MT4A: A No-Programming Test Automation Framework for Android Applications

Tiago Coelho
Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
tiago.coelho@fe.up.pt

Bruno Lima
INESC TEC and Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
bruno.lima@fe.up.pt

João Pascoal Faria
INESC TEC and Faculty of Engineering, University of Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jpf@fe.up.pt

ABSTRACT

The growing dependency of our society on increasingly complex software systems, combining mobile and cloud-based applications and services, makes the test activities even more important and challenging. However, sometimes software tests are not properly performed due to tight deadlines, due to the time and skills required to develop and execute the tests or because the developers are too optimistic about possible faults in their own code. Although there are several frameworks for mobile test automation, they usually require programming skills or complex configuration steps. Hence, in this paper, we propose a framework that allows creating and executing tests for Android applications without requiring programming skills. It is possible to create automated tests based on a set of pre-defined actions and it is also possible to inject data into device sensors. An experiment with programmers and non-programmers showed that both can develop and execute tests with a similar time. A real world example using a fall detection application is presented to illustrate the approach.

CCS Concepts

•**Software and its engineering** → **Application specific development environments;** *Software testing and debugging;*

Keywords

Software testing, Mobile applications, Testing framework, Test automation, Android

1. INTRODUCTION

Nowadays, mobile applications are getting more and more complex and, besides the development challenges, testing them to ensure their stability and robustness is one of the biggest challenges [17]. Many mobile applications have been developed in critical areas such as transportation, healthcare and banking. Hence, testing is a fundamental life-cycle activity, with a huge economical and societal impact if not performed adequately [23]. Nevertheless, there are some companies and/or developers that do not develop adequate tests for their applications because of reduced deadlines and because, sometimes, developers are very confident regarding possible faults in their own code. Although there are several frameworks for mobile test automation, they usually require programming skills or complex configuration steps.

Hence, to facilitate the creation of automated tests by non-developers, we present a framework to build black-box tests [18] for Android applications, without programming, in a quick way. Through a desktop application, anyone, even without any knowledge about programming, can develop black-box tests with just a few clicks, run the tests in the emulator or on a real device and check the test results in the desktop application. With this high-level of abstraction, we intend to boost the development of tests for Android applications and allow any person to develop those tests.

The rest of the paper is organized as follows: section 2 describes the state of the art. In section 3 is described the proposed approach. A real world example is presented in section 4 to demonstrate the framework. Validation results are presented in section 5. Finally, section 6 concludes the paper and presents the future work.

2. STATE OF THE ART

As mobile applications become more complex, the software engineering tools, frameworks and processes are essential to ensure the development of high-quality software in a timely and cost-effective way [2]. According to Wasserman [26], there are important areas for research in software engineering related to mobile devices. The research on new methods and techniques to test applications for mobile devices, such as Android devices, is an important area of research [2] because of several factors, such as device fragmentation and the number of different versions of operating systems available in the market.

This section provides an overview of existing test automation frameworks for mobile applications, as well as acceptance and regression test automation frameworks accessible for non-programmers.

2.1 Test Automation Frameworks for Mobile Applications

Test automation is fundamental in iterative and evolutionary development, to allow repeating test execution with little cost. Besides that, all statistic methods to assess software reliability are based on fully automatic testing methods for performing sufficient test cases [3].

Thus, some frameworks have been developed to help users in software testing. The next subsections present some test automation frameworks existing in the market for Android applications.

2.1.1 Appium

Appium [14] is a framework to develop tests for mobile applications for iOS and Android. It supports Android taking advantage of UI Automator [11] and Selendroid [5], iOS through UI Automation and web with the Selenium driver for Android and iOS. Internally, it uses the JSONWrite protocol to interact with Android and iOS. One of the advantages is that it is possible to develop test scripts in almost every programming language but it is necessary to code to develop tests – something that our framework wants to remove.

2.1.2 Calabash

Calabash [27] is a framework for native or hybrid applications, Android and iOS. Calabash tests are described in Cucumber [15] and then converted to Robotium [19] or Frank [25] in real time (if the application is for Android or iOS, respectively). It supports about 80 controllers and new controllers can be implemented in Java or Ruby. Its simple syntax allows people without much technical knowledge to run tests on both platforms but it is necessary to use the terminal and have some scripting skills.

