Ilkka Kortelainen

Automated GUI Testing for Android Applications

Bachelor's Thesis
in Information Technology
December 12, 2012

University of Jyväskylä

Department of Mathematical Information Technology

Jyväskylä

Author: Ilkka Kortelainen
Contact information: ilkka.j.kortelainen@student.jyu.fi
Title: Automated GUI Testing for Android Applications
Työn nimi: Android-ohjelmien graafisten käyttöliittymien testauksen automatisaatio
Project: Bachelor's Thesis in Information Technology
Page count: 21
Abstract: Mobile devices are becoming more and more important in our society. As the number of mobile device users grows, so does the importance of proper application quality control and verification. Test automation is one of the key factors in increasing application quality. This paper is a literature review on current research on automated GUI testing of Android applications.
Suomenkielinen tiivistelmä: Mobiililaitteet ovat yhä tärkeämmässä asemassa yhteiskunnassamme. Kun mobiililaitteiden käyttäjien määrä kasvaa, samoin kasvaa tarve sovelluksien paremmalle laadunvarmistukselle. Testauksen automatisointi on yksi avaintekijöistä sovelluksien laadun parantamiseen. Tämä tutkimus on kirjallisuuskatsaus Android-ohjelmien graafisten käyttöliittymien testauksen automatisaation nykytutkimuksesta.
Keywords: Android, Test Automation, Testing Tools, GUI testing, GUI automation
Avainsanat: Android, Test Automation, Testing Tools, GUI testing, GUI automation

Contents

1 Introduction **1**

2 GUI testing methods **3**

2.1 A GUI Crawling-Based Technique for Android Mobile Application Testing . . . 3
2.2 Automated GUI Testing on the Android Platform . . . 5
2.3 Automating GUI testing for Android applications . . . 7
2.4 Experiences of System-Level Model-Based GUI Testing of an Android Application . . . 8
2.5 GUI testing using computer vision . . . 10

3 Analysis of the testing methods **11**

3.1 The GUI crawler . . . 11
3.2 Android Instrumentation Framework and Positron Framework . . . 12
3.3 JUnit and Monkey . . . 13
3.4 MBT with Monkey and TEMA tools . . . 14
3.5 Sikuli Test . . . 15

4 Conclusions **17**

References **17**

1 Introduction

Mobile devices are increasingly important in modern society. The gap between tasks performed with desktop computers and mobile devices is shrinking rapidly. As the devices get bigger screens, faster processors and the graphical user interfaces (GUIs) become more complex, new ways to ensure the quality of applications must be found.

The verification of mobile applications differs from traditional desktop applications in how the applications are constructed and the environment they are run in. In desktop environment the testers of non-GUI applications have several tools to automate their testing needs. For example, to test that a function call multiplyByTwo(4) works correctly, a test can be written where this function is called followed by an assertion function call such as assert(multiplyByTwo(4) == 8) to check the correctness of the answer and report an error if the answer differs from the expected one. The finished script can then be automated to run as many times and as often as needed, which reduces the tester's need for repetitive actions (Chang, TH., Yeh, T. & Miller RC., 2010). Also there is no chance of manual mistakes while executing the test.

In comparison, manual GUI testing is a time consuming and error-prone task. Let's consider testing the functionality of the "play" button in a video player: when pressed, the button should change from the "play" button to "pause" button. To verify the correct functionality, a tester must find a the "play" button on the screen, click it, and check that the "pause" button has replaced it. Every time this functionality is to be verified, a tester must repeat the same steps over and over again (Chang et al., 2010).

Testing mobile GUI applications has some special requirements compared to the desktop applications: The GUIs are event-driven and the user can click anywhere on the screen he chooses making the applications non-deterministic. To test every possible state the GUI is in is almost impossible. This makes mobile GUI testing harder compared to normal functional testing. Manual GUI testing is prone to errors and it is very hard to reproduce exactly the same conditions every time we want to run the same test. This translates to a high workload on the testers (Hu, C. & Neamtiu, I., 2011).

Automating the GUI testing in mobile devices is one of the next big challenges in the mobile industry. Without automation the quality of the software we are using will not be as good as it could, and, without any doubt, should be.

Google's Android platform is one of the most popular mobile device platforms

at the moment with a leading market share in smartphones in Q3 of 2011. There are many different methods and tools to automate Android GUI testing. Choosing the right method and tool for the job is critical for the successful verification of an application.

This paper presents a literature review of current research on automated GUI testing on Android devices and tries to answer the following question: for different kinds of testing, what is the most practical automatic GUI testing approach for testing Android applications. It is important to differentiate between different testing needs. Regression testing may need a different tool set than e.g. performance testing and it is improbable that one single method is best for everything.

The following should be considered important factors in every good GUI test automation tool (Chang et al., 2010):

- The testers should be able to write their own automation scripts;
- The scripts should be easy to maintain: the less the scripts have to be modified because the application changes, the better;
- The testing tool should minimize the effort of writing the test scripts;

There are also some secondary factors, such the possible cost of the tool, the effect of frequency of application updates and the diversity of different devices that run Android that must be taken into account when comparing the different testing methods, if possible.

