Mobile Farm for Software Testing

Maxim Mozgovoy
University of Aizu
Aizu-Wakamatsu, Japan
mozgovoy@u-aizu.ac.jp

Evgeny Pyshkin
University of Aizu
Aizu-Wakamatsu, Japan
pyshe@u-aizu.ac.jp

Abstract

We introduce an approach to user interface testing with a particular focus on non-native GUI based mobile applications. We particularly address the domain of entertainment and education software including mobile games. We describe a prototype system based on inexpensive components and open source software, intended to support product development cycle for companies on lean budget. On the base of a prototype system discussed in this paper we expect to develop a distributed infrastructure that would allow users to use facilities of users' own computers and connected devices as a part of a common testing framework. The approach presented in this work is also suitable for wider range of mobile applications with a high variety of human-computer interaction mechanisms.

Author Keywords

Mobile application; non-native GUI; automated testing framework; time-consuming test suites; smoke testing

ACM Classification Keywords

D.2.5. [Software Engineering]: Testing and Debugging; H.5.2. [Information Interfaces and Presentation]: User interfaces

GUI testing – software testing process, where the test scripts are developed to access the GUI elements programmatically, in order to define and test situations triggered by appearance of certain user controls on the screen.

Automated tests – tests executed automatically with the help of special testing frameworks. Automated testing is an important part of continuous integration.

Smoke tests – restricted test suites aimed at checking whether the whole application works and provides its basic functionality and reacts to user interface controls properly (the term probably came from plumbing testing) [13].

Non-native GUI – user interface which is not based on platform-native GUI. It is usually developed without using standard libraries. GUI elements can be hand-drawn (for example, in the applications developed with Unity).

Introduction

In mobile applications, the user interface (UI) is a crucial element of human-computer interaction (HCI). Regardless of the type of software (whether the question is about desktop or mobile applications), creating open platforms for software development and testing automation is one of strategic parts of software quality assurance [2]. That is why the developers and researchers working in the domain of software quality assurance appreciate the efforts to create specifications for *low cost*, *pragmatic*, *sharable* and *open source*-based solutions enabling scalable process of mobile application testing [15]. Such efforts can be considered as important steps towards creating a better "test automation culture" [9] of application developers.

In particular, writing automated UI tests needs features for accessing applications similarly to software users. In turn, graphical UI (GUI) testing provides an interesting case of testing automation for both desktop and mobile applications [15, 17]. Many testing automation frameworks (such as Jemmy library[1]) allows accessing GUI elements from within the test scripts. This enables defining and testing situations triggered by appearance of certain user controls on the screen, and to perform different user-side operations such as pushing a button, selecting a menu item or tab, clicking or hovering an area, and so on.

Though many testing automation approaches can be equally used for different kinds of software applications, mobile software has important particularities. On the one hand, because of large number of mobile devices with a big variety of characteristics, developers want to be sure that the application works on different devices. From various recent reports we know that the process of verification, whether a mobile application (to be run on large number of devices) behaves as expected, is still challenging and time consuming [18, 7]. In such applications as mobile games, we might have to run the relatively long-lasting process and to collect many screenshots necessary for further analysis of possible application failures. On the other hand, mobile applications often do not rely on a platform-native GUI, but utilize hand-drawn elements designed without using standard GUI libraries. Apart from entertainment applications (such as games), non-native GUI is often used in educational software. Another interesting case is a large number of applications integrated with geographical maps relying on graphical elements which are not commonly supported in standard testing frameworks.

In the above mentioned cases, device emulators (being a commonly used solution) are not enough. Real devices are necessary because we have to check whether our software works on them (and non only on simulators), including the issue of potential intensive CPU and GPU load, in order to reveal battery drain or low performance problems.

[1] https://jemmy.java.net/

Non-Native GUI Applications in Frame of Recent Mobile Software Testing Research

As of today, analysis of peculiar characteristics of mobile software from the perspective of testing automation is one of the common trends in software research agenda. Many studies and practical solutions address the important issues of mobile software testing automation.

In [7] the authors describe the *AutoClicker*, a system aimed at automating tests of large-scale applications running on multiple devices. This work introduces a system for scheduling mobile tests with multiple connected devices (both real and emulated) in parallel.

Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing

"Current state of the art of automated testing tools for mobile apps posses limitations that has driven a preference for manual testing in practice. As of today, there is no comprehensive automated solution for mobile testing that overcomes fundamental issues such as automated oracles, history awareness in test cases, or automated evolution of test cases." [11]

Major Concerns

Large-scale resource-intensive applications; Multiple devices; Tests are time and energy consuming; Reducing costs; Continuous integration is a "must-have" part of the process.

Objectives

Distributed infrastructure that would allow users to use facilities of users' own computers and connected devices as a part of the whole testing framework.

The focus of the work [3] is on reducing the costs of mobile application testing by considering the possibility of test migration, which can be partially achieved for applications that share some functionality or GUI components.

The study [5] addresses the challenges of time-consuming energy tests. The problem of testing device energy consumption is closely related to testing resource-intensive applications: first, it is difficult to measure the energy consumption of a device under test accurately, since the test itself might be executed on a device and drain its battery; second, if device is plugged to a power source (which is required for long-lasting tests), the collected data might be inaccurate due to the impact of charging current.

The tool discussed in [14] detects the Android application crashes and generates the reports containing the detailed description of the application failure context (which includes screenshots, failure reproduction steps, application exception stack traces, etc.). Support for making the failures reported and reproduced is the essential component of a continuous integration pipeline.

Many existing automated UI testing systems for mobile applications work well for native GUI applications (see, for example, [1, 6]) but can not be directly used for the case of non-native GUI based mobile software. Thus, testing non-native applications serves as a good example of an open issue among other challenging problems of mobile software development that require particular attention from the community [11].

The standard approach for arranging mobile software tests is to connect mobile devices to a server running some special software, and to execute test scripts on a remote machine (Figure 1).

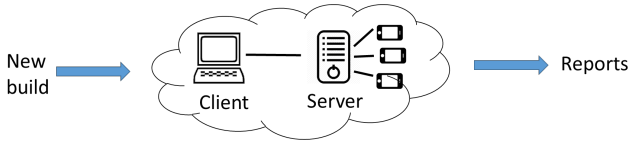

Figure 1: Client-server interaction in a mobile testing environment.

For the server software, there are many known solutions, such as *Appium*[2] and *Calabash*[3]. Practically, a test script is a set of instructions similar to the following fragment:

```
wait 10 sec
tap location (100, 50)
assert that OK button appears
press OK button
```

For native GUI applications, we can implement such scripts relatively easily, since user controls can be accessed programmatically from within automated tests. Unfortunately, in a case of non-native GUI applications (which do not use standard GUI libraries), such a model does not work. Various sources [19], including our own works [17, 16], suggest to use pattern recognition methods to identify GUI elements on the screen. This approach requires using pattern matching and image transformation algorithms, and might cause significant slowdowns of the testing process.

In general, it is difficult to estimate a "typical" test duration: simple smoke tests can reveal the absence of crashes within seconds, while *stress tests*, designed to check the application stability in a long time interval, can take hours.