2.1.3 Robotium

Robotium [19] was one of the most used frameworks in the early days of the appearance of Android. This framework was created to make it easier to test user interfaces for Android applications. It is open-source and extends JUnit, but, as with Appium [14], it is necessary to code to develop those tests.

2.1.4 Selendroid

Selendroid [5] is a framework to automate tests for native or hybrid mobile applications. It is possible to write tests using the Application Programming Interface (API) of Selenium 2 because Selendroid reuses the same infrastructure of Selenium for the web. It can be used in emulators or real devices, being mostly used for testing the user interface.

2.1.5 Espresso

The Espresso framework [12] provides a set of APIs to build user interface tests for Android applications. With the provided APIs it is possible to write automatic tests that are concise and that run reliably. Espresso is more suited to write tests using the white-box technique, where the test code uses implementation details from the application to be tested.

2.1.6 UI Automator

UI Automator [11] provides a set of APIs to build tests that interact with the user or system. With these APIs it is possible to perform operations such as open a Settings menu or launch an application in a test device. This framework is suited to write tests using the black-box technique, where the test code does not depend on details of the internal implementation of the application being tested.

2.1.7 Ranorex

Ranorex [9] is a Graphical User Interface test automation framework for testing mobile, desktop and web applications. It is provided by Ranorex GmbH and it uses standard programming languages such as VB.NET and C#, working on top of the Windows operating system. It has an editor, Ranorex Studio, where developers can perform coding, debugging and project management activities.

2.1.8 Research Frameworks

Kim et al. [13] proposed a method of developing performance tests. For that is used a database established through the performance tests conducted on the Android emulator to the level of unit test. It also provides a tool that supports the proposed method for performance testing.

Amalfitano et al. [2] proposed a framework that uses an algorithm that tracks an application and automatically builds an application UI template and get test cases that can be automatically executed. This framework is developed in conjunction with the Robotium [19] to analyze the components of an Android application running. Using this information, it generates events that may be triggered for various trace components and detect faults.

Delamaro et al. [4] proposed a framework to perform tests using the technique of white-box mobile applications. This technique is supported by a testing environment that provides generation facilities, execution and data collection, comparing the static analysis results with results obtained during execution.

2.1.9 Summary

Currently, there are already plenty of frameworks to develop tests for Android applications. However, almost all, in addition to the prerequisites, require a set of configuration and coding activities to develop tests. Even when it is not necessary to code, it is necessary to know how to manipulate the command line to run tests or to see the results of those tests. On the other hand, nowadays, the sensors are widely used and their testing is very important and these frameworks do not support tests for all sensors available in the device.

In Table 1 it is presented the framework's summary in order to compare them: if they support tests based on some high-level tests specification (for instance a XML or JSON file as input of tests defined with Cucumber), if they allow to test all types of sensors as GPS, accelerometer, among others, and if it is necessary to program in order to develop tests. Most frameworks do not allow testing all types of sensors. Although Calabash is close to fulfill all the requirements of Table 1, our framework differs from Calabash because it allows testing all types of sensors and it is not necessary to know how to manipulate the command line: it is all done in an appealing interface.

	Specification based	Sensors Test	Code Abstraction
Appium	No	Partially	No
Calabash	Yes	Partially	Yes
Robotium	No	Partially	Yes, with Recorder
Selendroid	No	No	No
Espresso	No	Partially	No
UI Automator	No	Partially	No
Ranorex	No	Partially	Yes, with Recorder

Table 1: Comparative analysis of test automation frameworks for mobile applications

2.2 Acceptance and Regression Resting Frameworks

This section provides a short overview of some acceptance and regression test automation frameworks existing in the market, accessible to some extent for non programmers.

2.2.1 FitNesse

FitNesse [6] is an open-source framework where customers, testers and programmers can test and collaborate to learn what their software should do and can compare its required features with what it actually does. It is based on the Framework for Integrated Tests (FIT) and tests can be written in many languages such as Java or Python. Although this framework provides a high-level of abstraction from code to define tests in a tabular way, it is necessary to write some fixture code to link the defined tests to the implementation under test.