One interesting point to consider is: can we depend on that calling low-level APIs to send keystrokes and verifying the result is really analogous to what the end-user does and sees. Is there any possibility of missing some of the errors when we depend on the information we get from the low-level APIs compared to human manual testing.

The research was conducted by making a literature review on the subject. This method was chosen because it is an appropriate way to compare all the different research on the subject to find out if some method is better than others in relation to the research question. It would also be impossible for me to gain access to all the hardware and software needed to do a comprehensive research on the subject myself.

Because the field of this research is comparatively young, there are many different kinds of answers to GUI testing automation problem. By conducting a literature review I got a fairly good insight in the current solutions and the direction the research is heading.

The rest of the thesis is structured as follows. Section 2 discusses the selection method of the articles that were chosen for the literature review and describes the content of each paper. Section 3 analyses the different methods and tools used in the selected papers. Section 4 highlights a couple of interesting methods and concludes the paper.

2 GUI testing methods

The following articles were selected to reflect the current state of automated Android GUI testing. The relevance of the results was evaluated by reading the abstracts. It must be noted that there were more potentially relevant results than the five that were taken into this study, as the scope of this review is limited. As such, this cannot be considered a systematic literature review (SLR) as discussed by Kitchenham et al. (2010). I did not assess the quality of the included studies with any formal metrics.

All but one of the selected texts are fully about Android GUI testing automation. The exception to this is the Chang et al. (2010) article on GUI testing using computer vision. I included this paper because it brings a unique point of view to the discussion: all the other methods described are mostly based on low-level APIs to check that the device's output is what it is supposed to be. The computer vision method uses screenshots of the display which are then compared to the expected output. In the end-user testing scenario this resembles more closely to what the real end-user would see compared to checking some UI components value from the APIs, but it also has some inherent technical difficulties.

Next, I will briefly go through the selected papers and describe their content.

2.1 A GUI Crawling-Based Technique for Android Mobile Application Testing

In their paper, Amalfitano, D., Fasolino, AR. & Tramontana, P. (2011) describe the open issues and problems in Android application testing and propose adopting a GUI crawling based testing technique used in traditional GUI testing. For this GUI crawling testing, Amalfitano et al. (2011) created a tool called A^2T^2 (Android Automatic Testing Tool), a Java application which consists of three main components:

- Java code instrumentation component;
- GUI Crawler;
- Test Case Generator.

The Java code instrumentation component is used to instrument the Java code running on the virtual machine (VM). The component instruments the code automatically and enables the detection of run-time crashes.

The GUI crawler component is used to automatically generate a GUI tree by deducing the possible event sequences in the GUI. The GUI tree's nodes represent user interfaces and the edges are event-based transitions between the interfaces (see Figure 1). The GUI crawler component produces a repository which details the GUI tree, describes the found interfaces and triggered event. The repository is also used to produce reports of the crashes which happened during testing and describes which event sequences lead to the crashes.

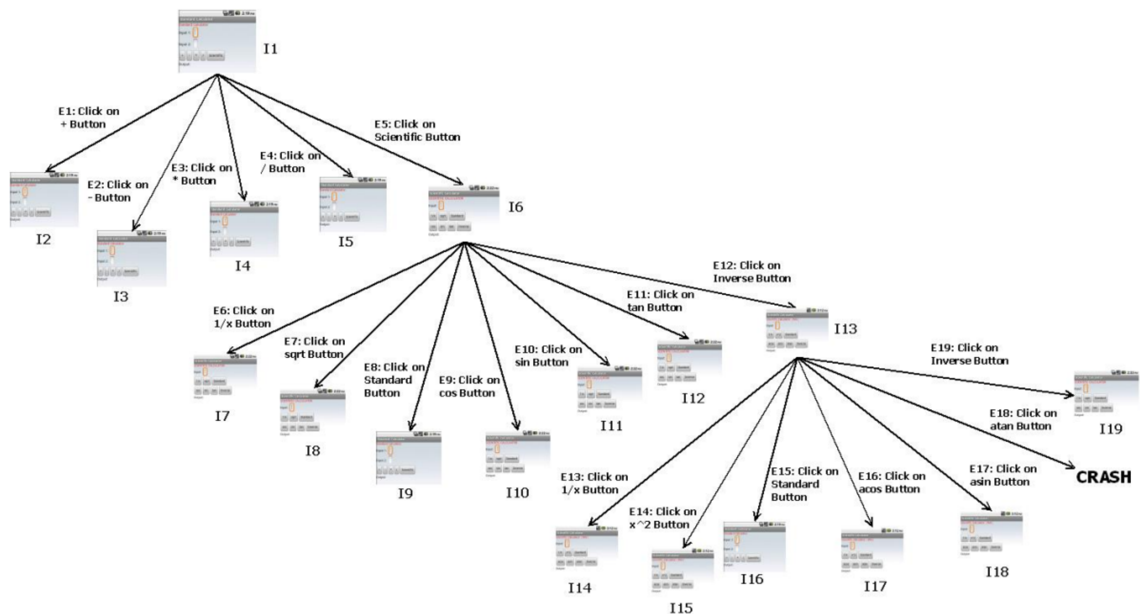

Figure 1: The GUI Tree obtained by crawling the example Android application (Amalfitano et al., 2011)

The Test Case Generator component is used to automatically generate test cases from the GUI tree produced by the GUI crawler component. The test cases themselves are Java test methods, which have the ability to replay different event sequences in the GUI tree, verify the existence of application crashes and assert if the interfaces between different test sessions are the same as those acquired during the GUI crawling process (Amalfitano et al., 2011).

