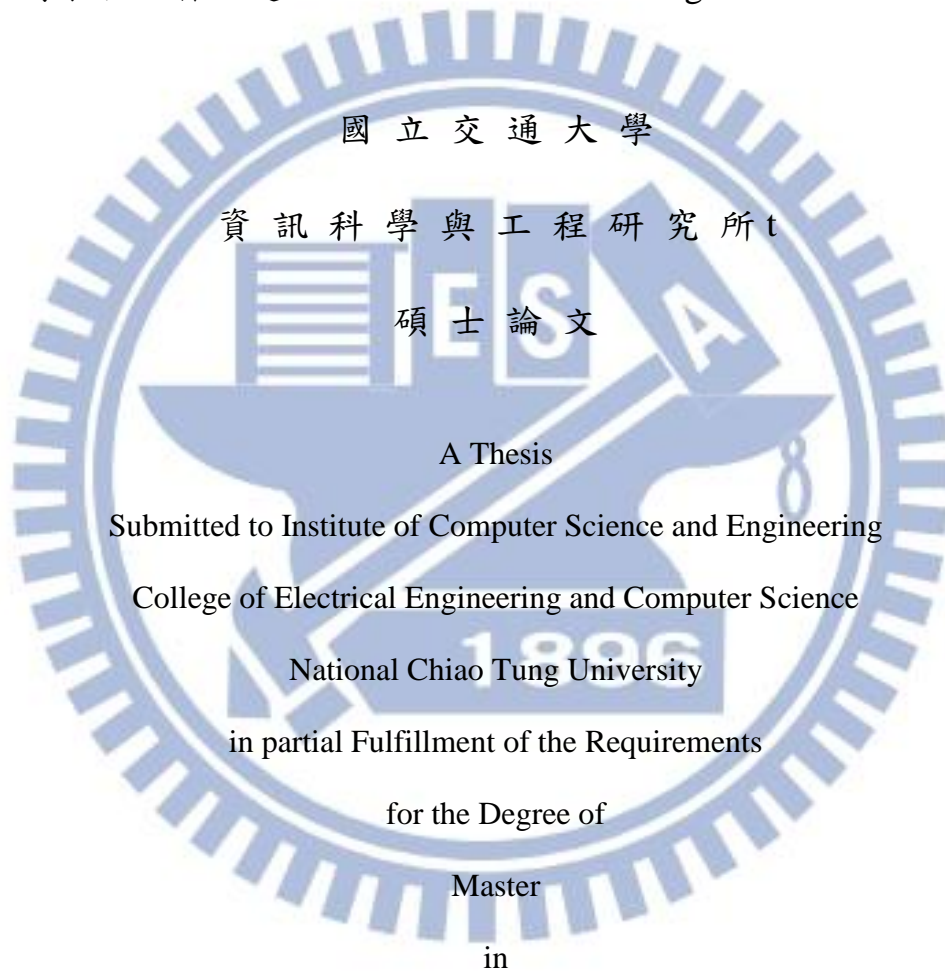
On the Accuracy of Automated GUI Testing for Embedded Systems

研究生：余尚哲

Student : Shang-Zhe Yu

指導教授：林盈達

Advisor : Ying-Dar Lin



Computer Science

September 2012

Hsinchu, Taiwan, Republic of China

中華民國 101 年 9 月

提升嵌入式系統圖形化介面測試之精確度

學生：余尚哲

指導教授：林盈達

國立交通大學資訊科學與工程研究所碩士班

摘 要

本基於省電與成本考量，嵌入式系統所配有的運算資源通常有限，要將現有的圖形介面測試技術套用在嵌入式系統軟體開發流程上是非常困難的。隨著智慧型手持裝置(如手機、平板)的普及，在嵌入式系統上進行自動化圖形化介面測試的需求也大幅提高，但現有的測試工具在使用時會有自動化不精確的問題。我們基於現有的測試工具 Sikuli 以遠端測試架構為基礎實作了圖形化介面自動化測試工具 SPAG，以提升測試事件重播的精確度。SPAG 將遠端手持式裝置的畫面顯示在本機端，讓使用者可以透過本機端畫面操作手持式裝置並錄製自動化測試項目。我們設計批次事件傳送與智慧等待兩種方法並實作在 SPAG 中，用以改善滑動等手勢的重播精確度，同時避免待測程式延遲造成的測試失敗。SPAG 會自動評估待測程式是否產生延遲，並等待額外的時間讓待測程式完成反應，以提高測試指令的完成機率。基於輔助功能技術，SPAG 能錄製流程中自動偵測待測程式狀態的改變，產生驗證指令，並在測試時自動驗證手持式裝置之狀態。我們選定一常用自動化測試工具 monkeyrunner 來比較測試精確度。在指定待測系統負荷工作量的情況下，SPAG 自動化測試的精確度仍然達到 90% 以上。若依照一般測試流程估計，SPAG 將可以比 monkeyrunner 少用 11%~71% 的測試回數。

關鍵字：嵌入式系統，圖形化介面，自動化測試，Android

On the Accuracy of Automated GUI Testing for Embedded Systems

Student: Shang-Zhe Yu

Advisor: Dr. Ying-Dar Lin

Department of Computer Science and Engineering
National Chiao Tung University

Abstract

The embedded systems are built with limited computation resources. Therefore, it is hard to apply conventional GUI testing tools on embedded systems. However, with the popular of smart handheld device, e.g. smart phone and pads, more software run on handheld devices and rising the need of software testing on handheld device. Current testing tools on embedded systems all have defect on reproducing GUI operations, so we design a Smart Phone Automated GUI testing tool (SPAG) based on Sikuli, in order to reduce testing defects. We design Batch Event and Smart Wait method to reproduce GUI operations accurately and ensure the device under test (DUT) can fully process all test operations. We also design the Hybrid Verifying method based on accessibility technology to generate most verify operations and verify them automatically. We use an experiment to compare our SPAG with monkeyrunner on test accuracy and time efficiency, in which SPAG archives 90% testing accuracy with all types of workload while monkeyrunner only has test accuracy of 27%~88%. We use the second experiment to analyze the contribution of our methods. The testing result shows our Smart Wait method brings more improvement on test accuracy.

Keywords: GUI, automated testing, embedded system, Android

誌 謝

本篇論文能從實作之中逐步發展成型直到今日所見成果，在這段求學期間最要感謝的是我的指道教授 林盈達老師。在每次的討論過程中孜孜不倦地指引我對於做研究的嚴謹態度與撰寫論文所需的知識與技巧。除此之外，林老師讓我了解人生中應重視的方向並以自身做了完美的示範，使我這兩年來受益匪淺。期許自己能夠牢記並掌握這些做事態度與技巧，來迎接將來人生各階段的挑戰。萬分感謝 朱宗賢老師花費極大心力在我的論文上，能夠不辭辛勞地指導我論文的撰寫方向與多次指導我修訂這篇論文，讓我最終能夠完成這篇論文。我也要感謝交大嵌入式測試中心(NCTU Embedded Benchmarking Lab)的甘東杰經理與 Shown 的指導與討論，讓我能夠從實做中逐步發覺現有測試工具有具有的議題，促成了本篇論文形成的動機；也感謝 EBL 能夠提供所需實驗器材，讓開發與實驗能夠順利進行。感謝我的母親與姊姊支持我在新竹的求學歷程，讓我能夠無須憂慮的完成學業。感謝我的女友雅婷，能夠在我遭遇到挫折的時候給予大量的支持與鼓勵。最後，感謝關心我、幫助過我的人。僅以此小小成就與大家分享！

余尚哲 謹誌

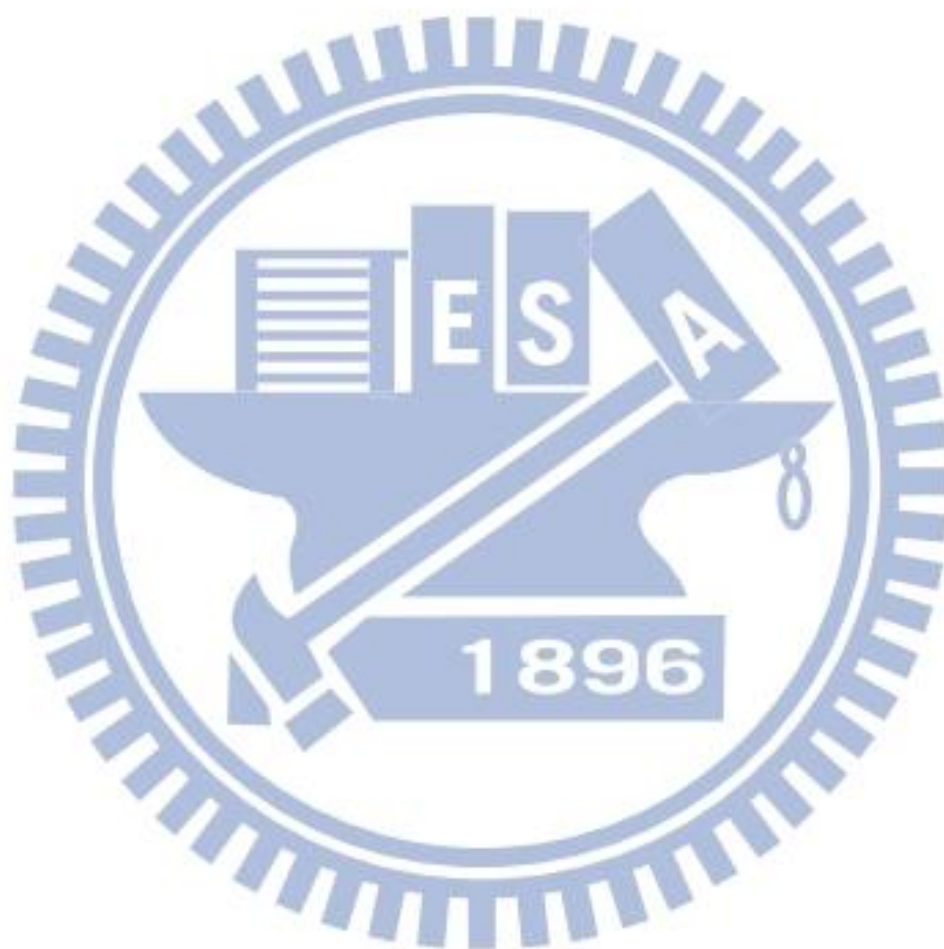
Contents

摘 要	I
Abstract	I
誌 謝	II
Contents.....	III
List of Figures	V
List of Tables.....	VI
Chapter 1 Introduction	1
Chapter 2 Background and Related Work	4
2.1 Challenges of automated GUI testing	4
2.2 Sikuli.....	4
2.3 Android Software Testing.....	5
2.4 Related Work	6
Chapter 3. Definitions and Problem Statement	8
3.1 Definitions	8
3.2 Problem Statement.....	14
Chapter 4. SPAG Design.....	15
4.1 Batch Event.....	16
4.2 Smart Wait.....	18
4.3 Hybrid Verifying	21
Chapter 5. Implementation.....	24
5.1 Device under Test	24
5.2 Recording and Replaying Events Sequence	25
<i>Recording input events from live demo.</i>	25
<i>Reproducing Event Sequence with Batch Event</i>	27
5.3 Smart Wait.....	27
5.4 Hybrid Verifying	28
Chapter 6. Experiment Result.....	30
6.1 Testbed	30
6.2 Comparison of test accuracy and time efficiency	31
6.3 SPAG Solution analysis.....	32
<i>Batch Event and Smart Wait</i>	33

Hybrid Verifying 34

Chapter 7. Conclusions and Future Work 36

References 38

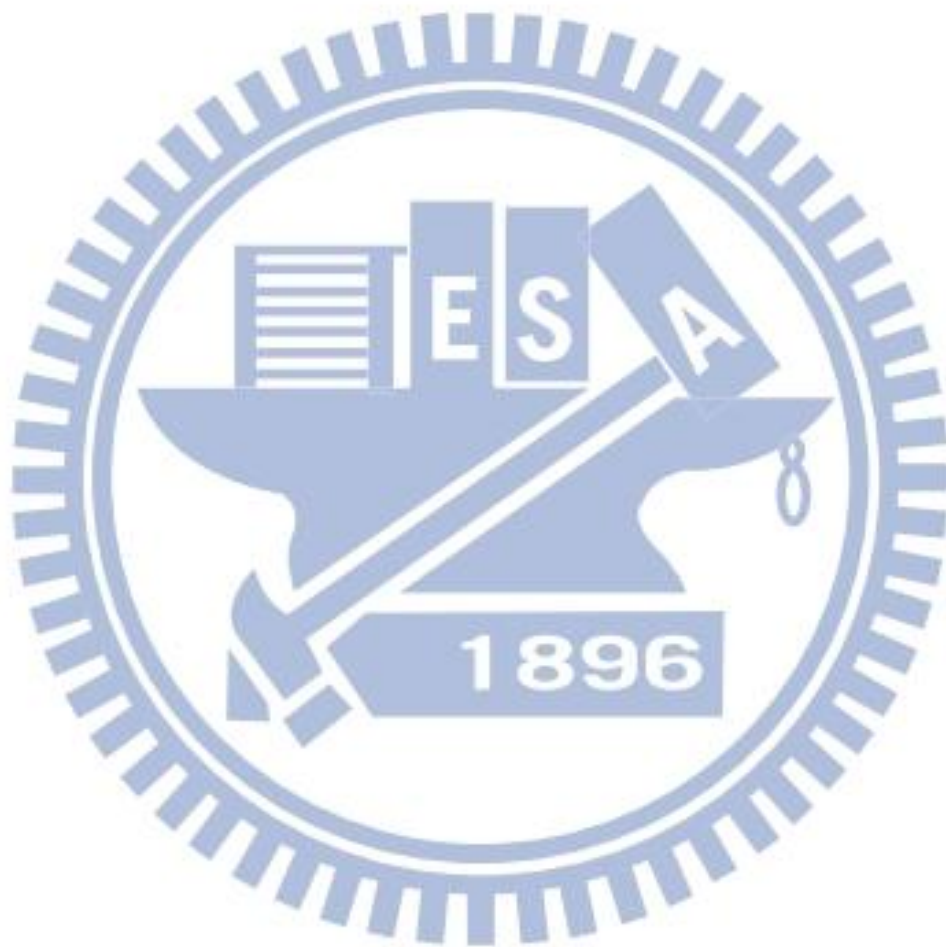


List of Figures

Figure 1. System architecture of record/replay method with DUT.....	8
Figure 2. Operations in a GUI test case C_1	9
Figure 3. GUI events in GUI operation O_1^G	10
Figure 4. Two approaches for sending event sequence.....	17
Figure 5. Flow of Batch Event method in replay stage.....	18
Figure 6. pseudo code of Smart Wait.....	20
Figure 7. Flow of Hybrid Verifying method in record stage.....	21
Figure 8. Flow of Hybrid Verifying method in replay stage.....	23
Figure 9. Screenshot of SPAG with screen of DUT.....	24
Figure 10. SPAG architecture in record stage.....	25
Figure 11. Example of SPAG's test script.....	26
Figure 12. SPAG architecture in replay stage.....	27
Figure 13. Testing with SPAG and monkey runner.....	31
Figure 14. the time efficiency of GUI testing.....	32
Figure 15. Testing with Batch Event and Smart Wait.....	33
Figure 16. Test case writing time with/without hybrid verifying.....	34