[2] http://appium.io
[3] http://calaba.sh

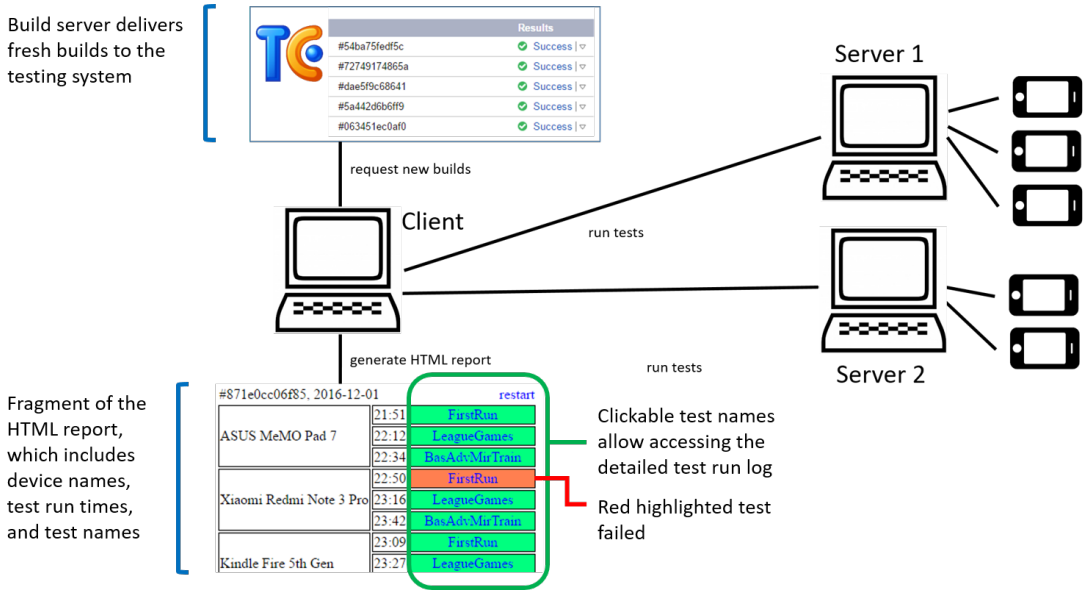

Figure 2: Testing organization: a prototype.

Furthermore, ideally every new build should be tested on a variety of mobile devices.

The easiest way to organize regular large-scale automated testing on real mobile devices is to use a cloud mobile farm provider (such as *Amazon Web Services*, *Bitbar*, etc.). and to obtain hassle-free setup, and availability of hundreds of mobile devices. However, most providers still do not support an adequate variety of devices (in particular, with respect to different vendors). Variety of devices is a concern for own test platform setup too, however, one can select the most interesting devices for a particular case easier.

The cloud farm testing might be up to 0.14–0.17 USD per minute per device, and costs are quite high for a team on lean budget such a startup company or a freelance developer. Assuming that the tests take at least 1 hour on each device per build (and there might be several builds per day), the costs for daily testing of fresh builds with only a dozen of devices may go up to 200–300 USD. Stress testing might require even higher expenses.

Of course, before selecting an appropriate model of automated testing infrastructure (i.e., renting the cloud services or building one's own testing farm), one has to evaluate the cost of deploying and running such a mobile farm against the costs of using the facilities provided by the cloud, including hardware expenses and workload required to set up, run and maintain the system.

Figure 3: A prototype mobile farm with eight connected devices.

Figure 4: Non-native GUI based mobile game.

Our Approach, Its Implementation and Evaluation

Figure 2 shows the architecture of our prototype farm. The system provides a functionality of getting builds from a build machine (such as *TeamCity* [12]), running all tests on all connected devices, generating HTML reports with application action logs and screenshots, as well as sending all the report-related data to the subscribed users of the system (developers, testers, administrating staff, etc.).

The current implementation allows us to analyze a number of important issues related to the problem of building a framework for automated testing of mobile applications on real devices, including differences in how the devices manage the installed software (with respect to different operating systems), and how to connect a reasonable number of devices to a powered hub (with respect to possible battery draining, difference in charging/connection interfaces, charging rate and time, the capability to charge and transfer data at the same time, etc.).

Currently we have a prototype (partially shown in Figure 3), consisting of the following components:

1) *Testing server*: a Windows-based mini-PC, used to run Appium test scripts.

2) *Appium server-1*: a Windows-based mini-PC, running Appium server software for Android devices. Currently it runs two instances of Appium, thus allowing to perform testing on two Android devices simultaneously.

3) *Appium server-2*: a Mac mini computer, running Appium server software for iOS devices. Running iOS tests on the devices connected to non-macOS machines is impossible.

4) *Testing devices*: Five Android devices by five different vendors, connected to the first Appium server, and three different iOS devices, connected to the second Appium server.

5) *USB hubs*: testing devices are connected with computers via *Plugable* USB hubs that support simultaneous data transfer and charging with charging rate up to 1.4 A depending on the device.

6) *Device stand*: all mobile devices are placed on the *FlePow* 10-port charging station dock, which is used simply as a convenient shelf; its charging capabilities are not used.