The team demonstrated the effectiveness of this technique by automatically finding run-time crashes in a small example program. The method also provides means for regression testing the application.

In the future Amalfitano et al. (2011) have plans to execute an empirical validation of the technique by testing the method with several real world applications compared to the simple test application they used in this study. The aim of the future work is evaluating the method's cost-effectiveness and scalability in a real world testing context.

2.2 Automated GUI Testing on the Android Platform

In their paper Kropp, M. and Morales, P. (2010) explain the importance of test automation. The new high-resolution displays and increased processor power of smartphones offer near desktop level user experiences to their users. Testing that the applications actually work correctly in this new environment has become a important success factor. However, as the GUI capabilities of the devices increase, the manual testing of the GUIs has become impractical and susceptible to errors. This has led to a situation where it is highly important for an efficient mobile application development to use some form of automated GUI testing.

The paper describes two approaches for automating mobile GUI application testing, the Android Instrumentation Framework and the Positron Framework, and explores the strengths and weaknesses of both of the approaches.

The Android Instrumentation Framework is an integral part of the Android development environment. Its test suites are based on JUnit, which makes it fairly easy to learn for a tester with JUnit experience (see Figure 2). Instrumentation refers to the ability to monitor all the interactions that the application has with the Android system by inserting tracking code, debugging techniques, performance counters, and event logs into the code, which also allow measuring its performance and control its behavior. The Android Instrumentation Framework includes supporting test classes, which allow the management of starting, running, controlling and terminating of the application in test mode (Kropp, M. & Morales, P., 2010).

The Android Instrumentation Framework's low level API lets the test code to execute simulated key presses and gestures to mimic user input on the device.

Kropp, M. & Morales, P. (2010) describe The Positron framework as a client-server model built on top of the Android Instrumentation Framework in order to manage the activity's resources, offering a Selenium-like (a tool for automating web applications) higher-level method for writing and executing test cases. In the Positron framework, each test case is treated as a client which connects to a server component which runs the activity we are testing. The underlying communication infrastructure and utilities required by the server are provided by the framework. The

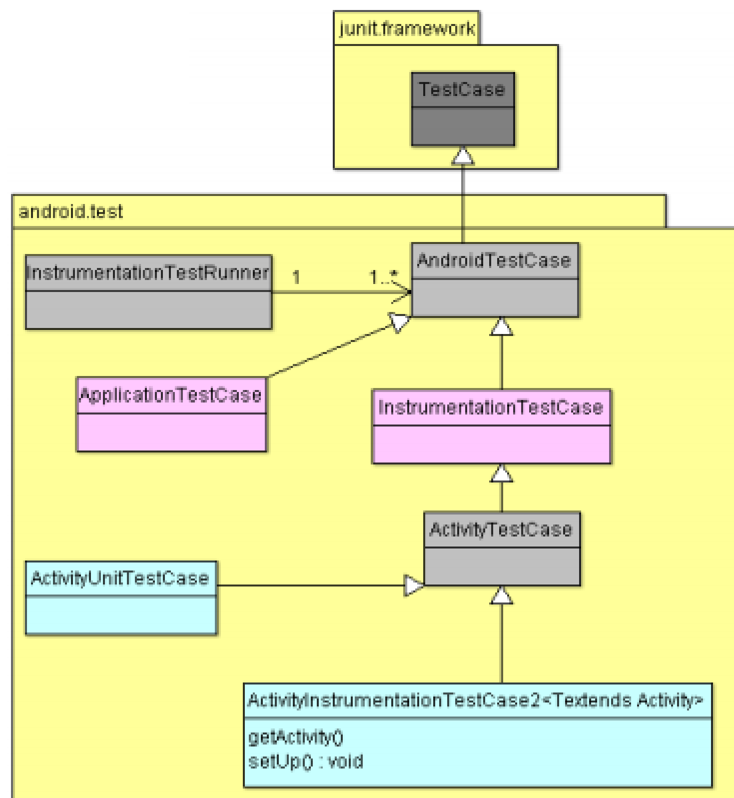

Figure 2: Android Instrumentation Framework Class Diagram (Kropp, M. & Morales, P., 2010)

Positron framework implements its own set of Selenium-like commands to facilitate the automated running of high-level GUI test suites.