List of Tables

Table I. List of notations and definitions used in problem statement	12
Table II. List of commands used in SPAG design.....	16



Chapter 1 Introduction

Automated GUI (Graphical User Interface) testing tools are software programs used to test application user interface and to verify the functionalities. In the process of testing embedded software, engineers first design a test case consist of operations, which include several GUI operations and a set of conditions devised to determine whether an application works correctly or not. After engineers convert the test cases to a script file, the script performs predefined operations on a device under test (DUT), such as a smart phone or tablet PC. To verify the result, the DUT captures the screen and sends it to host PC, where an automated GUI testing tool performs verify operation. Take a popular open source automated GUI tool Sikuli, an Android device controlling tool AndroidScreenCast, and an Android smart phone for examples. Software engineers first write a Sikuli script to describe the timing and order of GUI operations, such as scroll screen and key press actions. At runtime, each action of the Sikuli script is performed on DUT screenshot window provided by AndroidScreenCast. These actions are interpreted into multiple motion events and key press events and transmitted to an Android smart phone, which is the DUT. After performing all received events, the AndroidScreenCast captures the screen of DUT and sends it back to the host PC, where the Sikuli verifies the correctness.

However, due to the uncertainty of runtime execution environment, such as timing delay variation in communication, interpreted events may not be reproduced at the DUT on time. As a result, intervals between events may be different from those expected. The non-deterministic event sequences may lead to an incorrect GUI operation. For example, Android Fling action happens when user scrolls on touch panel and then quickly lifts his finger. A sequence of motion events is used to represent the action. When replaying these event sequences, each motion event should be triggered on time in order to reproduce the Fling action with the same scrolling speed. Otherwise, the scrolling speed of the reproduced

filing action will be different from what is expected and therefore result in an incorrect result. In order to address the issue of non-deterministic events, a commonly used method is to use trackball instead of Fling action. However, trackball is not always equipped with a smart phone.

Uncertain runtime execution environment may cause another problem because it may interfere or delay the execution of application, especially under the circumstance that the DUT is in a heavy load condition. A delayed application may fail to process an event correctly if the response to previous event is not yet completed. For example, an event may be dropped if AUT receives the event ahead of time and is not ready to process it yet. To solve the problem, an intuitive method is to delay the executing of the operations. However, it requires experienced engineers to set the delay for each operation properly, so that the application can receive the reproduced events.

The other problem is how to verify test results efficiently. Traditional pixel-based takes relative long time on image transferring and processing. The situation becomes even worse for smart phones due to limited computation and communication capacities. This creates a strong need to develop a new method for automatically verifying the response of GUI operations.

In summary, based on our observations, automated GUI testing for smart phones faces three major challenges: non-deterministic events, execution interference and non-deterministic layout. In order to overcome the challenges, we design a Smart Phone Automated GUI testing tool (SPAG), which is based on Sikuli. To avoid non-deterministic events, we batch the event sequence and reproduce them on the DUT. In addition, SPAG can monitor the CPU usage of target application at runtime and dynamically change the timing of next operation so that all event sequences and verifications can be performed on time, even though the DUT is under a heavy load condition. Finally, SPAG adopts Android accessibility service to get necessary information for verifying DUT status. While cooperating with

traditional image matching method, SPAG can automatically generate most verify operations and check the testing results faster.

We conducted several experiments on a popular Acer Liquid smartphone in order to investigate the applicability and performance of SPAG. We compared our method with monkeyrunner[1]. In our experiments, we first investigated the effect of resource utilization of system during test. In addition, we studied the improvement of test accuracy achieved by our batch event and smart wait method. Finally, we explored how our hybrid verifying method reduces the cost of generating test cases.

The rest of the work is organized as follows. Chapter 2 mentions a GUI automated tool named Sikuli, and gives a survey of related works. Chapter 3 gives the definition of variables we used in this work, and describes our statement our problem. Chapter 4 details our solutions of batch event, smart wait and hybrid verifying. Chapter 5 is the implementation details of these three solutions based on Android system. Chapter 6 presents the experiment results and discussion. Finally, Chapter 7 concludes this work and future works.

Chapter 2 Background and Related Work

The chapter first describes the challenges of automated GUI testing. It then introduces two well-known open source projects of automatic GUI testing. Finally, related works are described.

2.1 Challenges of automated GUI testing

Based on the automated GUI testing process, we divide it into two parts: reproducing the interaction between human and DUT, and verifying testing results. For a simple event, such as pressing a hardware key, it can be reproduced by inserting a proper event. However, for multi-touch and complicated gestures, it becomes non-trivial to reproduce the interaction between human and DUT due to the timing constraints. In addition to reproduce predefined events accurately, it is also challenging to verify results. There are three commonly used methods for verifying testing results: bitmap comparison, objects identification and optical character recognition (OCR). Although bitmap comparison is easy to implement, it is extremely sensitive to the changes of GUI. The sensitivity can lead to extra maintenance cost, especially for immature software, which is frequently modified over the period of development. Objects identification relies on system provided API (Application Programming Interface) to obtain object information of the screen. By compared with the reference objects, we can evaluate the correctness of the testing results. However, system-provided APIs are very different from OS to OS, which limits the portability of this method. In addition, not all OS provide enough APIs for objects identification. OCR is an alternative method to reduce sensitivity and increase portability, which converts scanned images into machine-encoded text. However, OCR may be slow and inaccurate, especially for non-text content identification.

2.2 Sikuli

Sikuli is a framework [2-4] which automates and tests GUI applications by using images (screenshots) on multiple platforms. Sikuli framework includes Sikuli Script, which is a visual

scripting API for Jython. Sikuli Script supports three types of visual scripting APIs: Find, Actions, and Event Observation. All visual scripting APIs requires an image as an operating target. By using find-type APIs, Sikuli can find target's location, check the existence of target, and wait until the target appears or disappears. In addition, Sikuli can adopt action-type APIs to search and click target, hover mouse pointer to target, or drag-and-drop between two targets. Furthermore, with Event-Observation-type APIs, programmers can register their event handling function to wait target to appear or vanish, or wait for the changes of the GUI contents.

Sikuli framework also contains an IDE, which is an integrated development environment for writing visual scripts with screenshots. By using Sikuli IDE, engineers can easily write GUI test cases, execute the script, automate GUI operations on desktop and verify GUI elements presented on screenshot. Although Sikuli provides several handy functions to ease the process of GUI testing, it's pixel-based image search and comparison mechanism can consume significantly system resources and prolong the testing process. As a result, Sikuli may not be able to apply to embedded systems with limited computation resources. In this work, based on Sikuli, we develop SPAG to support the automated GUI testing of embedded systems.

2.3 Android Software Testing

There are several GUI testing utilities for developers to test their programs, such as Android Instrumentation Framework provided by Android software development kit (SDK) and Robotium. Profiling codes are inserted into proper locations in order to collect necessary information at runtime. These utilities require partial or full source codes of tested applications and are not suitable for black box testing. On the contrary, SPAG does not rely on the availability of source codes. Monkeyrunner is another testing tool provided by Android SDK. This tool can reproduce predefined actions, such as key press and screen touch, by generating associated events. However, Monkeyrunner is sensitive to external interference,

especially under the circumstance that the DUT is in a heavy load condition. SPAG, however, monitors the status of AUT at runtime and dynamically changes the timing to issue commands.

Android 1.6 provides new accessibility features to help users with disabilities use GUI applications. The accessibility features also allow developers to create accessibility services that work in the background and receive notifications of various GUI events. For example, special events are triggered when the state of the activities is changed or some GUI components are focused. These events provide useful hints about the widget where the event originated, such as the type of widget and its text content. However, Android accessibility currently does not allow programmers to list the full contents of the screen. This omission clearly limits the usefulness of accessibility because a GUI screen might have some important widgets, labels or objects that are not focusable.

Hierarchy Viewer is the other utility provided by Android SDK that allows application developers to examine the layout of the Android GUI. Hierarchy Viewer communicates with an Android emulated device through the Android Debug Bridge (adb). Hierarchy Viewer provides the detail information of GUI layout, such as the ID, type, text content, location and size of all the GUI widgets in the screen. However, Hierarchy Viewer can only run on the top of an emulator, which limits its applicability.

2.4 Related Work

There have been many research efforts dedicated to automated GUI testing. The most common approach of automated GUI testing is model-based testing (MBT), which models the behaviors of target software and then uses the test cases generated from the models to validate the DUT [5] [6] [7]. T. Takala et al. adopted Monkey and Window services to generate GUI events [6]. L. Zhifang [8] utilizes the concept of virtual devices to test applications. Their method relies on image-based pattern matching which is sensitive to the quality of images. On the contrary, SPAG uses GUI components for pattern matching in order to improve the

stability and the speed of the validation.

Constructing a universal testing framework has been discussed in [8-11]. Several techniques and architectures were developed to realize complex application test. MoGuT [8], a variant of the FSM based test framework, used image flow to describe event changes and screen response. However, it lacks flexibility. Gray-box testing adopted APIs to construct calling context and parameters from input files [11]. Based on a logging mechanism, the gray-box testing verifies testing results. This method is simple and powerful for testing predictable software components. However, for complex software, it becomes difficult to describe the testing logic and calling context. MoibleTest [7, 9, 10] is a SOA based framework, which includes extendable script interpreter, universal communication interface and agent-based testing mechanism. Although MobileTest can be used for testing mobile devices, the accuracy of the test results is not clear.

Accessibility technologies provide different aids to disabled computer users. Most computing platforms nowadays support accessibility functionalities because it is mandated by the law [12] in U.S.. For GUI testing, the accessibility technology is a useful mechanism that provides an interface for programs to access to GUI objects. Grechanik, et al. [13] used accessibility technologies to obtain GUI structure information of the GUI-based applications (GAPs) for maintaining and evolving test scripts. They also used GUI metadata to generate programming objects in order to automate GUI testing [14] on windows. These research results demonstrate that accessibility technologies are applicable to black-box GUI testing. Recently Chang, et al. [4] claimed that accessibility API can be used to assist pixel-based GUI interpreting in order to obtain a more accurate association between the visual representation and internal structure of a GUI. However, the major limitation of the accessibility-based methods is coverage, which differs from machine to machine. For example, accessibility API on Android 1.6 does not allow developers to list the full contents of the screen. A GUI view might also contain certain widgets or labels that are not focusable, and are thus inaccessible. As a result, there is a clear need to develop a new method to overcome the limitation.

Chapter 3. Definitions and Problem Statement

3.1 Definitions

We adopt a commonly used software testing technique called record-replay for embedded systems, which includes record stage and replay stage. In the record stage, shown in Figure 1(a), the screen of the DUT is first redirected to the host PC, on which the test tool runs. The test engineer then interacts with the DUT remotely. Whenever the engineer performs a GUI action on the host PC, such as key press and finger touch, the test tool sends associated GUI event sequences to the DUT in order to control the DUT on live. The performed GUI actions are also saved into test cases with verifications. In the replay stage showed in Figure 1(b), the test executor reads GUI actions and replays them on the DUT. The test executor then verifies the testing results based on the response of the DUT.

