Improving Accuracy of Automated GUI Testing for Embedded Systems

Ying-Dar Lin, National Chiao Tung University

Edward T.-H. Chu, National Yunlin University of Science and Technology

Shang-Che Yu, National Chiao Tung University

Yuan-Cheng Lai, National Taiwan University of Science and Technology

// The Smart Phone Automated GUI (SPAG) batches and reproduces event sequences on the device under test to ensure that they are performed on time. //

AUTOMATED GUI TESTING for smart phones faces two major challenges: nondeterministic events and execution interference. Owing to uncertainty in the runtime execution environment, such as timing delay variations in communication, the device under test (DUT) might not reproduce interpreted events on time. As a result, actual intervals between events can differ from the predefined intervals given in the test script. Nondeterministic event sequences can easily lead to incorrect GUI operations. For example, the Android fling action occurs when a user scrolls a touch panel and then quickly lifts his or her finger. The device uses a sequence of motion events to represent the operation. When an automated GUI tool replays these event sequences, each motion event should be triggered on time to reproduce the fling with the same scrolling speed. If not, the scrolling speed of the reproduced fling action will lead to an incorrect result. To address the issue of nondeterministic events, a commonly used method is to use a trackball instead of the fling action. However, not all smart phones are equipped with trackballs.

An uncertain runtime execution environment can interfere with or delay an application's execution, especially when the DUT is under a heavy load. A delayed application can fail to process an event correctly if the response to the previous event hasn't been completed. For example, an event might be dropped if the application under test (AUT) receives the event ahead of time and isn't ready to process it. To solve this problem, an intuitive method is to delay the execution of the operations. However, this requires experienced engineers to set the delay for each operation properly so that the application can receive the reproduced events.

We aimed to design an automated GUI testing system to maximize accuracy within the uncertainty of runtime execution environments. The accuracy of an automated GUI

 0740-7459/14/$31.00 © 2014 IEEE

testing tool is defined as the success rate of examining a bug-free application. The higher the success rate, the higher the accuracy. Thus, we designed the Smart Phone Automated GUI (SPAG) testing tool, based on Sikuli, a popular open source automated GUI tool.[2,3] Using the Sikuli integrated development environment, we can write GUI test cases, execute the script, automate GUI operations on a desktop, and verify GUI elements presented on a screenshot. To avoid nondeterministic events, we batched the event sequence and reproduced the events on the DUT.

In addition, SPAG can monitor the target application's CPU usage during runtime and dynamically change the timing of following operations so that all event sequences and verifications can be performed on time, even when the DUT is heavily loaded. We conducted several experiments on an Acer Liquid smart phone to investigate the applicability and performance of SPAG and compared our method with MonkeyRunner (http://developer.android.com/tools/help/monkeyrunner_concepts.html). For related work on GUI testing, please see the sidebar.

Overview

We adopted a commonly used software testing technique called record/replay for embedded systems. Figure 1a shows the recording stage, where the screen of the DUT is first redirected to the host PC, which runs the test tool. An engineer interacts with the DUT remotely: whenever the engineer performs a GUI operation on the host PC, such as a key press or a finger touch, the test tool sends events associated with the GUI operation to the DUT and records them

RELATED WORK ON GUI TESTING

Researchers have dedicated much work to automated GUI testing. The most common approach is model-based testing (MBT), which models target applications' behaviors and uses the test cases the models generate to validate the device under test. Tommi Takala and his colleagues adopted MonkeyRunner and Window services to generate GUI events,[1] and Zhifang Lin and his colleagues utilized the concept of virtual devices to test applications.[2] These methods rely on image-based pattern matching, which is sensitive to images' quality. The Smart Phone Automated GUI (SPAG) testing tool uses GUI components for pattern matching to improve the stability and the speed of validation.