According to Kropp, M. & Morales, P. (2010), while the Android instrumentation framework offers greater flexibility and direct access to the GUI controls through its low-level API, it also forces the tester to write more test code, which increases the possibility of errors and causes bigger effort to maintain the test cases when something changes in the application. The Positron framework provides a higher-level interface for writing automated GUI tests than the Android instrumentation framework, which reduces the effort for both writing and maintaining the test code significantly. The strengths of both the frameworks are: use of instrumentation for managing UI resources through the activity; the possibility for user interaction simulation by sending key events; execution on the target platform; usage of standard JUnit assertions in verifying the state and the behaviour of the GUI. Among the weaknesses of these approaches are that the test case writer must have a detailed knowledge of the source code of the application under test to find the correct UI

resources in the code (Kropp, M. & Morales, P., 2010).

Even though the Android platform as one of the currently most emerging environments, this area is still in its beginning. The two analysed frameworks provide basic GUI testing functionality on various levels, but compared to GUI desktop testing tools, both frameworks yet show notable limitations (Kropp, M. & Morales, P., 2010).

2.3 Automating GUI testing for Android applications

In their paper Hu, C. and Neamtiu, I. (2011) report on a bug study and describe an approach for detecting Android GUI bugs.

The bug study and categorization of Android-specific bugs showed that a significant amount of Android bugs manifest themselves in a way that is markedly different from the more traditional, e.g. desktop/server application bugs. To identify and detect the most frequent Android bug categories, Hu, C. and Neamtiu, I. (2011) performed an empirical study where they collected and categorized bugs on 10 popular applications in the Android Market, Google's official repository for Android applications.

In their method of detecting Android GUI bugs Hu, C. and Neamtiu, I. (2011) used a combination of test case and user input generation which were followed by runtime monitoring and log file analysis. The team used JUnit to generate the required test cases. After generating the test cases, Monkey, an automatic event generation tool was used to create the required used inputs in both deterministic and random ways and to send those generated events to the target application. While the test was running, they recorded the interaction between the system and the application in a log file which was then analysed for potential bugs after the test case finished running.

Activities are the main GUI components of an Android application; an activity error usually occurs due to incorrect implementations of the Activity class.

Event errors occur when the application performs a wrong action as a result of receiving an event. By design, Android applications are expected to be prepared to receive and react to events in any state of an activity they happen to be in, e.g. an application must be able to handle the interruption caused by an incoming phone call in every state. If developers do not provide correct implementations of event handlers associated with certain states, the application can enter an incorrect state or crash as a result of an event (Hu, C. and Neamtiu, I., 2011).

Dynamic type errors arise from runtime type exceptions.

Program	Activity bugs		Event bugs		Type errors	
	Old	*New*	*Old*	*New*	*Old*	*New*
Skylight1	3	0	2	3	0	0
CMIS	0	0	0	0	0	0
Delicious	0	0	0	0	0	0
ConnectBot	2	2	6	2	2	0
DealDroid	1	0	0	0	0	0
Rokon	0	0	6	0	2	0
Andoku	0	1	0	0	0	0
Opensudoku	1	0	1	1	0	0
GuessTheNumber	1	0	1	0	0	0
MonolithAndroid	0	0	2	0	0	0
Total	8	3	18	6	4	0

Figure 3: Old (re-discovered) bugs and new (not previously reported) bugs (Hu, C. & Neamtiu, I., 2011)

The method proved effective in finding some types of errors (Figure 3): it was able to discover the existing, already known bugs while finding some new bugs from the 10 applications they selected for the paper. (Hu, C. and Neamtiu, I., 2011).

2.4 Experiences of System-Level Model-Based GUI Testing of an Android Application

The Takala, T., Katara, M. & Harty, J. (2011) paper describes experiences in model-based graphical user interface testing of Android applications. Takala et al. (2011) describe an implementation of model-based testing and test automation on Android, including details on application modeling, test design and execution and the kind of problems that were found in the application which was tested during the process.

According to Takala et al. (2011) model-based testing (MBT) is a testing method where the system we are interested in testing is described with a formal model in such detail that the model can be used to automatically generate test cases. The test case generation is based on algorithms that process the model of the application and generate the desired tests. The advantages of MBT are the reduced need for test script maintenance as we are maintaining the models of the application instead of large sets of test scripts and a basically unlimited variety of different tests that can be generated from the models.

To make it possible to automate the GUI testing of Android application, Takala et al. (2011) needed to be able to send the application GUI events, such as key presses, and to verify the state of the GUI. The team solved the first requirement by using the

network interface of the Monkey application. The Android's Window service was selected as the way to verify the GUI contents (see Figure 4).