2.2.2 Robot Framework

Robot Framework [7] is an open-source test automation framework for acceptance testing. Test cases are defined in a simple scripting notation, utilizing a keyword-driven testing approach. New keywords can be created by composing existing ones (using the same syntax that is used to create test cases) or by extending keyword libraries implemented in Python or Java. This framework provides a lot of abstraction from code but it is often necessary to implement application specific keywords in Java or Python.

2.2.3 Robotium Recorder

Robotium Recorder [24] is an extension of Robotium, with an annual cost, which is used to develop regression tests for Android applications using the record-playback technique. Interactions of the user with the device or emulator can be recorded into a test script, which can be replayed later automatically. It has the limitation of just generating tests for user interactions with the device's UI and being applicable only for regression testing (and not for initial testing).

2.2.4 Ranorex Recorder

Ranorex Recorder [9] is a part of Ranorex with the functionality of capture-replay, which provides maintainable recordings via user actions in the application under test (AUT), transforms those actions into code and generates report files for error detection. It has limitations similar to Robotium Recorder.

2.2.5 Summary

Both the FitNesse and Robot frameworks allow defining tests without coding, even if later it is necessary to write some code to automate the tests defined. Although they attempt to abstract the person who develops the test from the code, Robotium Recorder and Ranorex Recorder are suitable for regression tests and mainly focused on user interactions. The approach followed in our framework aims at further simplifying test definition (by just selecting actions and writing values to insert / find in the application) and eliminating any need to write fixture code.

3. THE MT4A FRAMEWORK

3.1 Architecture and Functioning

Figure 1 depicts the architecture of Mobile Testing For All (MT4A) for developing and executing tests for Android applications without programming. The architecture is divided in three components: Server, Desktop and Device/emulator.

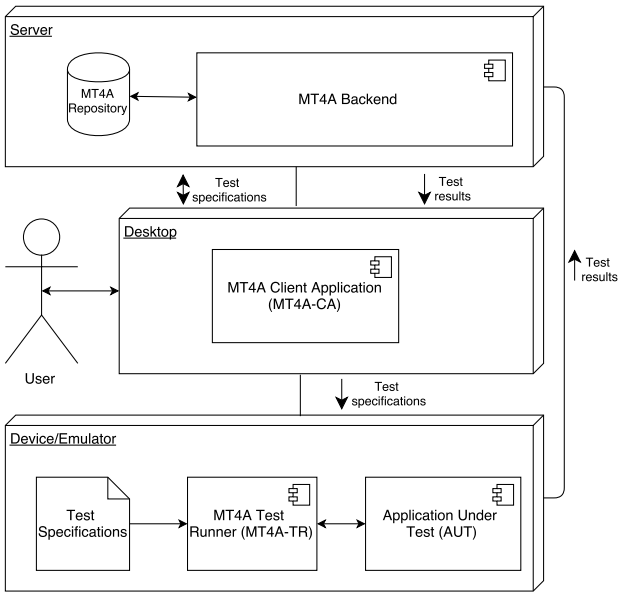

Figure 1: Framework architecture (annotated UML deployment diagram).

The life-cycle of a test, represented graphically in Figure 2, starts at the Desktop node.

The users start by logging into the MT4A Client Application (MT4A-CA). The login system was designed to allow users to develop their own test suites for one or more packages.

At the desktop application the user can then see which devices or emulators are connected to the desktop computer through Android Debug Bridge (adb) [10] and then, choosing one of these devices, can see all the packages installed in that device (it is also possible to see the application name and icon if that application is at Google Play Store). The user has to select the target application in the desktop application (Client App.), identifying it by its name or package.

Then the user develops the tests just using clicks and entering some values that identify the elements to interact with or to be inserted in the application.

A JSON file containing the specifications that the user is defining in the desktop application is created in parallel while the user develops a test and is stored in the server.