Based on record-replay technique, engineer generates a GUI test case by recording test steps and adding verifications into it. In order to generate executable test cases, the GUI actions and verifications are stored as GUI operations and verify operations respectively, where executing a GUI operation will reproduce the GUI actions recorded by tool and

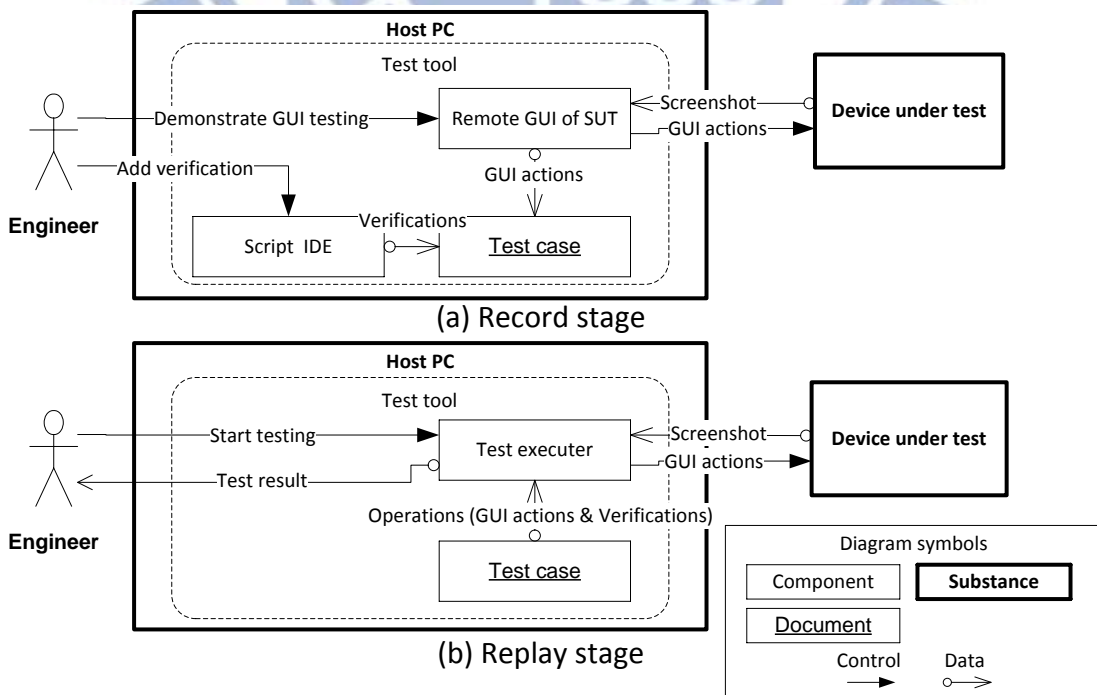


Figure 1. System architecture of record/replay method with DUT

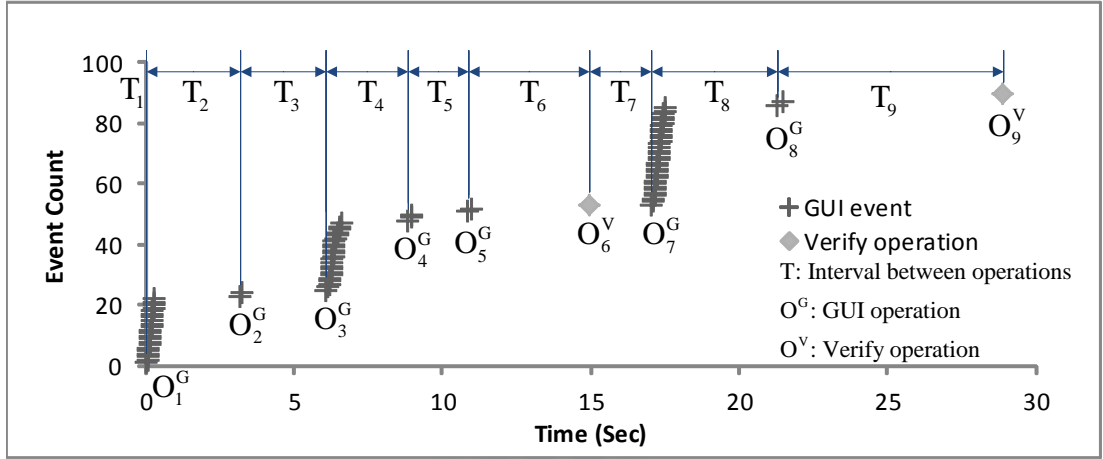


Figure 2. Operations in a GUI test case C_1

executing verify operation will performs the verification inserted by engineer. In additional, the intervals between each operation are also specified in test case, in order to let DUT has enough time to respond to operations. Following the description above, we define C as a GUI test case, which contains several operations and intervals between each two adjacent operations. We define the pattern of C by regular expression $(T(O^G|O^V))^+$, in which O^G is a GUI operation used to reproduce desired GUI actions such as key press and finger touch, O^V is a verify operation used to verify the DUT's response is desired one or not, and T is the interval between the occurrence times of each two adjacent operations. If necessary, we will use O to represent an operation which cloud be O^V or O^G in the following description. With the defined pattern $(T(O^G|O^V))^+$, a test case can be denoted by a sequence of intervals and operations $T_1, O_1, \dots, T_i, O_i^G, \dots, T_j, O_j^V, \dots, T_n, O_n$, in which O_i^G is a GUI operation defined above and also the i -th operation in the sequence of C , O_j^V is a verify operation defined above and also the j -th operation in the sequence of C and T_i is the interval between the occurrence times of O_{i-1} and O_i . For example, the test case showed in Figure 2 can be denoted by $T_1, O_1^G, T_2, O_2^G, T_3, O_3^G, T_4, O_4^G, T_5, O_5^G, T_6, O_6^V, T_7, O_7^G, T_8, O_8^G, T_9, O_9^V$.

To define the GUI operation in detail, a GUI operation is actually constituted by many GUI events. Similarly, delays between the occurrence times of each two adjacent GUI events are stored into test case in order to mimic the desired GUI action when executing test case.

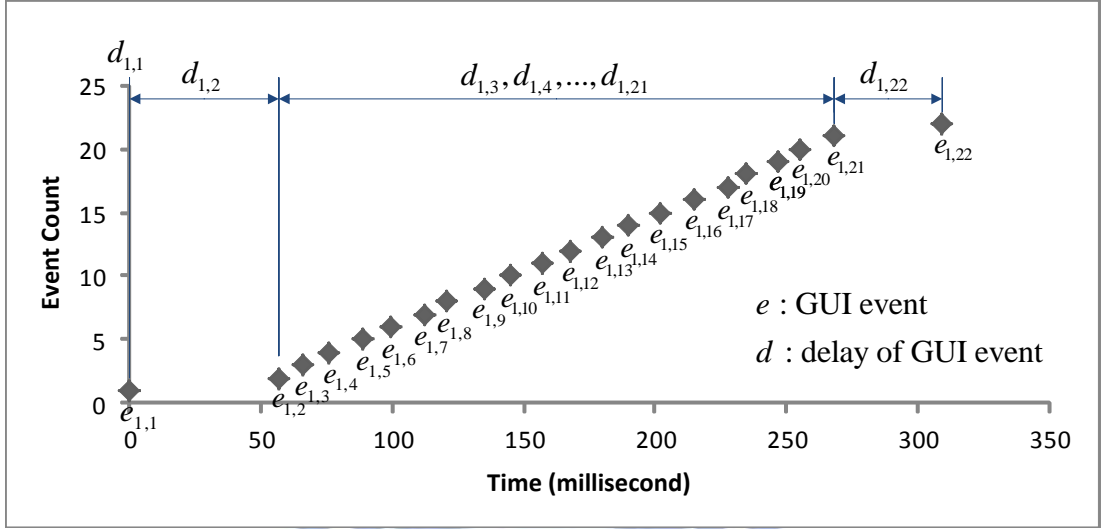


Figure 3. GUI events in GUI operation O_1^G

We define the pattern of O^G by regular expression $(de)^+$, in which e is a GUI event and d is a delay between the occurrence times of each two adjacent GUI events. In additional, we define O_{i,m_i}^G as the i -th GUI operation which consists of m_i GUI events to be reproduced. With the defined pattern $(de)^+$, a GUI operation O_{i,m_i}^G can be denoted by sequence $d_{i,1}, e_{i,1}, \dots, d_{i,m_i}, e_{i,m_i}$, in which the i means O_{i,m_i}^G is the i -th GUI operation in C , m_i is the number of events in this GUI operation, $e_{i,j}$ is the j -th event and $d_{i,j}$ is the delay between the occurrence times of $e_{i,j-1}$ and $e_{i,j}$. For example, Figure 3 shows the first GUI operation $O_{1,22}^G$ in C_1 with detail timing of GUI events, in which the x-axis is the execution time of the GUI operation, y-axis is the event count number and each data point is a GUI event. With the defined pattern $(de)^+$, the $O_{1,22}^G$ showed in Figure 3 can be denoted by $d_{1,1}, e_{1,1}, \dots, d_{1,22}, e_{1,22}$, in which $d_{i,j}, e_{i,j}$ are defined above.

In replay stage, all operations are scheduled with predefined intervals and all GUI events are scheduled with predefined delays. Since all intervals and delays may have error during replaying, we use C' to describe the executed C . We define the pattern of C' by regular expression $(T'(O'^G|O'^V))^+$ and the pattern of O'^G by regular expression $(d'e')^+$, in which O'^G is a executed GUI operation generated from executing O^G , O'^V is a executed verify operation with the verifying result generated from executing O^V , and T' is the

reproduced interval between the occurrence times of each two adjacent executed operations. If necessary, we will use O' to represent an operation which could be O'^G or O'^V in the following description. With the defined pattern $(T'(O'^G|O'^V))^+$, a test case can be denoted by a sequence of reproduced intervals and executed operations $T'_1, O'_1, \dots, T'_i, O'_i, \dots, T'_j, O'_j, \dots, T'_n, O'_n$, in which O'_i is an executed GUI operation defined above and also the i -th executed operation in C' , O'_j is an executed verify operation with verifying result defined above and also the j -th executed operation in C' . For example, executing C_1 showed in Figure 2 will generate C'_1 , which can be denoted by $T'_1, O'^G_1, T'_2, O'^G_2, T'_3, O'^G_3, T'_4, O'^G_4, T'_5, O'^G_5, T'_6, O'^V_6, T'_7, O'^G_7, T'_8, O'^G_8, T'_9, O'^V_9$. With the defined pattern $(d'e')^+$, an executed GUI operation O'^G_{i,m_i} can be denoted by sequence $d'_{i,1}, e'_{i,1}, \dots, d'_{i,m_i}, e'_{i,m_i}$, in which the i means O'^G_{i,m_i} is the i -th executed GUI operation in C' , m_i is the number of events in this GUI operation, $e'_{i,j}$ is the j -th reproduced event and $d'_{i,j}$ is the delay between the occurrence times of $e'_{i,j-1}$ and $e'_{i,j}$. For example, the executed GUI operation $O'^G_{1,22}$ generated by executing $O^G_{1,22}$ showed in Figure 3 can be denoted by $d'_{1,1}, e'_{1,1}, \dots, d'_{1,22}, e'_{1,22}$.

In contrast to the past work focused on the accuracy of bug detection, this work addresses the accuracy of testing method, thus we only test bug-free test cases and consider that the accuracy of all verify operations are a little and can be ignored. In practical, the GUI testing procedure retries failed test case to ensure that the failure is due to bugs from AUT rather than other issues, such as system becomes unresponsive under heavy disk I/O. If a test case fails first time and passes on retry, the failure may be considered as a false positive result. Since a retry is raised by failed test and each retry costs extra time, an efficient test method should cost as little time as possible on a test case while keep the test success rate.

In order to evaluate the accuracy of the GUI testing method, we use $C_1 \dots C_n$ to denote n test case. The test procedure repeats testing each test case until a passing result is obtained.

We define $C'_{i,1} \dots C'_{i,k_i}$ as k_i executed test case generated by executing C_i , in which the first $k_i - 1$ testing are failed and generate executed test cases $C'_{i,1} \dots C'_{i,k_i-1}$ until the last testing returns passing result and generate executed test case C'_{i,k_i} . In order to quantify the quality of a test method, we define $ACC(M) = \frac{n}{\sum_{i=1}^n k_i}$ to evaluate the test accuracy, or the success ratio of executing bug-free test cases, in which M is the test method used in testing, n is the number of test cases and k_i is how many times C_i is reproduced. We also define $EFF(M) = \frac{\sum_{i=1}^n T(C_i)}{\sum_{i=1}^n \sum_{j=1}^{k_i} T(C'_{i,j})}$ to evaluate the test efficiency, or the ratio of expect test time against the real test time of test cases, in which M is the test method used in testing, $T(C_i)$ is expected test time of C_i and $T(C'_{i,j})$ is real test time of the j -th reproduced result of C_i . For example, if we use method m to execute single test case C_1 with 10 seconds expected test time. If the test fails 4 times before passes and takes 2,2,7,5,11 second respectively, the $ACC(M)$ is $\frac{1}{5} = 20\%$ and the $EFF(M)$ is $\frac{10}{2+2+7+5+11} \approx 37\%$.

Table I. List of notations and definitions used in problem statement

Symbol	Definition
C	A test case generated based on record-replay technique. In general, C consists of several GUI operations, verify operations with proper intervals between each two adjacent operations.
$C_1 \dots C_n$	n test cases
$(T(O^G O^V))^+$	A regular expression describing the pattern of C .
O	A operation which could be O^G or O^V .
T	An interval between the occurrence times of each two adjacent O .
O^G	A GUI operation used to reproduce desired GUI actions such as key press and finger touch.
O^V	A verify operation used to verify the DUT's response is desired one or not.
$T_1, O_1, \dots, T_i, O_i^G, \dots,$ $T_j, O_j^V, \dots, T_n, O_n$	A sequence which denotes C .
O_i	The i -th operation in the sequence denoted C , which could be O_i^G or O_i^V .