Several techniques and architectures were developed to cope with complex application tests. MoGuT, a variant of the finite-state machine (FSM) based test framework, uses image flow to describe event changes and screen response.[3] However, it lacks flexibility. Grey-box testing adopted APIs to construct calling contexts and parameters from input files.[4] Based on a logging mechanism, grey-box testing verifies testing results. However, for complex software, it becomes difficult to describe the testing logic and calling context. Recently, Cuixiong Hu and his colleagues developed an approach to automate the testing process of Android applications using JUnit and MonkeyRunner tool.[5] Wei Yang and his colleagues proposed a method to automatically extract a model of an application.[6] However, both of the methods used a fixed delay between consecutive GUI operations, whereas SPAG determines the delay dynamically by using the Smart Wait function. Domenico Amalfitano and his colleagues designed a method to automatically generate a model of application by using dynamic crawling.[7] However, their method required the source codes of the applications under test. SPAG doesn't require the source code.

References

1. T. Takala, M. Katara, and J. Harty, "Experiences of System-Level Model-Based GUI Testing of an Android Application," *Proc. Int'l Conf. Software Testing, Verification and Verification* (ICST 11), IEEE CS, 2011, pp. 377–386.
2. L. Zhifang, L. Bin, and G. Xiaopeng, "Test Automation on Mobile Device," *Proc. 5th Workshop Automation of Software Test* (AST 10), ACM, 2010, pp. 1–7.
3. O.-H. Kwon and S.-M. Hwang, "Mobile GUI Testing Tool Based on Image Flow," *Proc. 7th IEEE/ACIS Int'l Conf. Computer and Information Science* (ICIS 08), IEEE CS, 2008, pp. 508–512.
4. V.R. Vemuri, "Testing Predictive Software in Mobile Devices," *Proc. Int'l Conf. Software Testing, Verification, and Verification* (ICST 08), IEEE CS, 2008, pp. 440–447.
5. C. Hu and I. Neamtiu, "Automating GUI Testing for Android Applications," *Proc. 6th Int'l Workshop on Automation of Software Test* (AST 11), ACM, 2011, pp. 77–83.
6. W. Yang, M.R. Prasad, and T. Xie, "A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications," *Proc. 16th Int'l Conf. Fundamental Approaches to Software Engineering* (FASE 13), Springer, 2013, pp. 250–265.
7. D. Amalfitano et. al., "Using GUI Ripping for Automated Testing of Android Applications," *Proc. 27th IEEE/ACM Int'l Conf. Automated Software Engineering* (ASE 12), ACM, 2012, pp. 77–83.

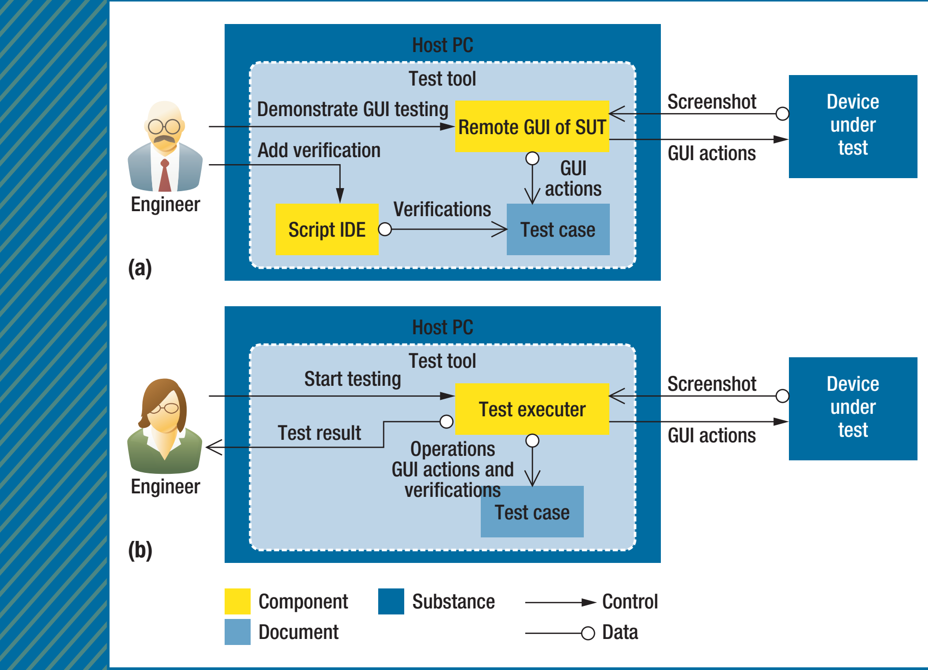


FIGURE 1. The system architecture of the record/replay method and the device under test: (a) the recording stage and (b) the replay stage.

in a test case. The test case also includes verification operations, added by the engineer, to verify the DUT's response. Figure 1b shows the replay stage, where the test executer first reads GUI operations from the test case and replays them on the DUT. Finally, the test executer verifies the testing results according to the DUT's response.