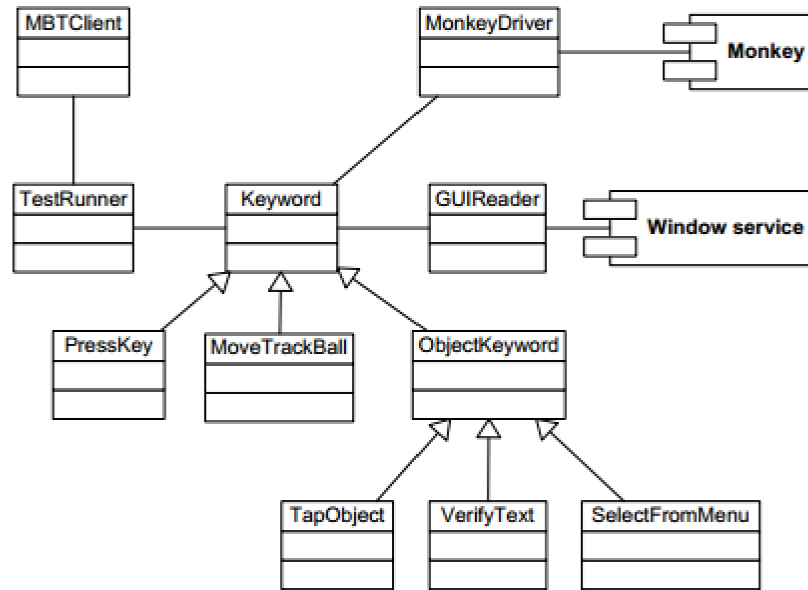

Figure 4: Architecture of the keyword-based test automation tool for Android (Takala et al., 2011)

One way to approach the design of GUI tests is the keyword-driven testing method. In keyword-driven testing the low-level UI operations and API calls are abstracted using keywords. Keywords usually describe basic user events, such as pressing hardware keys, tapping or dragging objects on the screen, or pressing a GUI button. Because the usage of keywords creates an abstraction between the test scripts and the actual low-level functionality, they are easier to maintain.

Takala et al. (2011) decided to use TEMA tools for the study. TEMA Tools is a model-based testing toolset which was originally created for the testing of Symbian S60 GUI applications, but has also been used in Linux, Android, Java Swing and Qt/Maemo environments according to the team. TEMA tools contain tools for all the different aspects of MBT: test modeling, test design, test generation and test debugging. On the Android device, the MBTClient class implements the client side TEMA tools test engine and runs test cases generated from the models. In addition to running whole model-based test cases, the TEMA test automation enables keywords to be executed straight from a file or from an interactive prompt.

The paper focused on a case study that was performed with a popular Android application, the BBC News Widget. The team's goal was to present real-world data using the testing method on Android platform and to discuss if there are benefits

in using the model-based testing in comparison with the more traditional, usually low-level, methods of graphical user interface testing. The second product of the paper was a description of a keyword-based tool for test automation that was implemented for the Android emulator during the case study.

The modeling method proved to be an effective way to find bugs in the application. Although the application has been in the Android marketplace for some time now, the method was able to find new bugs in it. Also, as the testing models do not usually change much between releases, the team now has a relatively easy-to-maintain way to do regression testing for the next version of the application. Takala et al. (2011) claim that setting up a regression test that goes through all actions in the model can be done quite easily in a few minutes.

2.5 GUI testing using computer vision

The paper by Chang, TH., Yeh, T. & Miller RC. (2010) presents Sikuli Test, an approach to GUI testing that uses computer vision to help GUI testers automate their tasks. In Sikuli Test the testers write a visual GUI test scripts which use pictures to specify which of the GUI components to interact with and what visual feedback to be expected. Testers are also able to generate test scripts by demonstrating the test actions. This is done by recording the user inputs and the screenshots of the application under test, identifying the components the tester interacted with and the visual feedback associated with the interactions. (Chang et al., 2010).

The testing method described in the paper worked well within the goals the team had set to it, but there were some big drawbacks to the method: While Sikuli Test can cope with situations where known visual assets appear or disappear on the screen, it is unable to handle situations where something unexpected is shown on the screen or something is missing. For example, if an image is accidentally placed in a blank area where there should be no images, Sikuli Test is unable to detect this error because it only tests the areas which should change during the test.

Another major drawback is that Sikuli Test is designed to only test the GUI's outward appearance. It does not understand what is going on below the visible GUI. As an example, Sikuli Test can assert that when the user presses the Delete button, the correct visual feedback is displayed, but it has no way of verifying that something is actually deleted from the device. This may require using another testing method in conjunction with the one (Chang et al., 2010).

3 Analysis of the testing methods

In this section I will try to answer the following questions about each of the methods described in the previous chapter:

- What method was used to automate the testing?
- What testing tools were used? What are the underlying mechanics in the Android platform the tool uses?
- What kind of tests does the tool support?
- Are testers able to write their own scripts?
- Is the script language easy to use? Does writing scripts take a lot of effort? This is hard subject to quantify, but some general idea of the difficulty involved will do.
- Are the test scripts easy to maintain if the target application of the tests or the device we are running the tests on changes?
- What kind of results were reported in the paper? Did the tool work as expected?
- What are the main weaknesses of the method?