T_i	The i -th interval between the occurrence times of O_{i-1} and O_i in the sequence which denotes C .
O_i^G	O_i^G is O_i in the sequence which denotes C , which is also a GUI operation.
O_i^V	O_i^V is O_i in the sequence which denotes C , which is also a verify operation.
$(de)^+$	A regular expression describing the pattern of O^G .
O_{i,m_i}^G	O_{i,m_i}^G is O_i^G in the sequence which denotes C , in which m_i is the number of events also the number of delays in O_i^G .
$d_{i,1}, e_{i,1}, \dots,$ d_{i,m_i}, e_{i,m_i}	A sequence which denotes O_{i,m_i}^G .
$e_{i,j}$	The j -th GUI event in $d_{i,1}, e_{i,1}, \dots, d_{i,m_i}, e_{i,m_i}$, which the basic element to compose a GUI action.
$d_{i,j}$	The j -th delay in $d_{i,1}, e_{i,1}, \dots, d_{i,m_i}, e_{i,m_i}$, which is between the occurrence times of $e_{i,j-1}$ and $e_{i,j}$.
C'	A executed test case, which consists executed GUI operation, executed verify operations with verifying result and reproduced intervals between each two adjacent operations.
$C'_{i,1} \dots C'_{i,k_i}$	k_i executed test case generated by executing C_i for k_i times.
$C'_{i,j}$	The j -th executed test case generated by executing C_i
$(T'(O'^G O'^V))^+$	A regular expression describing the pattern of C'
O'	A operation which could be O'^G or O'^V
T'	An interval between the occurrence times of each two adjacent O' .
O'^G	A executed GUI operation generated by executing O^G .
O'^V	A executed verify operation generated by executing O^V . O'^V also contains the verifying result from executing O^V .
$T_1, O_1, \dots, T_i, O_i^G, \dots,$ $T_j, O_j^V, \dots, T_n, O_n$	A sequence which denotes C' .
O'_i	The i -th executed operation in the sequence denoted C' , which could be O'^G_i or O'^V_i .
T'_i	The i -th reproduced interval between the occurrence times of O'_{i-1} and O'_i in the sequence which denotes C' .
O'^G_i	O'^G_i is O'_i in the sequence which denotes C' , which is also an executed GUI operation.

O_i^V	O_i^V is O_i' in the sequence which denotes C' , which is also an executed verify operation.
$(d'e')^+$	A regular expression describing the pattern of O'^G .
O_{i,m_i}^G	O_{i,m_i}^G is O'^G in the sequence which denotes C' , in which m_i is the number of reproduced events also the number of reproduced delays in O'^G .
$d'_{i,1}, e'_{i,1}, \dots,$ d'_{i,m_i}, e'_{i,m_i}	A sequence which denotes O_{i,m_i}^G .
$e'_{i,j}$	The j -th reproduced GUI event in $d'_{i,1}, e'_{i,1}, \dots, d'_{i,m_i}, e'_{i,m_i}$, which the basic element to compose a GUI action.
$d'_{i,j}$	The j -th reproduced delay in $d'_{i,1}, e'_{i,1}, \dots, d'_{i,m_i}, e'_{i,m_i}$, which is between the occurrence times of $e'_{i,j-1}$ and $e'_{i,j}$.
ACC(M)	The test accuracy, or the success ratio of executing bug-free test cases, in which M is the test method used in testing.
EFF(M)	The test efficiency, or the ratio of expect test time against the real test time of test cases, in which M is the test method used in testing

3.2 Problem Statement

Given a test case C with the pattern of test case $(T(O^G|O^V))^+$ and the pattern of GUI operation $(de)^+$. For example, the test case C_1 showed in Figure 2 can be denoted by $T_1, O_1^G, T_2, O_2^G, T_3, O_3^G, T_4, O_4^G, T_5, O_5^G, T_6, O_6^V, T_7, O_7^G, T_8, O_8^G, T_9, O_9^V$ and the GUI operation showed in Figure 3 can be denoted by $d_{1,1}, e_{1,1}, \dots, d_{1,22}, e_{1,22}$. We design a test system to record C and replay C with replay method M , aim to increase the ACC(M) and EFF(M) on embedded system GUI testing.

Chapter 4. SPAG Design

Accurately reproducing GUI operations and dynamic controlling intervals between operations are two key design requirements of SPAG. In this chapter, we design Smart Phone Automated GUI (SPAG) to improve the accuracy of reproducing GUI operation and assist writing of verify operations automatically. SPAG includes three key mechanisms. They are event batching, smart waiting and hybrid verifying.

In the SPAG system design, all key mechanisms have record stage and replay stage. In order to store test cases in record stage and execute test cases in replay stage, SPAG records test case in form of $(T O)^+$, transforms all T , O^G , O^V into commands and store these commands into a test script in record stage. The commands are executed to perform the test later in replay stage. In record stage, SPAG accepts O_i^V added by user or SPAG itself to generate $CMD(O_i^V)$. the $CMD(O_i^V)$ performs GUI verification. SPAG monitors T_i and cpu_i to generate $CMD(T_i, cpu_i)$, in which T_i is i -th interval between the occurrence time of each operation and cpu_i is the target application's CPU usage during T_i . The $CMD(T_i, cpu_i)$ delays all following commands for T_i or more time depend on the actual CPU usage cpu_j' during testing. For a GUI operation, SPAG tracks the delay between each recorded GUI event and generate a sequence of commands in order to replay the GUI events with their delay in replay stage. SPAG receives $O_{i,m_i}^G(d_{i,1}, e_{i,1}, \dots, d_{i,m_i}, e_{i,m_i})$ and transforms O_{i,m_i}^G into $CMD(d_{i,1}, e_{i,1}), \dots, CMD(d_{i,m_i}, e_{i,m_i})$, in which m_i is the number of delays and events contained in O_{i,m_i}^G .

Table II. List of commands used in SPAG design

Command	Definition
$CMD(v_i)$	A command used to perform the v_i , in which v_i is i -th verify operation in a test case.
$CMD(T_i, cpu_i)$	A command used to delay its next command for T_i , in which T_i is i -th interval between occurrence of operations and cpu_i it the CPU usage of target application during interval T_i .
$CMD(d_{i,j}, e_{i,j})$	A command used to reproduce $e_{i,j}$ with $d_{i,j}$, in which $e_{i,j}$ is l -th GUI event in k -th GUI operation in a test case and $d_{i,j}$ is the delay of $e_{i,j}$.

4.1 Batch Event

In practice, A GUI operation may consist of more than one GUI event. The application under test (AUT) monitors incoming GUI events and recognizes GUI operations among the GUI events. For example, when a user performs a gesture like a swipe action on the Android operation system, the system pulling multiple touch events sampled from hardware and dispatch events to current on-top application. The on-top application keeps tracking GUI events internally to recognize GUI operations, or gestures, among the received GUI events. Since GUI operations are recognized inside of application, SPAG can't directly reproduce GUI operations in a black box testing. As a substitute, SPAG records GUI events on host PC and replays GUI events into DUT, thus the GUI events are dispatched to AUT and trigger the desired GUI operations inside of AUT.

In order to trigger the GUI operation $O_{i,m_i}^G(d_{i,1}, e_{i,1}, d_{i,2}, e_{i,2}, \dots, d_{i,m_i}, e_{i,m_i})$ correctly, SPAG has to reproduce $e_{i,1}, e_{i,2}, \dots, e_{i,m_i}$ into the mobile device on time. However, for time-sensitive GUI operation, such as onFling gesture, the reproduced delays times $d'_{i,1}, d'_{i,2}, \dots, d'_{i,m_i}$ between GUI events need to be absolutely exact as $d_{i,1}, d_{i,2}, \dots, d_{i,m_i}$. Otherwise, the O'_{i,m_i}^G may trigger an GUI operation with different property. For example, the onFling gesture has properties velocityX and velocityY. Both velocities are calculated from the displacement and time difference between GUI events. Therefore, velocityX and velocityY may be different if the reproduced delays vary for each reproduced onFling gesture. A conventional approach is to wait $d_{i,1}$, inject $e_{i,1}$ and go on, but this approach may be

error-prone, especially for mobile applications. For example, Figure 4 shows two approaches used to replay GUI events with the control delay variation, which cause the control data transferring between host PC and device vary from time to time. Figure 4 shows the two approaches to reproduce a GUI operation $O_{i,5}^G(d_{i,1}, e_{i,1}, \dots, d_{i,5}, e_{i,5})$. The conventional approach in Figure 4(a) sends $e_{i,1}, \dots, e_{i,5}$ to DUT with $d_{i,1}, \dots, d_{i,5}$ separately. The delay between transferred GUI events $d'_{i,2}, \dots, d'_{i,5}$ are affected by the control delay variation and become different from $d_{i,2}, \dots, d_{i,5}$. As a result, $O'_{i,5}^G$ may triggers a different GUI operation with different property. On the contrary, our proposed batch event in Figure 4 (b) transfers all GUI events and their delays in a batch to reduce the effect of the control delay variation.

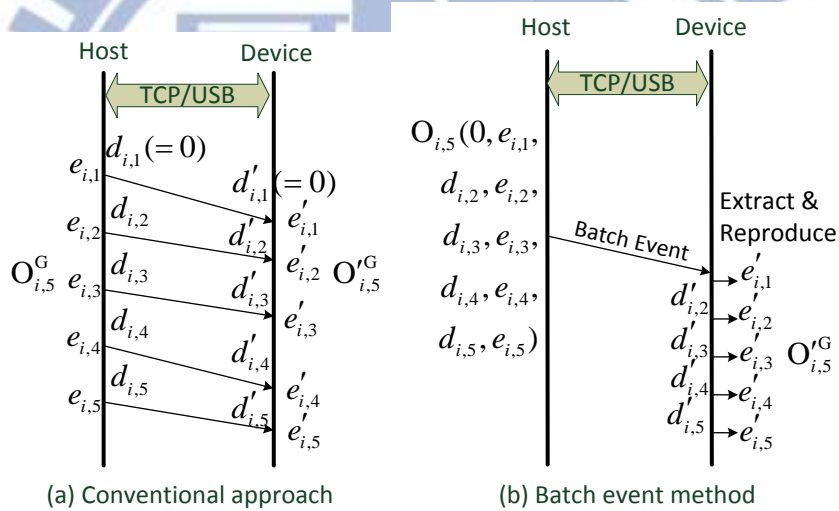


Figure 4. Two approaches for sending event sequence

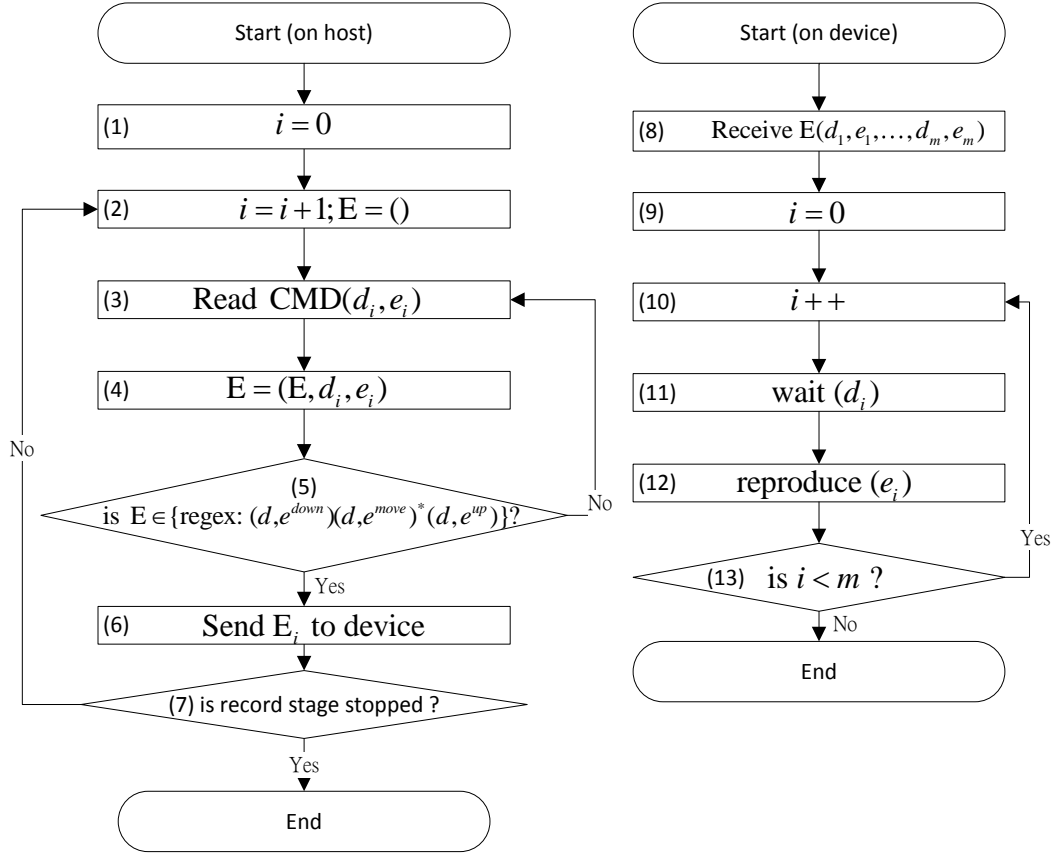


Figure 5. Flow of Batch Event method in replay stage

Figure 5 illustrates how batch event replays GUI actions, in which SPAG keeps reading $CMD(d, e)$ from the test script (step 3) and appends them to a sequence container E (step 4) until the sequence of E matches a predefined pattern $(d, e^{down})(d, e^{move})*(d, e^{up})$ (step 5). The pattern is defined in Android framework used to identify any gestures among all inputs, in which d is GUI event delay, $e^{down}, e^{move}, e^{up}$ are GUI events with action state $ACTION_DOWN$, $ACTION_MOVE$ and $ACTION_UP$ respectively. When the pattern of gesture is matched, SPAG sends E through a UAB cable to DUT in once (step 6). Meanwhile, the device receives (step 8) and reproduces each events e_1, e_2, \dots, e_m with delay d_1, d_2, \dots, d_m respectively (step 9 to 13) to trigger expected GUI operation on AUT. Therefore, SPAG reduces the variance between d_1, d_2, \dots, d_m by sending control data at once rather than several times for each GUI operation.