Project-specific test scripts are managed by the in-house developed coordination system that monitors project changes in *TeamCity*, runs tests for the fresh builds, and generates HTML reports. Both Appium servers run Appium v.1.6.5.

Software developers evaluate a number of factors when choosing a testing framework. Among the most commonly mentioned criteria, the following can be cited [4]: 1) functionality – the sheer number of actions supported by the framework; 2) portability – the ability to test apps under different platforms; 3) language bindings – support of test scripts written in different programming languages; and 4) non-intrusiveness – the ability to test user-end versions of the apps rather than additionally instrumented for tests.

Sometimes test engineers also take into account such factors as execution speed, simplicity of setup and operation, and reliability of the framework [10]. Since our system is based on Appium, it shares all the strengths and weaknesses of this platform. Our main emphasis is on extending existing Appium functionality for integration with a build machine, parallel multi-device test execution, and fine-tunable customized test scenarios, required by the developers.

Summary of Achievements

1. Mobile testing farm architecture.
2. Working prototype of a distributed mobile testing farm that supports Windows, Android, and iOS devices.
3. Appium extensions for handling app distribution across testing devices, load balancing and additional types of UI interaction, not supported in the current Appium implementation.
4. Practical approach for integrating automated smoke testing into the process of cross-platform mobile software development as an element of the continuous integration pipeline.
5. Pattern matching-based technique for handling non-native GUI elements in mobile apps.

Assessment and Lessons Learned

We used the above described mobile farm implementation in the extensive testing process in the project "World of Tennis: Roaring '20s" [4]. This is a mobile game being developed with *Unity* (Figure 4 shows a sample game scene screenshot containing a number of hand-drawn UI elements). This project illustrates well many above mentioned concerns of resource and time consuming GUI testing.

Some months of experiments taught us a few useful facts about mobile farms:

- A mobile testing framework helps to identify the specific target platforms in advance. The most recent version of Appium supports Android 4.2+ and iOS 9.3+. Older operating systems are also supported, but they require different setup and significant changes in testing scripts. Old Apple devices with non-retina screens also have to be handled differently from modern iPhones and iPads.
- Mobile devices discharge while testing. One might think that it is enough to plug a device into a computer or a USB hub to keep the level of battery at an acceptable level. Unfortunately, typical USB charging rates in this case are inadequate: the devices will eventually discharge. Even powered USB hubs can be insufficient; one needs a hub specifically designed for simultaneous charge and data transfer. Even in this case specific devices will refuse to charge (such as Samsung Galaxy Tab E that needs a wall charger) or will charge very slowly (such as iPad 3).
- While Appium is considered a mature project with a significant user base, there are still numerous issues that can lead to hangs, crashes, and in general unreliable test execution. However, there is definitely a progress in this project, and we have seen some issues fixed very recently.
- Each device has its own quirks. There are issues related to a particular version of the operating system, firmware, or even default onscreen keyboard. For example, one of our devices had random crashes until we installed CyanogenMod. Another device reported the lack of free space after several dozens of install/uninstall cycles of the application under testing. A mere reboot could help, but the ultimate solution was to install an alternative Android version.
- In addition to the physical infrastructure some intelligent algorithms can be used for post-processing of test report data. For example, on the base of battery drain statistics, we can make necessary adjustments in running tests on particular devices.

[4] http://worldoftennis.com/

Conclusion

As noted in [15], for mobile software running on multiple differently configured devices in different execution contexts, continuous testing is an essential component of the development process to ensure the quality of mobile applications.

The reason why companies charge so much for mobile testing frameworks is that it is not easy to build them. The main goals of this project are to define a methodology, to build a working system, to design a set of sample applications using this testing framework, and to collect some evaluation results of testing the professional software products.

Creating a mobile testing farm is not always better than renting the alternative cloud facilities. However, it is particularly important that our approach is not to implement only smoke testing, but to make it a mandatory stage of a continuous integration pipeline, similar to unit testing and automated builds.

Our primary experiments show that the procedures for testing automation of the non-native GUI based-applications require the combined use of several technologies including traditional automated unit tests, functional testing frameworks, and even image recognition, so we believe that the proposed approach provides a feasible solution for everyday automated smoke testing and can be considered as a possible extension of tools and methods for mobile software analysis and verification automation.