C denotes a test case that includes n operations $\{O_1, O_2, \ldots, O_n\}$. An operation can be a GUI operation or a verification operation: a GUI operation can be a key press or a finger touch, and a verification operation is used to verify the test result. The interval between O_{i-1} and O_i is given by T_i. A GUI operation consists of a sequence of events $\{e_{i,1}, e_{i,2}, \ldots, e_{i,m}\}$. For example, when a user performs a fling operation, the Android system generates the associated move events.

Owing to the uncertainty of runtime execution environments and variations in the communication delay between the host PC and the DUT, the DUT might not reproduce each event $e_{i,j}$ on time. Such nondeterministic event sequences can lead to an incorrect GUI operation and invalidate verification operations. Furthermore, the runtime execution environment of the DUT might also affect the interval T_i between O_{i-1} and O_i. The GUI application might drop the new arrival events of O_i because the previous events of O_{i-1} haven't been processed yet. Such dropped events will also lead to test failures.

SPAG Design

We designed SPAG to accurately reproduce GUI operations and verify test results. In the record stage, SPAG monitors GUI operations and stores these GUI operations and associated CPU times of the DUT in a test script. An engineer also adds verification operations to the test script to verify the results. In the replay stage, GUI and verification operations are batched and sent to the DUT so that the events can be triggered on time. Based on the CPU utilization of the DUT, SPAG dynamically modifies the duration of two operations. The testing results are sent back to the host PC for verification.

Event Batch

In the replay stage, the application running on DUT continues monitoring the GUI events and takes corresponding operations. For example, a gesture, such as a swipe operation, includes several multitouch events. After receiving the multitouch events, the application scrolls the screen up. However, some GUI operations are sensitive to the timing of associated events. For example, the onFling GUI operation consists of many move events. The speed of onFling is sensitive to both displacement and time difference between two continuous move events. If the actual interval between two move events is longer than the interval described in the test script, the speed of the reproduced onFling GUI will be slower than expected, and the incorrect GUI operation could lead to test failure. Therefore, in the replay stage, it's crucial to trigger each event at the DUT on time to avoid possible test failures.

In our implementation, SPAG stored the associated events of each GUI operation and event intervals in the test script. In addition, a tag, such as ACTION_DOWN, ACTION_MOVE, or ACTION_UP, was attached at the end of each GUI operation to differentiate continuous GUI operations. In the replay stage, SPAG first batched

all events and sent them to the DUT. Next, a module at the DUT rather than a module at the host PC triggered the events to remove the effect of commutation uncertainty between the DUT and host PC.

Smart Wait

In the replay stage, the recorded GUI operations are sent to the associated application accordingly. However, the execution time of the application can be longer than expected if the execution environment is heavily loaded, and the prolonged application might have failed to process a GUI operation correctly if the operation came earlier than expected. For example, if the DUT received the push-bottom operation ahead of time and the AUT wasn't ready to process the GUI operation, it would be dropped and lead to test failure. A practical method to avoid execution interference was to ask experienced engineers to set the duration of each pair of GUI operations so that the application could process GUI operations on time while maintaining a reasonable testing time. But, the cost of manually adjusting durations is high.