Does the method satisfy the requirements that have been deemed important in the research question? How about the secondary factors? I will divide these results into groups based on the type of testing the tool supports. From these groups one can see the strengths and weaknesses of each method compared to each other and select the one that is most practical for the specific testing needs at hand.

3.1 The GUI crawler

Kropp, M. & Morales, P. (2010) automated their tests by a GUI crawling method using A^2T^2 testing tool. The GUI crawler uses Robotium, a test framework created for testing Android applications. Robotium provides means for the run-time analysis of the Android application's components. The tool supports crash testing and regression testing.

This method seems to be mostly focused on automatically crawling the applications UI and generating the crash test cases, and as such does not offer much flexibility for the testers to write their own tests.

In the paper Kropp, M. & Morales, P. (2010) used the method on a simple calculator application. The crawler successfully acquired the GUI tree of the application and generated 17 crash test cases.

The test cases found some crashes from the application. For example, one crash was caused when pressing a certain button on the calculator. The crash was caused by the lack of a try/catch exception handling code block for a input of a non-numeric value in a input TextEdit widget (Kropp, M. & Morales, P., 2010). I believe these types of errors are among the most common ones found with this type of testing.

The lack of scripting available to the testers and the limited scope of crash/regression testing poses a serious limitation to the usefulness of the tool in many testing activities. I could not find any outside references on the testing tool used, if it was open source or even available somewhere.

If there is no need for other types of testing, the paper showed that the method can be used for running crash and regression testing and that it is capable of detecting some types of errors in a completely automatic manner, which certainly minimizes the effort in writing the test scripts.

3.2 Android Instrumentation Framework and Positron Framework

Kropp, M. & Morales, P. (2010) used two methods to automate their tests: the Android Instrumentation Framework and the Positron Framework. The Android Instrumentation Framework is an integrated part of the Android SDK, while the Positron framework is a client-server model built on top of the Android Instrumentation Framework. In both of the approaches testers are able to write their own test cases.

The Android Instrumentation Framework test cases are basically JUnit cases which should make using the method easy for an experienced JUnit tester according to Kropp, M. & Morales, P. (2010). This approach allows for writing test cases at very low level, which ensures efficient runtime and fast response.

The Positron framework offers a higher-level interface to write the GUI test suites with. It implements a client-server model where each test case is treated as a client, which connects to a server component which runs the activity. The upside of this model is that the amount of code needed is less than in the Android Instrumentation Framework, but at the same time the framework connects to the application under test each time when the test class needs to use activity resources, which slows down the execution.

Both the methods could be considered as viable alternatives for some testing

tasks with exceptions, i.e. the Positron framework would not be ideal for performance testing due to its slow execution times. Both of the approaches should be reasonably easy to get into for a tester who is familiar with JUnit or some other comparable approach.

The maintainability of the scripts produced with these methods is probably not the highest, at least when talking about Android Instrumentation Framework, as we are dealing with pretty low-level cases with no inherent support for abstraction. The authors did not offer any real world examples of using the method or their effectiveness in the paper.

To use instrumentation, the tester must have access to the application's source code. Instrumentation can only control applications which have been started in the same process as the actual testing application, and as such it is not suited for system-wide GUI testing where interactions between multiple applications are necessary. (Takala et al., 2011).

3.3 JUnit and Monkey

Hu, C. & Neamtiu, I. (2011) used JUnit for test case generation. JUnit was used to test whether an activity was properly created, whether the activity performed according to UI specification and that the activity's state was correct. The team used Monkey for automatic event generation. To discover a wide range of issues, the team used random sequences: the sequences were generated using Monkey, and input to the application under test.

After the test cases were generated, the team ran them on the application through the Dalvik VM (the Android platforms virtual machine). To monitor the execution of test cases, the VM was configured to log the details of each test case into a trace file. The traces captured three kinds of events: GUI events, method calls, and exceptions. The VM operation was monitored to detect application bugs that cause the VM to shut down prematurely. These log files were further analysed to identify potential bug patterns.

The team's method is quite similar to Kropp, M. & Morales, P. (2010) use of The Android Instrumentation Framework, except that Hu, C. & Neamtiu, I. (2011) generate the tests automatically. This automatic test case generation might be a good thing when we just need an automatic way to test the GUI components getting used randomly, but it does not suit all testing needs. As the method uses JUnit, the testers are able to write their own testing scripts, but that defeats the purpose of the method. Because of JUnit it should be reasonably easy to get into for experienced

testers. The test scripts should be easy to maintain as they are all automatically generated.

Hu, C. & Neamtiu, I. (2011) found quite a few already known and new bugs of certain types using this method (see Figure 3). As we do not have any facts about the real number of bugs in the tested applications, it is hard to say anything conclusive about the effectiveness of the method.

3.4 MBT with Monkey and TEMA tools

Takala et al. (2011) used a model-based testing approach where they used Monkey for even generation and the Window service to verify the GUI's state. The Android device is connected to TEMA tools, a tried and true set of model-based testing tools (see Figure 5).