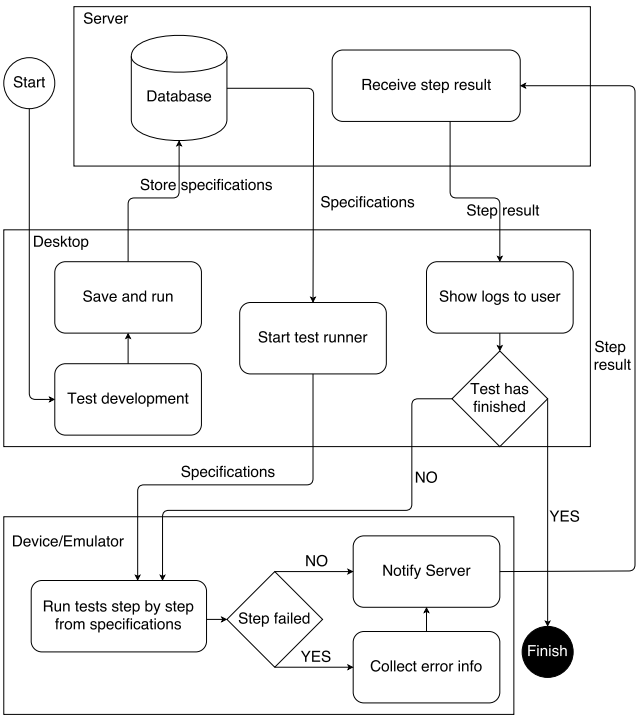

Figure 2: Test flow.

Later, when the user activates the option to run a test, the corresponding file is sent to the device or emulator using the adb and the Test Runner, which is responsible for reading the test specification file and run the tests, is also triggered via adb.

Then, the Test Runner in the device/emulator layer runs the tests step by step and notifies the server about the success or failure status of a particular step via web-sockets. This means that the Internet is a requirement but if at one step of the test, the device/emulator does not have access to the Internet, the status of each step will be saved to be sent later when the Internet is on or the test ends.

In the server layer, when the server is notified with the status of a step by the device, it notifies the Client App. with the same status, and the result is displayed to the user.

3.2 Technologies Used

To develop this framework we used several technologies.

In the desktop layer, it is used Electron, allowing us to "Build cross platform desktop apps with JavaScript, HTML, and CSS" [8] for OS X, Windows and Linux. Inside Electron we used javascript (AngularJS and NodeJS) to build all the User Interface (UI) and all the controllers that build the tests.

In the device layer, we used Java and the Android SDK and the UI Automator framework as a base for our own framework.

In the server layer, we used NodeJS to build the logic and MongoDB for our database where we stored the tests developed by the users.

To inject sensor data from the APK to the emulator, it is used a telnet client that communicates with the emulator.

The communication between the desktop computer and the device is made via adb and the device-server and server-desktop communications are implemented with sockets.

3.3 Test Actions

To build the tests we support a set of actions listed below together with an illustrative example:

- **Press** – can be used to tap or press on a button or other user interface element;
- **Insert** – can be used to insert a value in an input box;
- **Check** – can be used to check if internet connection or device location is on/off;
- **Wait** – can be used to wait for some element to appear on the screen;
- **Set** – can be used to turn on and off internet connection or device location;
- **Verify** – can be used to verify an incoming or outgoing sms, mms or call;
- **Inject** – can be used to inject values into sensors, for instance accelerometer or GPS.

The user can choose one of these actions to make a step for a particular test and the next things to provide for the chosen action depends of which action he chooses for that step.

4. EXAMPLE: FALL DETECTION APP

For demonstration and validation purposes, we used an example from the AAL4ALL project [1], related with a fall detection Android application.

In this application, when a person falls, the application detects the fall using the smartphone's accelerometer and provides the user a message which indicates that it has detected a drop giving the possibility for the user to confirm whether he/she needs help. If the user responds that he/she does not need help (the fall was slight, or it was just the smartphone that fell to the ground), the application does not perform any action; however, if the user confirms that needs help, the application raises two actions in parallel. On the one hand, it makes a call to a previously clearcut number to contact a health care provider (in this case can be a formal or informal caregiver); on the other hand, it sends the fall occurrence for a Personal Assistance Record database and sends a message to a portal that is used by a caregiver (e.g. a doctor or nurse) that is responsible for monitoring this

care receiver. The last two actions are performed through a central component of the ecosystem called AALMQ (AAL Message Queue), which allows incoming messages to be forwarded to multiple subscribers. Before the application starts monitoring falls, it is necessary to configure login credentials and a number to contact in case of fall.

4.1 A Simple Test

Suppose that a user needs to create a test to check the login process in the fall detection app. With our framework, after installing the desktop application, the user is already prepared to create this test. After login (or register if the user doesn't have an account) the user can select the device (and install the Test Runner if he/she is using the framework for the first time) and then the AUT. At the Client App., the devices and applications lists appear as shown in Figure 3.