To improve the efficiency of the test process, SPAG automatically adjusted delay time between two GUI operations based on CPU time used to perform GUI operations. The function is called Smart Wait. In this function, p denotes the process that performs the GUI operations. In the record stage, when operation O_{i-1} took place, SPAG monitored the CPU time cpu_i of process p at duration T_i between O_{i-1} and O_i. This was achieved by parsing data from the Linux OS virtual directory /proc. From /proc/<PID>/stat, we obtained the time the process had spent in both the user space and kernel space. In addition, we obtained from /proc/stat the time the CPU had spent in both the user and kernel space. Based on this information, SPAG calculated the CPU usage cpu_i of the process p at duration T_i. Both cpu_i and T_i were stored in the test script as CMD(T_i, cpu_i). Note that p' denotes the process that performed the GUI operations in the replay stage. When O_{i-1} was executed, SPAG monitored the CPU time cpu_i' of p'. If cpu_i' was smaller than cpu_i, SPAG assumed that O_{i-1} was incomplete and calculated a proportional delay time for remaining GUI operations. For example, in the recording stage, if O_{i-1} used 5 milliseconds of CPU time out of 4 seconds for execution, then cpu_i was 5 milliseconds and T_i was 4 seconds. SPAG inserted a command CMD(4000 ms, 5 ms) in the test script right after O_{i-1}. In the replay stage, when O_{i-1} was replayed, SPAG first waited 4 seconds and read the associated cpu_i' from the DUT. If cpu_i' was 2 ms, SPAG assumed that O_{i-1} was unfinished and estimated its completion time as 4 s × 5 ms/2 ms = 10 s. In this case, the next operation O_i was postponed by 6 seconds.

Implementation

SPAG integrated two popular open source tools: Android screencast and Sikuli. Android screencast is a desktop application that redirects the screen of the DUT to the host PC and allows an engineer to interact remotely with the DUT by using a mouse or keyboard. Sikuli is a desktop application that automatically tests GUIs via screenshot images. In the recording stage on the host PC, SPAG records all GUI operations performed inside the redirected screen of the DUT. An engineer uses Sikuli's IDE to insert a verification operation at the end of one or several continued GUI operations by selecting a region of the redirected screen. The class name and activity name of the redirected screen are also logged at that time. In the replay stage, SPAG reproduces GUI operations by sending associated events to the DUT.

We adopted both Smart Wait and Event Batch to reduce the uncertainty of the runtime execution environment. Event Batch aims to remove the communication uncertainty between the DUT and PC, whereas Smart Wait aims to remove the uncertainty of the DUT runtime execution environment. They can be applied together or separately depending on the communication uncertainty and runtime execution environment. When performing a verification operation, SPAG first checks the class name and activity name of the redirected screen. If the check fails, SPAG instantly makes an image comparison between the redirected screen and the predefined image. Note that the methodologies of Event Batch and Smart Wait are portable. To take advantage of these two techniques to perform

Pull Quote

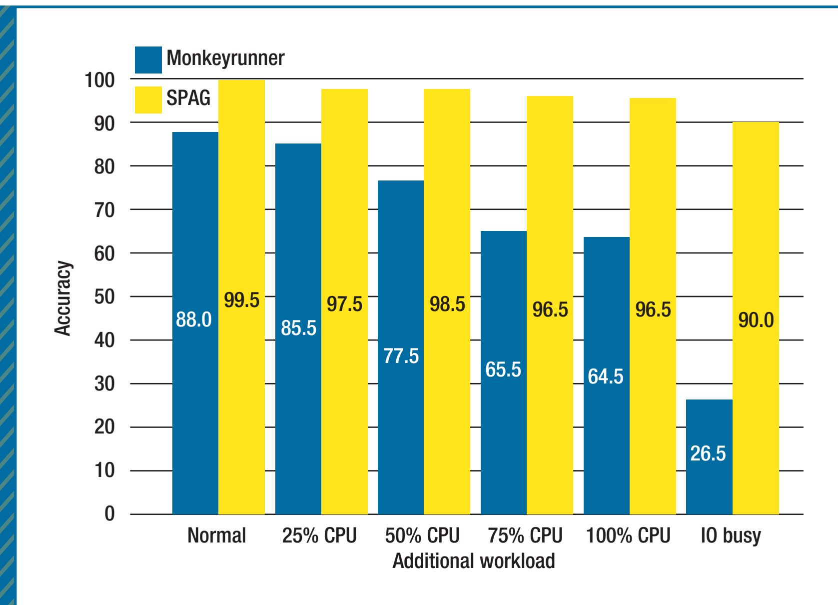


FIGURE 2. Testing with the Smart Phone Automated GUI (SPAG) and MonkeyRunner. The accuracy of MonkeyRunner dropped significantly when the CPU utilization increased or the I/O subsystem was busy. The accuracy of SPAG was over 90 percent in all configurations we tested.