4.2 Smart Wait

By experience of our smart phone test team, it is more frequent that GUI testing gets

failure when DUT is under a heavy loading. Therefore, engineers usually test the test case for several rounds to tune the delays until they are long enough to tolerate the delays may happened on DUT but as short as possible for the test efficiency. The efficiency problem can be solved by executing commands with event driven, which is to keep waiting until the expected event comes, then continue to next command. However, the event driven approach is not applicable in the black-box or gray-box testing because the accessible text-based events may not cover all state transitions. Even if the image comparing is used to fill the rest coverage, the large number of images is too trivial to create and maintain.

In order to reduce the cost on writing a test case and to improve the test efficiency, our Smart Wait automatically handles application delay by extending part of the waiting times in the test case based on the AUT's CPU usage. For example, if there is a test case containing 10 steps and each step delays 7 seconds, the total execution time is around 70 second. However, the engineer needs to extend each delay to 20 seconds to avoid any failures happened due to the application delay. The modified test case now spends 200 seconds to finish testing, which is 186% more time than original. On the contrary, the smart wait method uses original delays and extends only a part of delays during testing. If 20% of delays are extended to 20 second by smart wait, the average test execution time will be 96 seconds, which is still slower than original but faster than conventional approach. In addition, it also reduces the process of test case tuning. The reason SPAG dose not monitor on other system resource such like disk IO is because the AUT still needs CPU resources to process other system resource. For example, to downloading a file with size of 1 megabyte, AUT always costs similar CPU time no matter how much real time spends on waiting for data.

In record stage, SPAG tracks the cpu_i and T_i between O_{i-1} and O_i and generate $CMD(T_i, cpu_i)$ in script. When replay stage starts, SPAG also tracks cpu'_i during executing operation O_{i-1} and judge whether or how long to wait based on T_i . When AUT dose not finish O_{i-1} yet, the cpu'_i is expected to be smaller than cpu_i , and if it does, SPAG will

postpone all rest operations for a certain time to wait AUT processing O_{i-1} . For example, in record stage, if AUT uses 5 milliseconds CPU time during 4 seconds executing time on an operation O_{i-1} , SPAG will generate command SmartWait(4000 ms, 5 ms) after generated CMD(O_{i-1}) in test case. During testing, SPAG also tracks cpu'_i on DUT. When SPAG reads to SmartWait(4000 ms, 5 ms), it waits for 4 seconds then checks cpu'_i . If O_{i-1} only cost 2 ms when time is up to 4 second, SPAG will estimate a new execution time by calculating $4 \text{ sec} \times \frac{5\text{ms}}{2\text{ms}} = 10 \text{ sec}$ and postpone all following operations for 6 second more to wait O_{i-1} to finish.

```

Function SmartWait( $T_i, cpu_i$ )
1  wait( $T_i$ )
2   $T'_i = T_i$ 
3   $cpu'_i = cpu^{\text{now}} - cpu^{\text{lastChecked}}$ 
4  while  $cpu'_i < cpu_i$  and  $d'_i < TIMEOUT$ 
5      do  $T_i^{\text{new}} = T'_i \times \frac{cpu_i}{cpu'_i}$ 
6          wait( $T_i^{\text{new}} - T'_i$ )
7           $T'_i = T_i^{\text{new}}$ 
8           $cpu'_i = cpu'_i + cpu^{\text{now}} - cpu^{\text{lastChecked}}$ 
9  return

```

Figure 6. pseudo code of Smart Wait

Figure 6 shows the pseudo code of smart wait approach. The function SmartWait(T_i, cpu_i) is called every time when SPAG reads CMD(T_i, cpu_i), in which T_i is the interval between O_{i-1} and O_i and cpu_i is the CPU time cost on O_{i-1} . After waiting for predefined T_i (line 1), SPAG gets the actually consumed CPU time cpu'_i during T_i (line 3) and compares it with cpu_i . If the actual consumption is small then expected, SPAG will decide to wait more time (line 4) and start estimating a longer interval time T_i^{new} (line 5), in which the T_i^{new} is the new expected interval estimated by equation $\frac{T'_i}{cpu'_i} = \frac{T_i^{\text{new}}}{cpu_i}$. This equation assumes that AUT will use the same speed to consume the CPU time in the rest time of processing O_{i-1} , thus AUT will finally reach consumption cpu_i at T_i^{new} . Finally, SPAG waits another time $T_i^{\text{new}} - T'_i$ for AUT (line 6). However, the actual executing time should have a distribution around the idea executing time. The probability of that estimated executing time is actually on time is 0.5. Therefore, SPAG performs another round of checking and

waiting begins (line 4,8,4,5,6) and repeat the procedure until $cpu'_i \geq cpu_i$ or reaches predefined timeout, in order to approach the actual executing time of O_{i-1} .

4.3 Hybrid Verifying

By using accessibility APIs, SPAG can access the GUI layout data, such as activity name and listed items. Retrieving and comparing GUI layout data can be very quick, while the pixel-based method takes relative long time on image transferring and processing. Therefore, using accessibility technology to verify GUI testing result is faster than using pixel-based method. Based on accessibility APIs, SPAG automatically generates command of verification operations when the DUT GUI layout is changed. This mechanism is greatly helpful when writing GUI test cases. However, the accessibility technology may have access limitations from machine to machine. In order to improve coverage of GUI verification, we use pixel-based method to verify those accessibility technology cannot apply on. For example, the accessibility API on Android cannot access GUI elements not in focus, such like elements

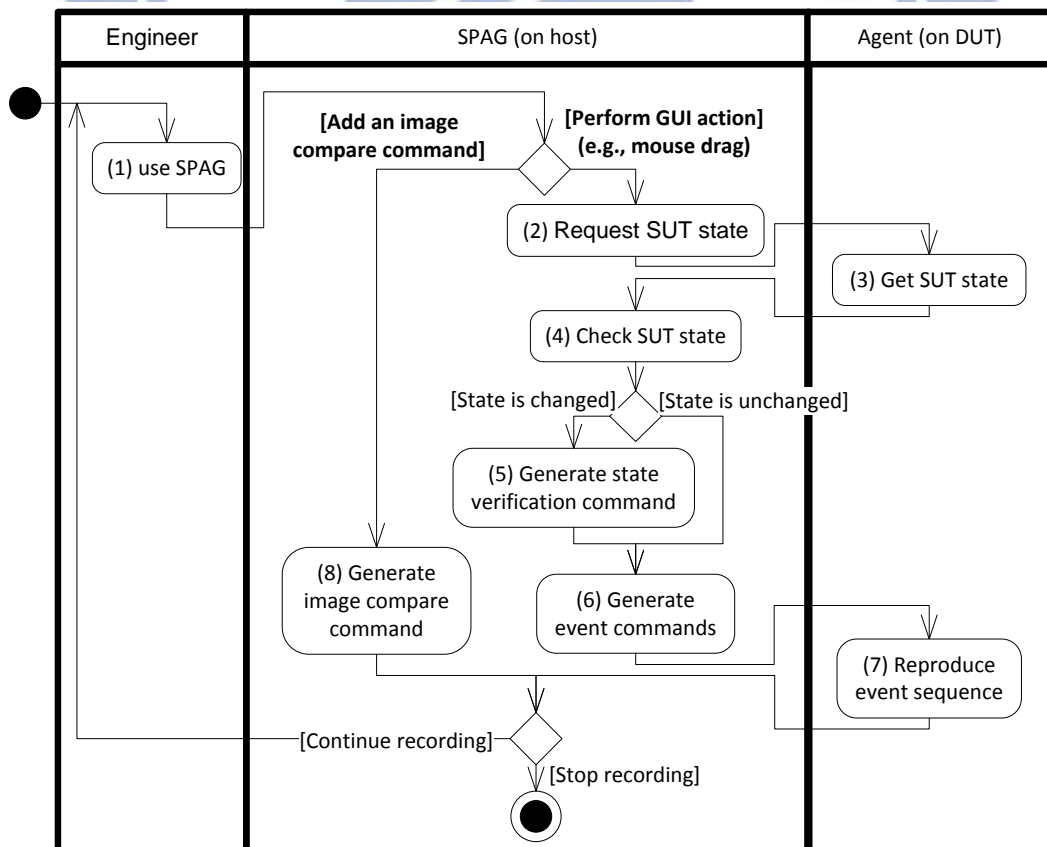


Figure 7. Flow of Hybrid Verifying method in record stage

behind a popup dialog. The engineer needs to add an image-comparing command manually, which contains the image of expected element layout, in order to verify such element with image.

The hybrid verifying works in record stage, in which SPAG monitors any changes on AUT's GUI layout, or AUT state, and automatically generate verify operations into test case. In order to save the generated operations for replaying latter, SPAG converts those operations into commands and saves them into a test script.

Figure 7 shows the flow of hybrid verifying in record stage, in which the engineer uses SPAG to record the test case by demonstration. some image-based verify operations may be inserted during the demonstration (step 1). In most of time, SPAG receives GUI operations performed by engineer and prepares to generate commands for each operation. After O_i is committed by engineer, hybrid verifying starts to check if the GUI layout has been changed or not since O_{i-1} has been performed. The checking procedure includes sending a GUI state request to the agent (step 2), the agent retrieve GUI layout data by accessibility API and returns to host PC (step 3), and SPAG compares the current GUI state after O_{i-1} with the GUI state checked before O_{i-1} is performed (step 4). If the GUI state is changed, SPAG will generated a command of verify operation with current GUI layout data, in order to verify if same change happens in replay stage (step 5). After the checking procedure finishes, SPAG then generates commands from received O_i into the script (step 6). Finally, The O_i is sent to DUT to be performed in order to archive the live interaction between the engineer and the DUT during record stage (step 7). The engineer may add an image based verify command in order to verify those GUI layout not covered by accessibility API based verify command (step 8).

Because SPAG generates all commands in proper order during record stage, it can execute the script without reordering any command to perform the testing. Figure 8 shows the flow of hybrid verifying working in replay stage. When test starts (step 1), SPAG starts

reading commands in the executing script one by one (step 2). Based on each command read from script, SPAG may verify the DUT state using same method described in record stage (step 3,4,5), handles a single GUI event as mentioned in section 4.1 (step 6,7) or verify the GUI layout by pixel-based method (step 8). The testing continues until any verification fails or all commands in the script are executed.

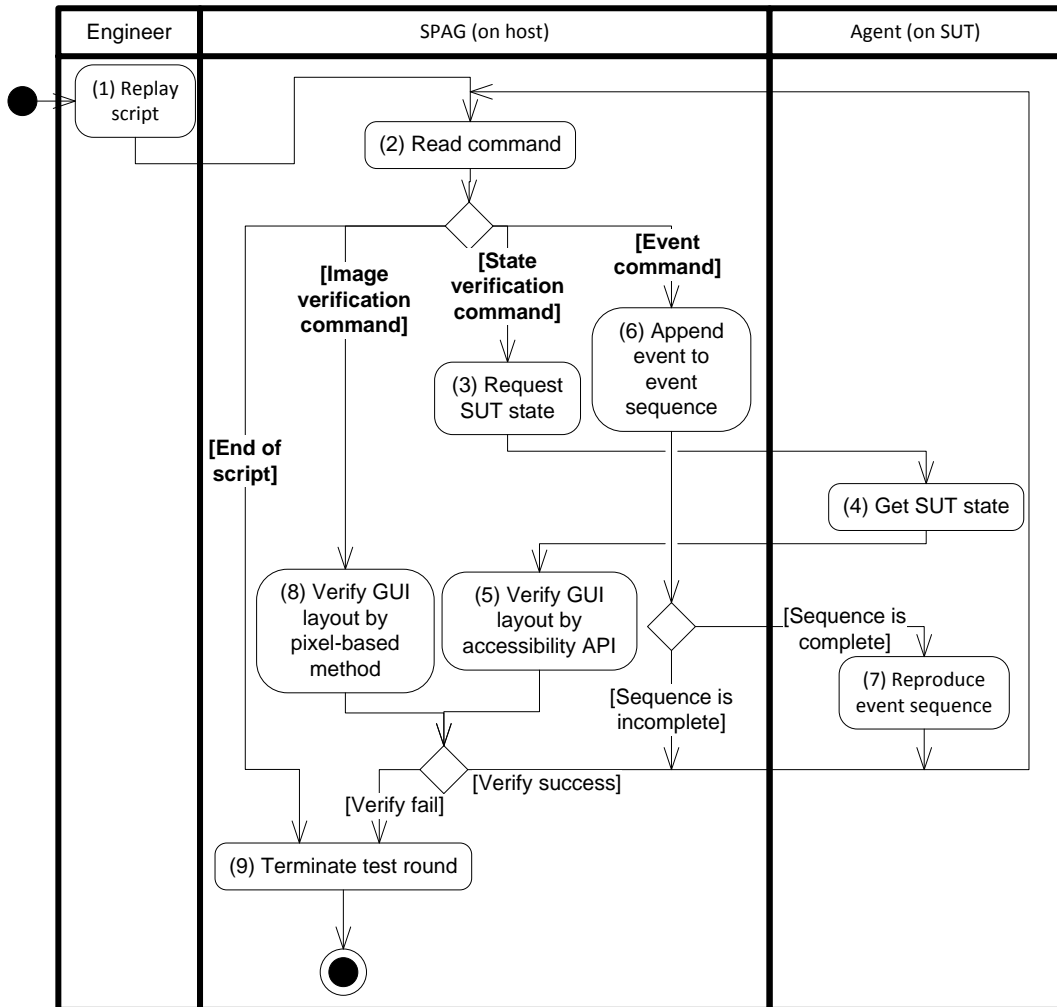


Figure 8. Flow of Hybrid Verifying method in replay stage

Chapter 5. Implementation

In this section, we first give an overview of our framework: Smart Phone Automated GUI testing tool (SPAG) while we introduce the mechanism of batch event and dynamic delay estimating. Finally, we describe the methodology of hybrid testing verification.