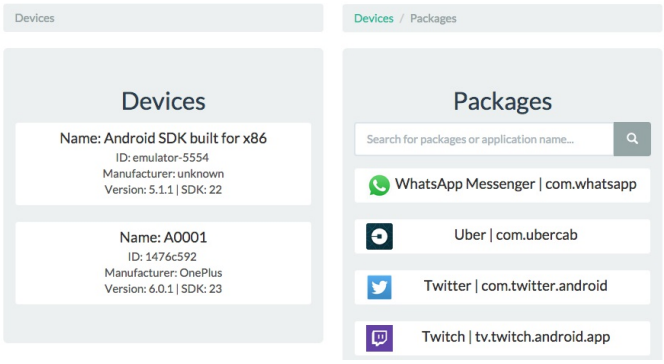

Figure 3: Devices and packages listed at the Client App

The next step is to construct the test. In our framework the test is constructed step by step where each step is one action that Test Runner will run in the AUT. In this particular case, to test the login process, the user needs to perform two steps:

1. insert his ID Card number
2. press a button

To build a test with these steps, the user needs to create a new test step, select an "insert" action, choose the text to insert and choose the element to insert the text (identified by its placeholder). Then, create a new step, choose a "press" action and select the element to press (in this case, the button, using his text). In Figure 4, on the left side, it is represented the AUT for this test case and, on the right side, the test created with MT4A to test the login process.

Now that the user has finished the test, he can run it and see the results in the Client App. The result of each step is presented to the user as shown in Figure 5. If one step fails, a red cross appears instead of the green check mark and a message shows the reason of failure.

4.2 A Complex Test

Let's suppose another scenario. The user needs to create a test to check if a fall occurs when the device is faced with some values at the accelerometer and needs to check if a call is made after the fall. To test this, the steps are:

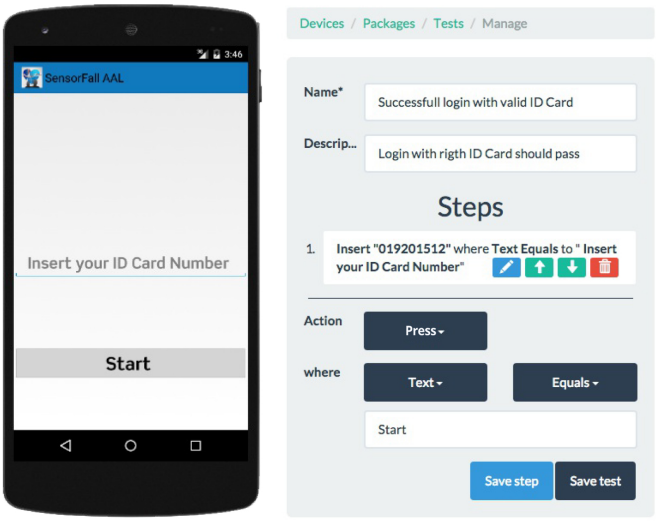

Figure 4: AUT at left and test creation with our framework at right

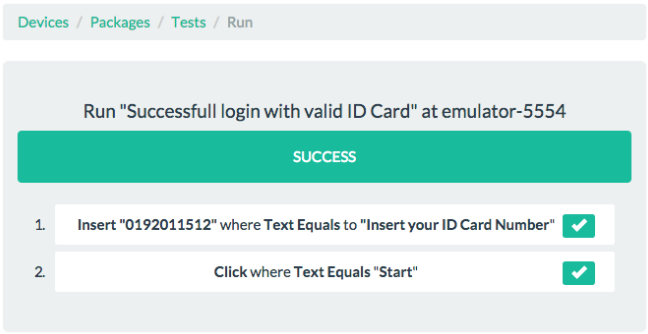

Figure 5: Test result at our framework

1. inject values into the accelerometer
2. wait for an alert asking the user if he/she actually fall
3. press the "Yes" button
4. check if a call is made by the application to a predefined number

The user can make this test through our framework with a few clicks and providing one TXT file with "X Y Z" accelerometer values to inject. Unfortunately, the accelerometer only can be tested in the Android emulator because it is not possible to inject real data into real devices' accelerometers.