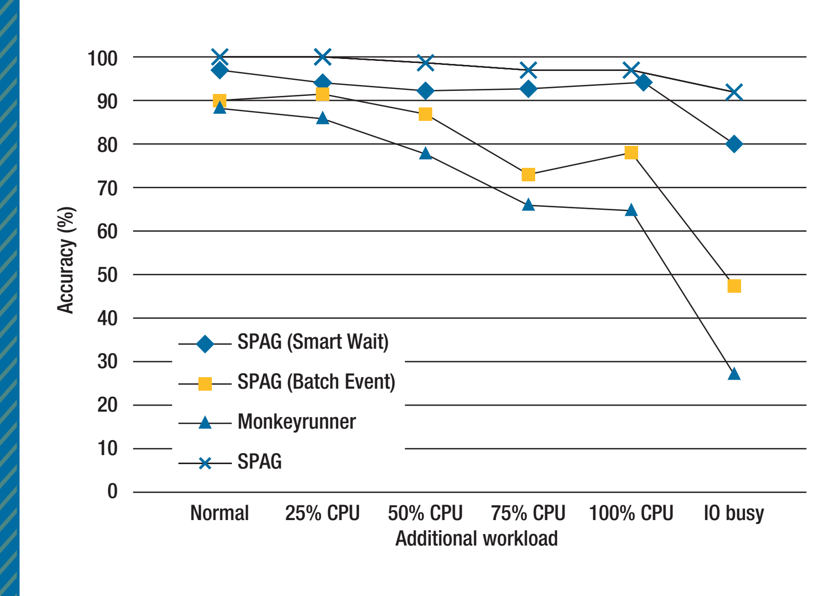


FIGURE 3. Testing with Event Batch and Smart Wait. Event Batch and Smart Wait can be applied together or separately depending on the communication uncertainty and runtime execution environment. The Smart Wait function contributed more than the Event Batch function in improving accuracy if the system is busy.

GUI testing on other platforms, you would need to use an equivalent of Android screencast to remotely control the DUT and integrate that tool with Sikuli or an equivalent tool to record user interaction.

Experimental Results

In this section, we first explain the experiment setup. Then, we evaluate the accuracy of SPAG and MonkeyRunner. Finally, we investigate the effects of smart wait and event batch on accuracy.

Experiment Setup

To investigate the accuracy of SPAG, we adopted the Acer Liquid smart phone for evaluation. We compared SPAG with MonkeyRunner, an automated testing tool included in the Android software developer's kit. MonkeyRunner reproduces predefined operations, such as key presses, by generating associated events and sending the events from the host PC to the DUT.[1] Our test script included five commonly used scenarios: browsing a contact entry, installing an application over Wi-Fi, taking a picture, making a video, and browsing Google maps over Wi-Fi. Figures 2 and 3 show how we used a busy-loop program to adjust the CPU utilization from 25 to 100 percent and adopted an intensive flash read/write program to simulate input/output burst condition. For each configuration, CPU utilization is 25 , 50, 75, or 100 percent. We repeated the same experiment 40 times and took the average value of accuracy for comparison.

Test Accuracy

The experiment evaluated the accuracy of SPAG and MonkeyRunner. We checked the accuracy of MonkeyRunner manually because

TABLE 1 Accuracy of SPAG and MonkeyRunner by percentage.