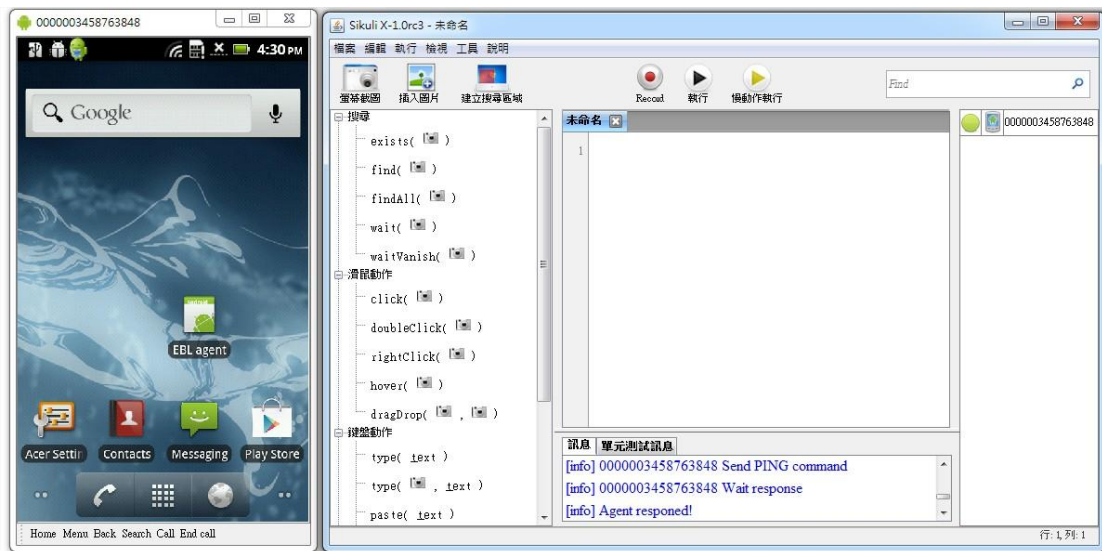


Figure 9. Screenshot of SPAG with screen of DUT

Figure 9 shows the result of implementation, in which our SPAG provides two windows on the host PC machine. The main window in right side is our test case editor based on Sikuli IDE, which records any GUI operations during record stage and generates corresponding commands in the script text area. The sub window on left side is our device window. The device window shows remote screenshot of DUT and updates its screenshot in 1 or 2 times per second, which allows engineers to interact with DUT through it.

5.1 Device under Test

We adopt Acer Liquid, a popular and powerful smart phone, as our DUT. The Acer Liquid quips with a Qualcomm 8250 768MHz processor, a 512 MB Flash ROM, a 256 MB RAM and a Wi-Fi IEEE 802.11 b/g interface. The operating system of Acer Liquid is Android 2.2. We use Java language to implement SPAG framework, which is a user space program with root privilege.

5.2 Recording and Replaying Events Sequence

Recording input events from live demo.

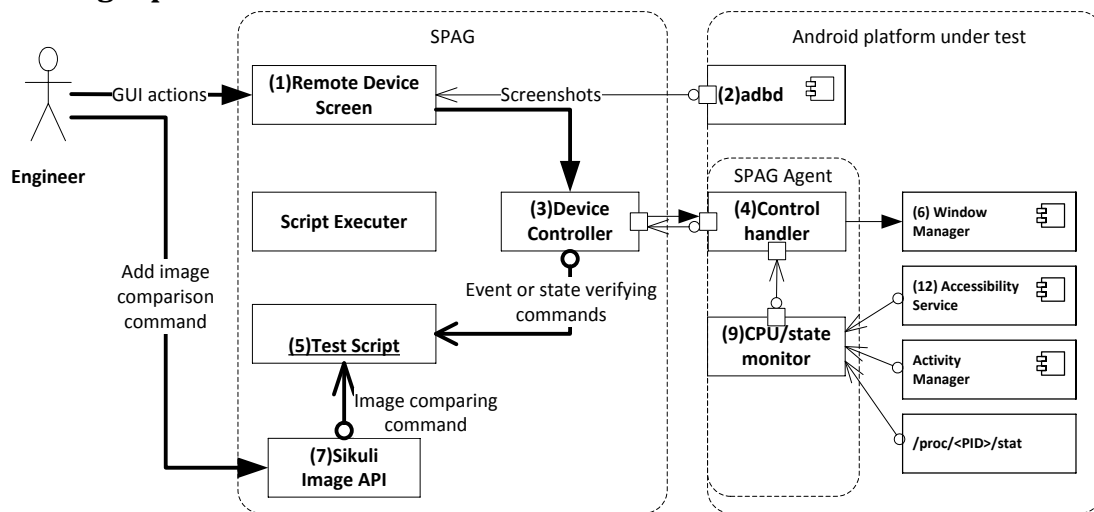


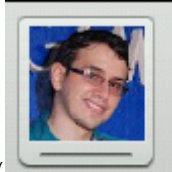
Figure 10. SPAG architecture in record stage

Figure 10 shows the architecture of SPAG in record stage, in which the DUT screen listens for any GUI operations comes from engineer (engineer->1) and update the screen image by calling the *screencap* function of adb in time (2->1). When the DUT screen receives any GUI operation, it passes the operations to device controller(1->3). The device controller sends those operations to the agent(3->4) and generates commands into test script(3->5). The control handler in agent receives operations and reproduce them by inject each GUI event into DUT(4->6) to trigger the desired GUI operations in order to record the GUI session on live. During the demonstrating, the engineer may insert the image comparing command after any GUI operation. When the insert button in Sikuli IDE is pressed(engineer ->6), it triggers Sikuli's command inserting procedures to lead engineer to select the region on the screen that engineer wants to verify in replay stage. SPAG then generates the pixel-based verify command and insert it into current editing script(6->5).

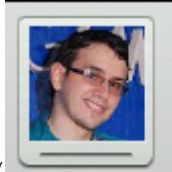
```

Device_0000003458763848 = AndroidDevice.getDevice("0000003458763848")
Device_0000003458763848.sleep( 2213, 28)
Device_0000003458763848.verify("com.android.contacts",
"com.android.contacts.ContactsListActivity")
Device_0000003458763848.pointer( 0, ACTION_DOWN, 197, 625)
Device_0000003458763848.pointer( 81, ACTION_UP, 197, 625)
Device_0000003458763848.sleep( 3376, 33)
Device_0000003458763848.verify("com.android.contacts",
"com.android.contacts.DialtactsActivity")
Device_0000003458763848.pointer( 0, ACTION_DOWN, 244, 664)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 246, 664)
Device_0000003458763848.pointer( 8, ACTION_MOVE, 247, 659)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 249, 647)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 249, 621)
Device_0000003458763848.pointer( 5, ACTION_MOVE, 249, 583)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 249, 546)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 254, 496)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 262, 438)
Device_0000003458763848.pointer( 5, ACTION_MOVE, 270, 389)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 275, 355)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 282, 323)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 285, 300)
Device_0000003458763848.pointer( 5, ACTION_MOVE, 286, 284)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 288, 274)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 289, 262)
Device_0000003458763848.pointer( 6, ACTION_MOVE, 290, 256)
Device_0000003458763848.pointer( 6, ACTION_UP, 290, 256)
Device_0000003458763848.sleep( 5524, 122)
Device_0000003458763848.pointer( 0, ACTION_DOWN, 211, 444)
Device_0000003458763848.pointer( 70, ACTION_UP, 211, 444)

```



```

if exists():
    print 'found him!'

```

Figure 11. Example of SPAG's test script

During the recording, SPAG will generate a test script such like Figure 11. In Figure 11, the commands the commands with prefix *Device_0000003458763848* are all generated by SPAG, which consist of three types of command: pointer, verify, and sleep. The three types represent $CMD(d_{i,j}, e_{i,j})$, $CMD(v_i)$ and $CMD(T_i, cpu_i)$ of our design in chapter 4. In addition, engineer may manually insert image compare command such like

```

if exists()

```

to verify some GUI layout can't be verify by accessibility API.

Reproducing Event Sequence with Batch Event

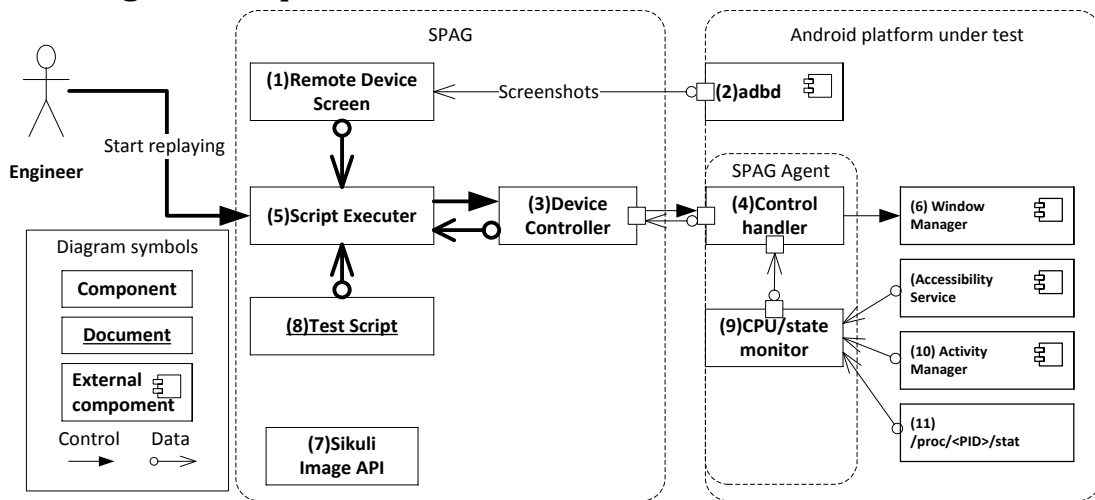


Figure 12. SPAG architecture in replay stage

Figure 12 shows how SPAG works in the replay stage, in which SPAG executes the test script and interacts with DUT. When the engineer starts the testing with a test script(engineer->5), the script executer starts reading commands from the specified test script(8->5). For each GUI event command, script executer calls the device controller to perform the reproducing(5->3). However, the batch event method buffers all GUI events until the events pass a predefined parsing rule, which is a pattern defined for a gesture: $(d, e^{down})(d, e^{move})^*(d, e^{up})$. The pattern is described in the reference of SimpleOnGestureListener[15] on Android's develop supporting website. The SimpleOnGestureListener is the default component used to detecting gesture in Android framework. When the GUI event matches the pattern, device controller will groups all events into a batch and sends the batch to the control handler(3->4) to reproduce the action on DUT. The batch event controlling procedure in both stage are the same, thus the GUI controlling in both stages are the same feeling to DUT.

5.3 Smart Wait

To implement Smart Wait algorithm, SPAG traces the delay between each command sent to agent and the CPU time used to process each action to estimate the response time in record stage. A piece of code is inserted into begin of the event sending function in the device controller(3), thus command delay can be calculated from the difference between each

command sending time. In order to get the command processing time in replay stage, the CPU/state monitor(9) obtains the PID of current process through the activity manager(10) to retrieve the total consumed CPU time of AUT written in file /proc/<PID>/stat(11). The consumed CPU time is measured in clock ticks, which is 100Hz in current Linux system. The consumed CPU time for each operation is calculated and feed back to SPAG when agent receives next command. In section 4.2 we use the pseudo code to describe rest implement of smart wait.

5.4 Hybrid Verifying


Verifying by Pixel-based image comparing is slower than program-level comparing; it is also error-prone due to the inexactly triggered GUI operations, which makes result screen different from expected. Based mainly on Android's Accessibility Services supplemented by Sikuli's image search API, SPAG increases the speed of verifying and reduce the verifying error. In Figure 10, SPAG uses the accessibility service(12) to monitor the current activity's package name and class name(12->9->4->3). Any changes on these two names will trigger the device controller to generate a verification command into test script(3->5). The engineer may use Sikuli image API(7) to insert image verify commands such as exists, find, findAll (showed in Figure 9) if monitoring the activity's package name and class name is not enough for verifying the test objective.