Pragmatically, our current proof-of-concept implementation helped us in resolving some problems of organizing extensive testing of the software components developed for the ongoing project of mobile game. The prototype allowed us to arrange a big number of experiments. The outcome of these experiments is twofold. First, we got an infrastructure to support the concrete project and to save time and resources required for testing. Second, we got a solution which demonstrated its applicability to many practical situations where our approach can be helpful, including developing mobile applications that are UI intensive, as well as hybrid applications [8].

As a future work, we expect to integrate all the parts (smoke test automation, continuous integration pipeline, image recognition, recommendations on choosing the appropriate test servers and hardware components) and to create a distributed infrastructure that would allow users to use facilities of users' own computers and connected devices as a part of the whole testing framework.

Acknowledgment

This work is partially supported by the grant 17K00509 of Japan Society for the Promotion of Science (JSPS) and by the University of Aizu competitive research grant P-26.

REFERENCES

1. Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A gui crawling-based technique for android mobile application testing. In *Software testing, verification and validation workshops (icstw), 2011 ieee fourth international conference on*. IEEE, 252–261.
2. Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, and Bryan Robbins. 2013. Testing android mobile applications: Challenges, strategies, and approaches. In *Advances in Computers*. Vol. 89. Elsevier, 1–52.
3. Farnaz Behrang and Alessandro Orso. 2018. Automated test migration for mobile apps. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. ACM, 384–385.
4. Avni Gupta. 2017. Mobile Automation: Why We Chose Appium. (January 2017). `https://www.axelerant.com/resources/team-blog/mobile-automation-why-we-chose-appium` Accessed: Jun 22, 2018.
5. Reyhaneh Jabbarvand and Sam Malek. 2017. Advancing energy testing of mobile applications. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 491–492.
6. Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real challenges in mobile app development. In *Empirical Software Engineering and*

Measurement, 2013 ACM/IEEE International Symposium on. IEEE, 15–24.

7. Taeyeon Ki, Alexander Simeonov, Chang Min Park, Karthik Dantu, Steven Y Ko, and Lukasz Ziarek. 2017. Fully Automated UI Testing System for Large-scale Android Apps Using Multiple Devices. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 185–185.
8. Dominik Kipar and others. 2014. Test automation for mobile hybrid applications: using the example of the BILD App for Android and iOS. (2014).
9. Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*. IEEE, 1–10.
10. Tomi Lämsä. 2017. Comparison of GUI testing tools for Android applications. (2017). `http://jultika.oulu.fi/files/nbnfioulu-201706142676.pdf`
11. Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 399–410.
12. Manoj Mahalingam. 2014. *Learning Continuous Integration with TeamCity*. Packt Publishing Ltd.
13. Steve McConnell. 1996. Daily build and smoke test. *IEEE software* 13, 4 (1996), 144.
14. Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2017a. Crashscope: A practical tool for automated testing of android applications. In *Software Engineering Companion (ICSE-C), 2017 IEEE/ACM 39th International Conference on*. IEEE, 15–18.
15. K. Moran, M. L. Vásquez, and D. Poshyvanyk. 2017b. Automated GUI Testing of Android Apps: From Research to Practice. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. 505–506. `DOI: http://dx.doi.org/10.1109/ICSE-C.2017.166`
16. Maxim Mozgovoy and Evgeny Pyshkin. 2017. Using Image Recognition for Testing Hand-drawn Graphic User Interfaces. In *11th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2017)*. IARIA, IARIA, 25–28. `https://thinkmind.org/index.php?view=article&articleid=ubicomm_2017_2_20_10083`
17. Maxim Mozgovoy and Evgeny Pyshkin. 2018. Unity Application Testing Automation with Appium and Image Recognition. In *Tools and Methods of Program Analysis*, Vladimir Itsykson, Andre Scedrov, and Victor Zakharov (Eds.). Springer International Publishing, Cham, 139–150.
18. Sergiy Vilkomir. 2018. Multi-device coverage testing of mobile applications. *Software quality journal* 26, 2 (2018), 197–215.
19. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 183–192. `DOI: http://dx.doi.org/10.1145/1622176.1622213`