Listing 1 shows an example of the JSON file generated by the MT4A-CA for the complex test. This file is sent via adb to the device or emulator for being parsed by the Test Runner and injected into the AUT.

Listing 1: JSON file generated to be used to test the AUT

```
[   {   "type": "details",
        "name": "Test a fall with a call
            after",
        "package_id": "pt.sapo.aal4all",
        "test_id": "d6dc0d12-7280-46fc-
            9605-5363ed"},
    {   "step": 1,
        "type": "inject",
        "sensor": "accelerometer",
        "file" : "accelerometer_values.txt
            "},
    {   "step": 2,
        "type": "wait",
        "ui_object": {
                "find_text": {
                "starts_with": "Did you
                    fall"
         }},
         "timeout" : "15000" },
    {   "step": 3,
        "type": "press",
        "ui_object": {
                "find_text": {
                "equals": "Yes"
        }}},
    {   "step" : 4,
        "type" : "wait",
        "call" : {
                "status" : "outgoing",
                "number" : "123456789"
                },
        "timeout" : "3000"}]
```

5. VALIDATION

In order to validate the proposed framework, we conducted an experiment with 22 volunteers (IT professionals), half with programming skills and half without programming skills. The participants were asked to create and execute two tests cases for the SensorFall AAL app [16] using the MT4A framework - a simple test case and a complex test case (the same test cases described in section 4), within a defined time limit for each test case. We collected the time spent by each participant for each test case. Tables 2 and 3 shows the corresponding statistics for the simple and complex test case, respectively, considering only the volunteers that succeeded in creating and executing the test cases.

	Programmers	Non-programmers
Mean (seconds)	$\overline{X_1} = 107.89$	$\overline{X_2} = 118.5$
Standard deviation	$S_{X_1} = 12.14$	$S_{X_2} = 11.22$
Participants	$n_1 = 9$	$n_2 = 10$

Table 2: Results for the simple test

The results show that, on average, the time spent by programmers and non-programmers are very close. Using the Shapiro-Wilk Normality Test [20], we verified that our data came from a normally distributed population. After that, to verify if the difference of means are statistically significant, we performed a Student's t-test, which is a test to analyze the means of two populations through the use of statistical analysis [21].

	Programmers	Non-programmers
Mean (seconds)	$\overline{X_1} = 425.3$	$\overline{X_2}$ 474.55
Standard deviation	$S_{X_1} = 64.46$	$S_{X_2} = 59.44$
Participants	$n_1 = 10$	$n_2 = 11$

Table 3: Results for the complex test

The two-sample t-test for unpaired data is defined as:

$$H_0 : \mu_1 = \mu_2$$

$$H_a : \mu_1 \neq \mu_2$$

where μ_1 and μ_2 represent the average times for the populations of programmers and non-programmers, respectively.

The first step to verify if the means are not statistically significant is to calculate the t value and the freedom's degrees. To find these values will be used the Equations 1, 2 and 3, corresponding to the t statistic, grand standard deviation and degrees of freedom, respectively.

$$t = \frac{\overline{X_1} - \overline{X_2}}{S_{X_1X_2} \cdot \sqrt{\frac{1}{n_1} + \frac{1}{n_2}}} \quad (1)$$

$$S_{X_1X_2} = \sqrt{\frac{(n_1 - 1)S_{X_1}^2 + (n_2 - 1)S_{X_2}^2}{n_1 + n_2 - 2}} \quad (2)$$

$$d.o.f = n_1 + n_2 - 2 \quad (3)$$

For the simple test case, we get t = -1.9806, $S_{X_1X_2}$ = 11.66 and d.o.f = 17. From the table of critical values for two tailed test with unpaired values [22], we get the critical value 2.11 for d.o.f = 17 and a significance level of 5%. Since |t|<2.11, we conclude that the difference of averages of the two groups is not statistically significant, for a significance level of 5%.

For the complex test, we get t = - 1.8588, $S_{X_1X_2}$ = 61.86 and d.o.f = 19. From the table of critical values for two tailed test with unpaired values [22], we get the critical value 2.093 for d.o.f = 19 and a significance level of 5%. Since |t|<2.093, we conclude that, in the complex test case, the difference of averages of the two groups is also not statistically significant, for a significance level of 5%.