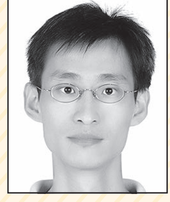
Workload	Skype		Twitter		Facebook	
	SPAG	MonkeyRunner	SPAG	MonkeyRunner	SPAG	MonkeyRunner
Normal	97.5	92.5	99.5	92.5	97.5	72.5
25% CPU	97.5	99.5	99.5	92.5	97.5	65.0
50% CPU	99.5	99.5	99.5	72.5	97.5	60.0
75% CPU	99.5	99.5	99.5	40.0	92.5	60.0
100% CPU	99.5	99.5	99.5	37.5	92.5	40.0
I/O Busy	99.5	72.5	95.0	20.0	92.5	40.0

it didn't support a sufficient image comparison function to verify testing results. MonkeyRunner's accuracy dropped significantly when the CPU utilization increased or the I/O subsystem was busy. For example, MonkeyRunner's accuracy dropped to 64.5 percent when CPU utilization was 100 percent and to 26.5 percent when an I/O burst occurred. This was because the tested application was deferred for execution when the system was heavily loaded. MonkeyRunner doesn't dynamically modify the duration of two continuous operations. As a result, the new communing events were dropped or ignored, which made MonkeyRunner tests fail. On the contrary, with the Smart Wait function, the accuracy of SPAG decreased only slightly when CPU utilization increased or I/O bursts occurred; its accuracy was over 90 percent in all the configurations we tested. Under normal conditions in which CPU utilization was less than 25 percent, the accuracy stayed at 99.5 percent.

With the same experimental setup, we also adopted three popular mobile apps, Skype, Twitter, and Facebook, to evaluate the accuracy of SPAG and MonkeyRunner. The

ABOUT THE AUTHORS

YING-DAR LIN is a professor in the Department of Computer Science at National Chiao Tung University. His research interests include embedded systems, network protocols, and algorithms. He received a PhD in computer science from UCLA. He is an IEEE Fellow. Contact him at ydlin@cs.nctu.edu.tw.



EDWARD T.-H. CHU is an assistant professor in the Department of Computer Science and Information Engineering at National Yunlin University of Science and Technology. His research interests include embedded system software. He received a PhD in computer science from National Tsing Hua University. Contact him at edwardchu@yuntech.edu.tw.

SHANG-CHE YU is a software engineer with Hope Bay Technologies, Taiwan. He received an MS in computer science from National Chiao Tung University. Contact him at comet.jc@gmail.com.

YUAN-CHENG LAI is a professor in the Department of Information Management at National Taiwan University of Science and Technology. His research interests include performance analysis and wireless networks. He received a PhD in computer science from National Chiao Tung University. Contact him at laiyc@cs.ntust.edu.tw.

major gesture activity of Skype was tapping, whereas that of Twitter and Facebook was flinging. Table 1 shows that the SPAG maintained a very high level of accuracy in all configurations, whereas MonkeyRunner performed poorly when the system was busy, especially for Twitter and Facebook. This is because MonkeyRunner can't trigger events on time that are associated with flinging.

Effects of Smart Wait and Event Batch on Accuracy

Figure 3 shows how in the case of a 100 percent CPU workload, the accuracy of SPAG was 77.5 percent with the Event Batch function and 92 percent with the Smart Wait function. Smart Wait contributed more than Event Batch in improving accuracy when the system was busy. This is because Smart Wait can be applied to all GUI operations, whereas Event Batch can only improve the accuracy of moving GUI operations, such as scrolling and flicking.

We designed SPAG to avoid nondeterministic events by batching the event sequence and reproducing them on the DUT directly. In addition, SPAG monitors target applications' CPU usage at runtime and dynamically change the timing of the next operation so that all event sequences and verifications can be performed on time, even though the DUT is heavily loaded. Our experiments showed that SPAG can maintain a high accuracy of up to 99.5 percent. According to our current design, as long as a smart phone is supported by Android screencast, we can test it with SPAG without needing to modify anything. In the future, we plan to design a fully platform-independent automated GUI testing system.

Acknowledgements

This work was supported in part by the National Science Council and the Institute for Information Industry in Taiwan.

References

1. T. Yeh, T.-H. Chang, and R.C. Miller, "Sikuli: Using GUI Screenshots for Search and Automation," *Proc. 22nd Ann. ACM Symp. User Interface Software and Technology* (UIST 09), ACM, 2009, pp. 183–192.
2. T.-H. Chang, T. Yeh, and R.C. Miller, "GUI Testing Using Computer Vision," *Proc. 28th Int'l Conf. Human Factors in Computing Systems* (CHI 10), ACM, 2010, pp. 1535–1544.