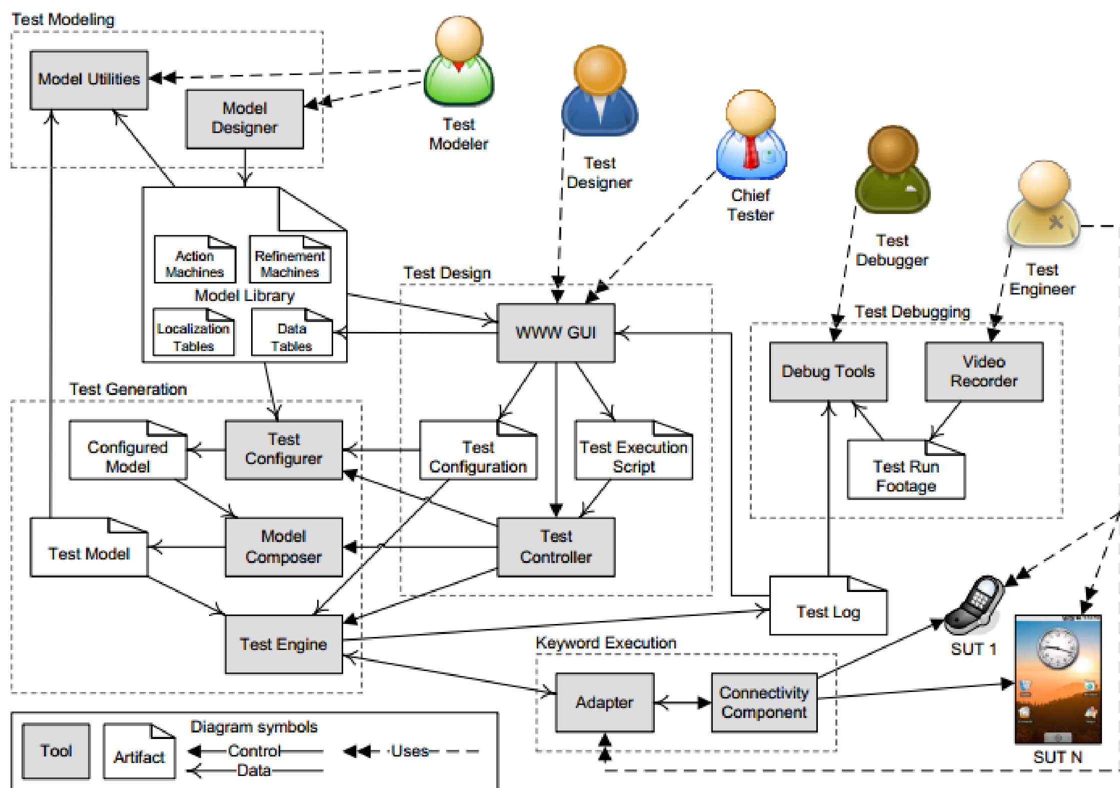

Figure 5: Architecture TEMA tools (Takala et al., 2011)

TEMA tools have a wide variety of tools for different parts of the testing process: For the test modeling there are tools for model design with a web GUI for designing

test objectives, and several utility tools. The test generator uses a variety of algorithms which in turn use the models and test objectives to generate test cases. The test debugging tools help in discovering what went wrong in failed a test case. Finally, the keyword execution tool runs the test cases on the SUT and is the only part of the tools that is dependent on the platform (Takala et al., 2011). In the case study the Android test automation tool was used as the keyword execution tool.

The approach allows for the testers to create their own models of the application under test. As a side benefit the actual process of creating the models is an effective way for manually testing the application. The modeling part of the testing task requires more effort than in the methods where the cases are generated automatically, but after the models are complete, the separation of the GUI functionality into models makes the maintenance a lot easier than dealing with low-level test scripts such as in JUnits case. As an example, Takala et al. (2011) noticed that the Android platform update to version 2.2 introduced some changes to the GUI e.g. in the home screen, which in turn required only some small changes to the model. Eventually the platform dependency was handled by using keywords to abstract the platform version differences in a way where the testing tool checks the platform version and chooses the appropriate action based on that information.

Model-based testing has been successfully used in real-life mobile device testing projects on different platforms such as S60 and Maemo (Takala et al., 2011).

The team reported 14 bugs found from the target of their case-study, the BBC News Widget.

The team mentions one disadvantage in the tool: it uses parts of the Android platform which are not included in the public API. As an example of the problem the team mentioned the possibility of format changes in data returned by some of the non-public APIs which in turn would some maintenance overhead between versions.

In my opinion this method is perhaps the only serious answer to professional Android GUI testing automation covered by this literature review. It has robust tools and tried-and-true approach to actually making a maintainable and flexible testing suite.

3.5 Sikuli Test

Chang et al. (2010) used Sikuli Test for test automation. Sikuli Test is a visual automation technology to test graphical user interfaces using screenshot images. The testers are able to write their own scripts and also use a "record-playback" mecha-

nism where the testers can record interactions which are needed for the test case. These actions as well as the screenshots of the device are saved and the associated user inputs and visual assertions are converted into a test case automatically. The created test case, when executed, repeats the actions just as if the testers were executing the same test case themselves. Sikuli Test is platform independent, and can be used to test applications on an device emulator and even in a different mobile platform than Android (Chang et al., 2010). This differentiates Sikuli Test from the other methods described in this paper, which are mostly concentrated on Android platform.