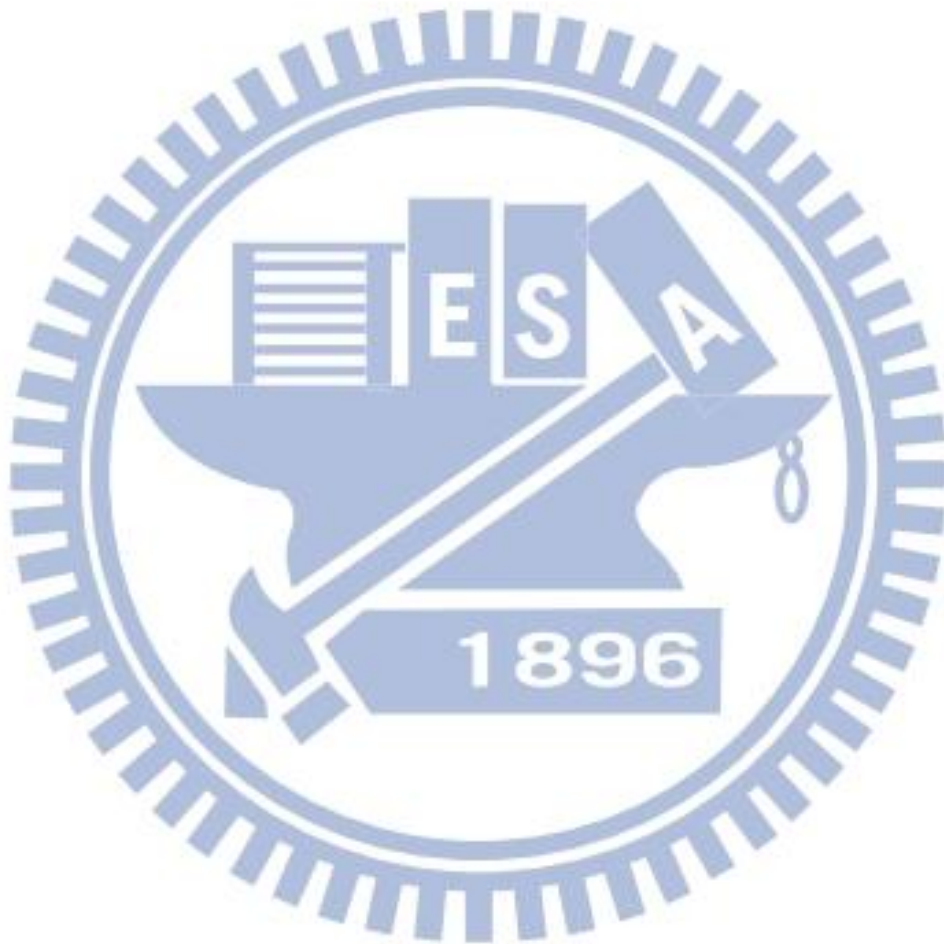
In the replay stage(Figure 12), when the script executor reads to a layout data verify command, it calls device controller to retrieve the activity's package name and class name(3,8->5) and compare the retrieved names with the those written in the command. For example, the layout data verify command

```
Device_0000003458763848.verify("com.android.contacts",  
    "com.android.contacts.DialtactsActivity")
```

is a verification command generated with hybrid verifying method. If the activity name and class are different in replay stage, the testing fails. In another hand, if the script executor



reads to an image verify command such as `if exists()`, it will call Sikuli's image matching API to compare the DUT screen with the image written in command (1,8->5). If the image is missed on DUT screen, the verification fails.



Chapter 6. Experiment Result

In this chapter, we first introduce the experiment environment. Next, Section 6.2 compares the test accuracy among GUI test tools. Finally, Section 6.3 adopts analysis of the proposed method.

6.1 Testbed

As we described in Chapter 5, the testbed includes a host PC with SPAG test IDE and an Acer Liquid smart phone with SAPG test agent. The test manager provides test case recording, editing, and replaying function. The test agent receives command and reproduces events on DUT. In addition, test agent may monitor DUT state and report the current state to test manager if we enable SPAG's Smart Wait. The SPAG testing flow includes four parts: input events recording; event sequence grouping and replaying; dynamically delaying command; and Hybrid Verifying.

In order to estimate the test accuracy of proposed methods, we design two experiments that use test accuracy to compare SPAG and another GUI testing tool and analyze the improvement of SPAG's event reproducing solutions. The experiments are running five test cases with fix different system workloads. We run 40 rounds test on every combination of these test cases and workloads. The tested workloads include CPU usage 0%, 25%, 50%, 75%, 100% and read/write flash memory. As mentioned in 4.2, embedded systems are highly influenced by heavy system workload. Therefore, this case study focuses on how the test accuracy changes under different system workloads. The test scenarios we choose are: browse a contact entry, install application over Wi-Fi, take a picture, Shoot a video, and browse Google map over Wi-Fi. We select the test scenario in accordance with two rules. First, the scenario must be a user behavior often happens in normal usage. Second, the scenario should contain CPU/IO intensive behavior, which is easily affected by system's CPU/IO workload, thus the test results are easier to distinguish.

We perform 40 round tests on each of all combination of 6 workloads and 5 test case and score each round by test result of each test round.

6.2 Comparison of test accuracy and time efficiency

The subject of first experiment is to compare the test accuracy of tools including our SPAG and monkeyrunner. The SPAG has implemented three solutions we proposed. The monkeyrunner can act like a record-replay test tool and it works in remote testing environment too. We use monkeyrunner to perform GUI test by using it's API to send one event at a time and uses fixed delay between each event. As a result, the behavior of monkeyrunner is similar to a GUI test tool without event reproducing solutions we proposed. Therefore, we consider monkeyrunner as a naïve GUI testing tool and compare the test accuracy between SPAG and monkeyrunner.

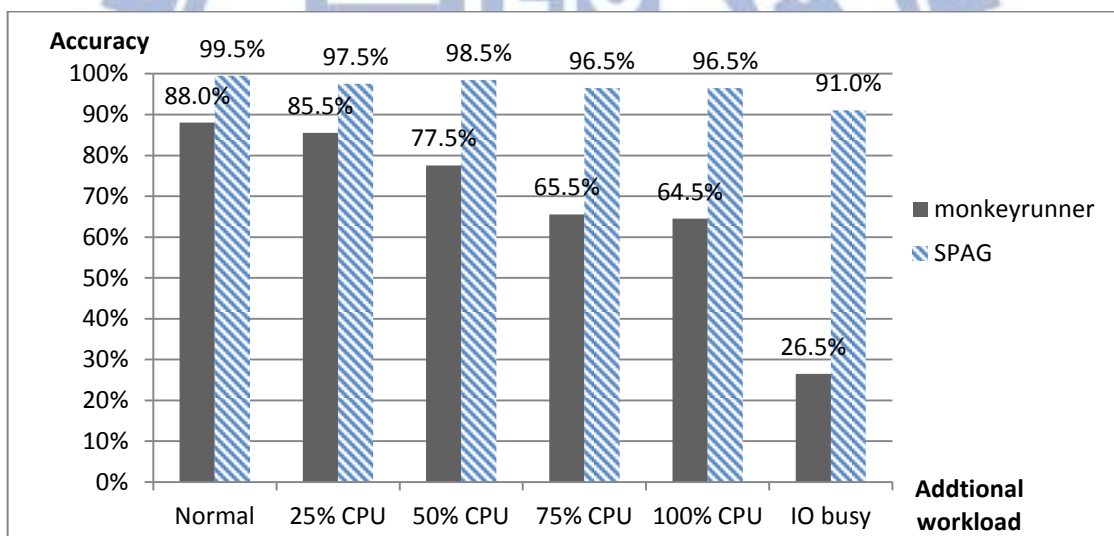


Figure 13. Testing with SPAG and monkey runner

Figure 11 shows the test result of test accuracy, in which the x-axis is the different workload types and the y-axis is the test accuracy. For example, in the case of 100% CPU workload in Figure 11(a), the test accuracy can archive 93% and 55% when the when the testing is executed by SPAG and monkeyrunner. As Figure 11 shows, the accuracy becomes lower when the CPU workload gets heavier. This is because the less CPU resource to AUT is assigned, the more frequent delay of AUT happens. In a result, testing failed when application

level event is triggered with different properties or next command executes when current command is not finished yet.

Compare to monkeyrunner, our solutions deliver the best test accuracy for all workloads. For instance, in the case of install Apps, a script running on system repeats installing and removing three apps in parallel. As result, the IO workload costs most IO resource part of DUT, which makes AUT encounter more IO waiting, decrease CPU utilization, and finally cause AUT delays for seconds. However, monkeyrunner cannot bear such delay and fail on response timeout error.

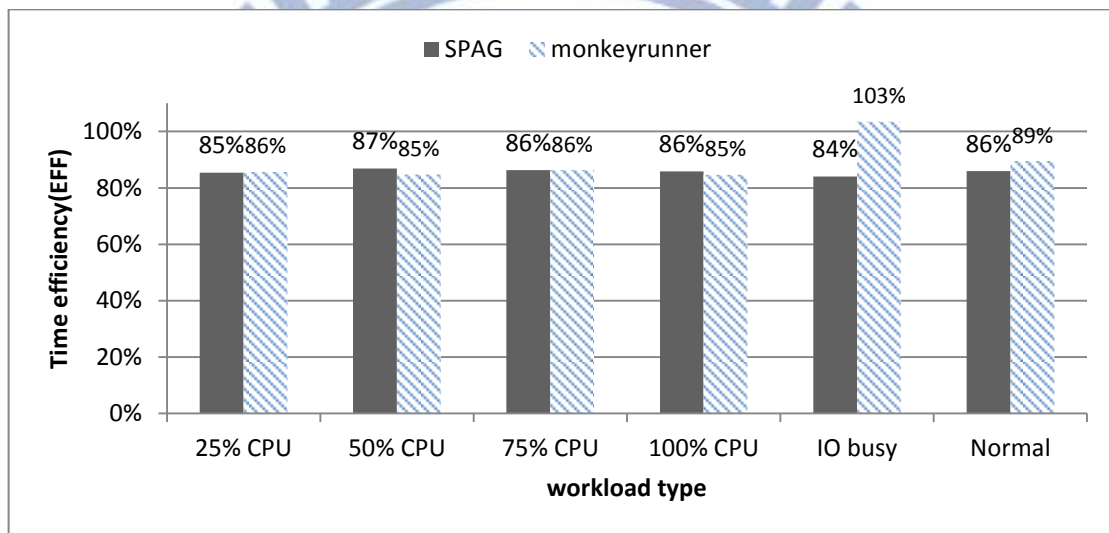


Figure 14. the time efficiency of GUI testing

Figure 12 shows the time efficiency of SPAG and monkeyrunner, in which the x-axis is the different workload types and the y-axis is the time efficiency. For example, when testing under IO busy workload, the SPAG has 84% of time efficiency while monkeyrunner has 103% time efficiency. The test result is the rate of expect testing time against actual testing time, as the EFF defined in chapter 2. Figure 12 shows the two GUI testing tools have similar time efficiency. The reason is that the monkeyrunner retries more times but costs less time for each testing due to its lower test accuracy.

6.3 SPAG Solution analysis

We have compared the accuracy of out proposed tool. Now we are going to test proposed solutions separately in order to understand what contribution each solution provides on

automated GUI testing.

Batch Event and Smart Wait

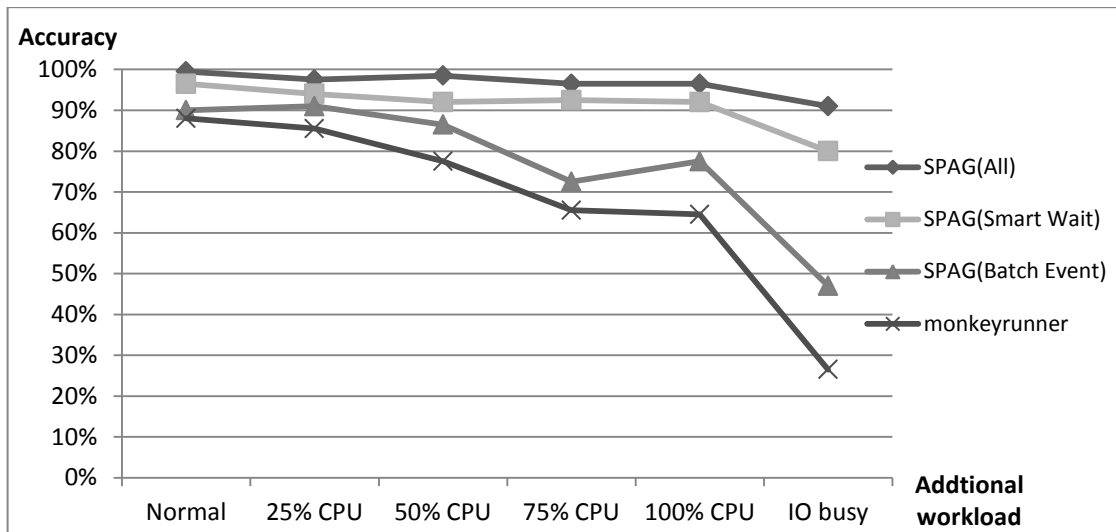


Figure 15. Testing with Batch Event and Smart Wait

Figure 13 shows the test result of Batch Event and Smart Wait methods, in which the x-axis is different workloads and the y-axis is the test accuracy. For example, in the case of 100% CPU workload, SPAG archives 77.5% and 92% accuracy by testing with Batch Event method and Smart Wait method separately. We use the test result of monkeyrunner as naïve test result in order to compare how much accuracy each method improves. In Figure 13, the SPAG with Smart Wait method is better than SPAG(Batch Event) because the Smart Wait can apply on all commands while the Batch Event only improves the accuracy on gesture input. However, the Batch Event method has a better performance if the test case focuses on gesture testing. There are no suitable tools for gesture testing currently.