These results show that, with our framework, people without programming skills can not only create automated tests for mobile applications but also that the time spent in creating and executing such tests is similar to the time spent by people with programming skills.

6. CONCLUSIONS AND FUTURE WORK

In this paper it was presented a framework to develop tests for Android applications without programming that supports testing all types of sensors. All the tests can be made with an appealing user interface via a desktop app.

With this framework, we expect to motivate more people to develop tests for Android applications. Experimental

results show that, with our framework, people without programming skills can develop and execute tests with a similar time compared with people with programming skills.

As future work, we intend to: implement an option to run the same test in multiple devices at the same time; develop a new module to allow this framework to be included in continuous integration systems; implement an option to create and edit tests collaboratively; extend the framework to iOS and Windows Phone.

7. ACKNOWLEDGMENTS

This research work was performed in scope of the project NanoSTIMA. Project "NanoSTIMA: Macro-to-Nano Human Sensing: Towards Integrated Multimodal Health Monitoring and Analytics/NORTE-01-0145-FEDER-000016" is financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF).

8. REFERENCES

[1] AAL4ALL. Ambient assisted living for all. http://www.aal4all.org.

[2] D. Amalfitano, A. Fasolino, and P. Tramontana. A gui crawling-based technique for android mobile application testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, pages 252–261, March 2011.

[3] B. Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[4] M. E. Delamaro, A. M. R. Vincenzi, and J. C. Maldonado. A strategy to perform coverage testing of mobile applications. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, pages 118–124, New York, NY, USA, 2006. ACM.

[5] Ebay. Selendroid : Selenium for android. http://selendroid.io/.

[6] FitNesse. Frontpage. http://www.fitnesse.org/.

[7] R. Framework. Robot framework. http://robotframework.org/.

[8] GitHub. Electron. http://electron.atom.io/.

[9] R. GmbH. Test automation for gui test | ranorex. http://www.ranorex.com/.

[10] Google. Android debug bridge | android studio. https://developer.android.com/studio/command-line/adb.html.

[11] Google. Testing support library - android developers. https://developer.android.com/tools/testing-support-library/index.html\#UIAutomator.

[12] Google. Testing support library - android developers. https://developer.android.com/tools/testing-support-library/index.html\#Espresso.

[13] H. Kim, B. Choi, and W. Wong. Performance testing of mobile applications at the unit test level. In *Secure Software Integration and Reliability Improvement, 2009. SSIRI 2009. Third IEEE International Conference on*, pages 171–180, July 2009.

[14] S. Labs. Appium: Mobile app automation made awesome. http://appium.io/.

[15] C. Limited. Cucumber. https://cucumber.io/.

[16] I. C. Lopes, B. Vaidya, and J. Rodrigues. Sensorfall an accelerometer based mobile application. In *Proceedings of the 2nd International Conference on Computational Science and Its Applications, Jeju, Korea*, pages 10–12, 2009.

[17] L. Nagowah and G. Sowamber. A novel approach of automation testing on mobile devices. In *Computer Information Science (ICCIS), 2012 International Conference on*, volume 2, pages 924–930, June 2012.

[18] S. Nidhra and J. Dondeti. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications (IJESA)*, 2(2):29–50, June 2012.

[19] RobotiumTech. Robotiumtech/robotium: Android ui testing. RobotiumTech/robotium:AndroidUITesting.

[20] S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

[21] G. W. Snedecor and W. G. Cochran. *Statistical Methods*. 1989.

[22] J. Stock and M. Watson. *Introduction to Econometrics (3rd edition)*. Addison Wesley Longman, 2011. Professor Stock receives royalties for this text.

[23] G. Tassey. The economic impacts of inadequate infrastructure for software testing, 2002.

[24] R. Tech. Robotium tech. http://robotium.com/.

[25] ThoughtWorks. Testing with frank - painless ios and mac testing with cucumber. http://www.testingwithfrank.com/.

[26] A. I. Wasserman. Software engineering issues for mobile application development. In *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, FoSER '10, pages 397–400, New York, NY, USA, 2010. ACM.

[27] Xamarim. Calaba.sh - automated acceptance testing for ios and android apps. http://calaba.sh/.