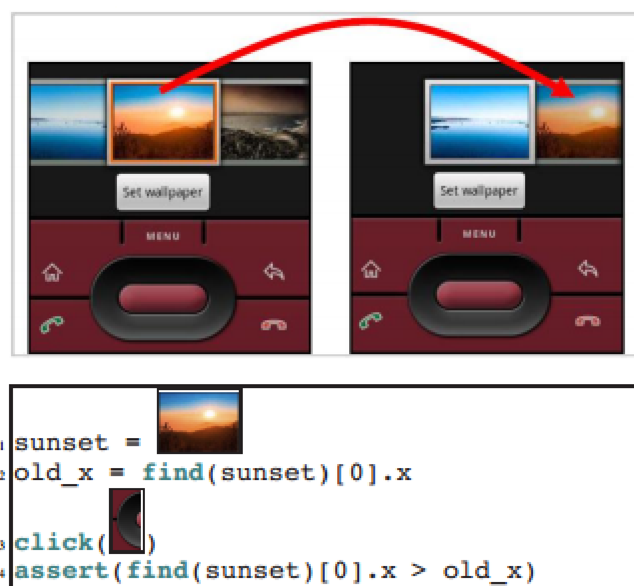

Figure 6: Sikuli Test script example (Chang et al., 2010)

Because of their visual nature, the testing scripts seem relatively easy to create and understand even for someone who is not experienced with the tool (Figure 6). The reusability analysis done on two applications showed the scripts were quite easy to maintain as long as the application's GUI evolved incrementally.

The method seems to be good for regression testing, especially if we are more interested in end-user experience, as the method is excellent for checking what is actually going on on the device's screen compared to asking some arbitrary variable values from a low-level API. As a drawback, the method is designed to test the GUI's visual feedback and does not provide a way to check the actual internal functionality i.e we can only test for things we can see. Chang et al. (2010) do not make any comments on the performance of the method, but I suspect that taking screenshots and comparing captured images with the reference ones is not the fastest process,

so I would be cautious to use this method for any time-critical testing. In addition to the problems in coping with unexpected visual feedback, Sikuli Test is limited to emulator environments, which further limits its usability.

4 Conclusions

The area of test automation in mobile devices is still very young and there are many different ways to handle the task. All the studied methods have their own strengths and weaknesses, some of them are better for a specific task than others.

All the methods described in the papers have their uses and can be just the right thing for a certain project, however in my opinion, two of the methods I explored proved the most interesting for further study: the model based testing with TEMA tools by Takala et al. (2011) and the computer vision utilizing Sikuli Test by Chang et al. (2010). Both of these methods satisfy the requirements that are common in good testing tools: the tester's ability to write his own scripts, the ease of maintenance of the scripts and the minimized effort of the script creation. It is interesting to note that both of the methods use external testing tools to manage the testing, which probably is a good thing considering the limited resources in a smartphone and the potential performance impact of heavy test framework running on the device.

Sikuli Test seems like a very potent tool for testing application UI's functionality. If the Sikuli Test team manages to get the test runner to work on actual devices instead of just emulators, this will be very interesting possibility for organisations with this kinds of testing needs.

MBT with TEMA tools, on the other hand, offers a solid, tried-and-true platform for automating test case generation and test execution. The focus on modeling good, reusable components which are easy to maintain is essential for bigger professional-level projects and the versatility of the tool implies, it will suit most testing needs an organisation might have.

As the testing methods and needs are continually evolving, we need also to re-evaluate our view on the testing tools from time to time. None of the methods studied here are yet mature enough to be the perfect solution for every case.

References

Amalfitano, D., Fasolino, AR. & Tramontana, P. 2011. *A GUI Crawling-Based Technique for Android Mobile Application Testing*. 2011 IEEE Fourth International Con-

ference on Software Testing, Verification and Validation Workshops (ICSTW), pp. 252–261.

Chang, TH., Yeh, T. & Miller RC. 2010. *GUI testing using computer vision*. CHI '10 Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 1535–1544.

Hu, C & Neamtiu, I. 2011. *Automating GUI testing for Android applications*. AST '11 Proceedings of the 6th International Workshop on Automation of Software Test, pp. 77–83.

Kitchenham, B., Pretorius, R., Budgen, D., Brereton, OP., Turner, M., Niazi, M. & Linkman, S. 2010. *Systematic literature reviews in software engineering – A tertiary study*. Information and Software Technology, 52(8) pp. 792–805.

Kropp, M. & Morales, P. 2010. *Automated GUI Testing on the Android Platform*. Proceedings of the 22nd IFIP International Conference on Testing Software and Systems: Short Papers, Montreal: CRIM, pp. 67–72.

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