Hybrid Verifying

The hybrid verifying method we proposed is used to automate the testing process. It monitors DUT's state and dynamically generates commands for verifying whether the package name and class name of current activity in testing are the same as recorded names. This method can reduce the cost on test writing case because it automatically adds verification commands for activity-level GUI layout changes on DUT. Engineer now can record a test case that moves between activities without writing codes to check whether the result activity changes especially.

In order to analyze how much cost spent on test case writing, we perform an experiment to test how much time costs on writing test cases with and without hybrid verifying. Figure 16 shows the experiment result with/without hybrid verifying, in which the x-axis is different test cases mentioned in section 6.1 and the y-axis is the time spent on writing those test cases. The result shows we can reduce small amount of writing time for contact UI testing, browser testing and installation testing. However, the writing time for recording video and taking picture can be reduced 63%~77% by applying the hybrid verifying method. The reason is that the recording video and taking picture are using customized UI and cause a class name change when press the shutter button, which makes the test case writing without hybrid verifying spent more time to check the shutter button has been pressed after a press operation, while the

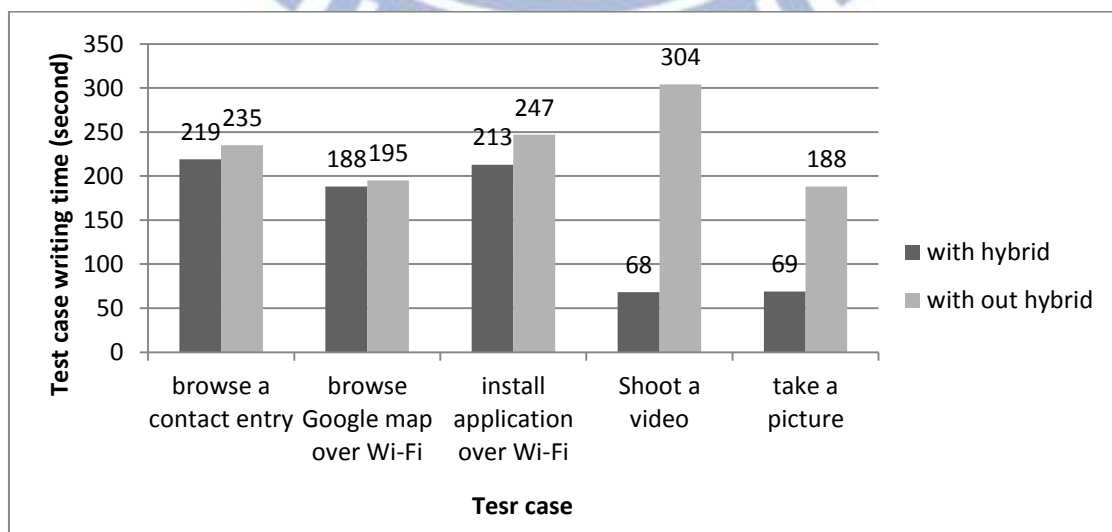
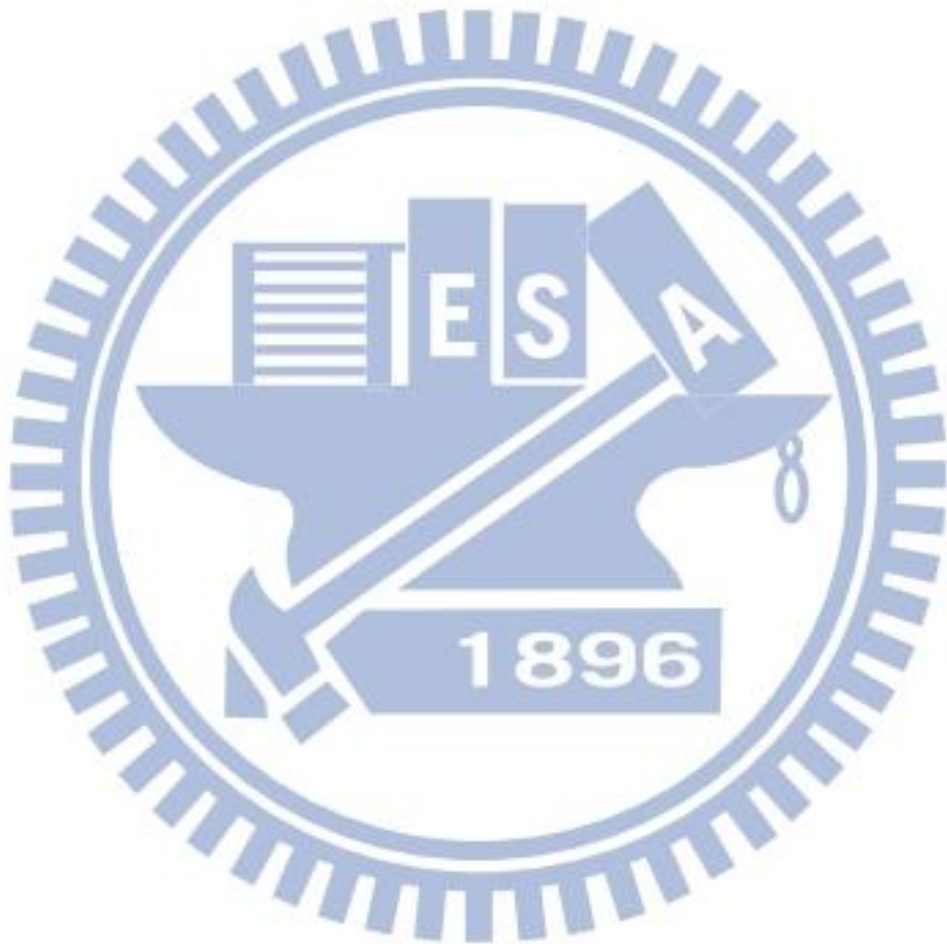


Figure 16. Test case writing time with/without hybrid verifying

verify command is automatically generated with hybrid verifying method. Since the smart wait reduces most test cases writing time, the time left for hybrid verifying to improve is very limited, but the improvement still exists based on our experiment.



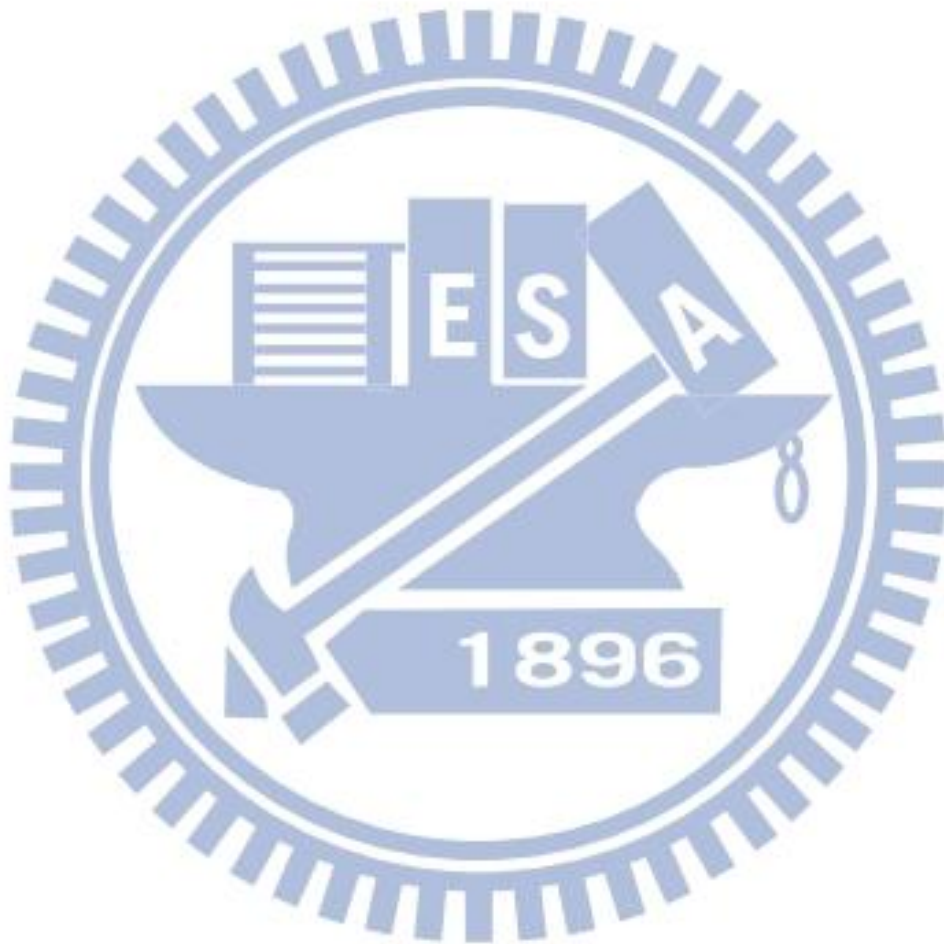
Chapter 7. Conclusions and Future Work

In this work, we design three solutions and implement a GUI testing tool based on Sikuli for improving the test accuracy and simplifying the testing process on embedded systems. The solution is based on manipulating the events reproducing and adopting GUI layout information provided by DUT. Our first contribution is to improve the accuracy of reproducing GUI operations and make the test process more robust against to application delay. We propose the Batch Event to reproducing the action consisting of several GUI events accurately by transferring the events in a batch. We also propose the Smart Wait method to extend the delay between operations dynamically by considering the CPU usage of target application during processing each operation. Unlike previewers tools, which consider only touch events inherited from conventional software testing, our SPAG is suitable for testing complex gesture inputs and testing with heavy system loading. The second contribution is to simplify the test case writing process. Based on the accessibility technology, SPAG can access the GUI layout information and automatically generating state verification commands to reduce the cost on manually writing verification commands. More state verifications also can reduce the chance to use pixel-based verifications, which is more error-prone, slower and resource-intensive.

In our experiment, we present the comparison on test accuracy and time efficiency under different workloads. Test cases are more likely to succeed by using SPAG. As result, SPAG can archive the test accuracy of 90% while the test accuracy of monkeyrunner is 27%~88%. The SPAG and monkeyrunner have similar time efficiency, it is due to the earlier failure caused by monkeyrunner's lower test accuracy.

In the future, we plan to adopt the new function of accessibility API in Android version 4.0 or higher in order to access detailed information on DUT. Moreover, we may adopt accessibility API to make the Smart Wait method can optionally check GUI layout

information in order to become more event-driven and increase the test accuracy. Since our methods assumes there is single pointer operating on DUT, if the input point are more than one, known as multi-touch gesture, our method might be invalid. Therefor another feature work is to support the recording and replaying on multi-touch gesture for embedded systems.



References

- [1] (2012). *Android SDK Tools: monkeyrunner*. Available: http://developer.android.com/guide/developing/tools/monkeyrunner_concepts.html
- [2] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: using GUI screenshots for search and automation," presented at the Proceedings of the 22nd annual ACM symposium on User interface software and technology, Victoria, BC, Canada, 2009.
- [3] T.-H. Chang, T. Yeh, and R. C. Miller, "GUI testing using computer vision," presented at the Proceedings of the 28th international conference on Human factors in computing systems, Atlanta, Georgia, USA, 2010.
- [4] T.-H. Chang, T. Yeh, and R. Miller, "Associating the visual representation of user interfaces with their internal structures and metadata," presented at the Proceedings of the 24th annual ACM symposium on User interface software and technology, Santa Barbara, California, USA, 2011.
- [5] Q. Xie and A. M. Memon, "Using a pilot study to derive a GUI model for automated testing," *ACM Trans. Softw. Eng. Methodol.*, vol. 18, pp. 1-35, 2008.
- [6] T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," in *Software Testing, Verification and Verification (ICST), 2011 IEEE Fourth International Conference on Software Testing, Verification and Verification*, 2011, pp. 377-386.
- [7] L. Zhifang, L. Bin, and G. Xiaopeng, "Test automation on mobile device," presented at the Proceedings of the 5th Workshop on Automation of Software Test, Cape Town, South Africa, 2010.
- [8] O.-H. Kwon and S.-M. Hwang, "Mobile GUI Testing Tool based on Image Flow," presented at the Proceedings of the Seventh IEEE/ACIS International Conference on Computer and Information Science (icis 2008), 2008.
- [9] J. Bo, L. Xiang, and G. Xiaopeng, "MobileTest: A Tool Supporting Automatic Black Box Test for Software on Smart Mobile Devices," presented at the Proceedings of the Second International Workshop on Automation of Software Test, 2007.
- [10] L. Zhi-fang and G. Xiao-peng, "SOA Based Mobile Device Test," presented at the Proceedings of the 2009 Second International Conference on Intelligent Computation Technology and Automation - Volume 04, 2009.
- [11] V. R. Vemuri, "Testing Predictive Software in Mobile Devices," presented at the Proceedings of the 2008 International Conference on Software Testing, Verification, and Verification, 2008.
- [12] *Section 508 of the Rehabilitation Act*. Available: www.access-board.gov/508.htm
- [13] M. Grechanik, Q. Xie, and C. Fu, "Maintaining and evolving GUI-directed test scripts," presented at the Proceedings of the 31st International Conference on Software Engineering,

2009.

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

[15] *GestureDetector.SimpleOnGestureListener* / *Android Developers*. Available: <http://developer.android.com/reference/android/view/GestureDetector.SimpleOnGestureListener.html>

