So You Want to Build a Farm: An Approach to Resource and Time Consuming Testing of Mobile Applications

Evgeny Pyshkin and Maxim Mozgovoy
University of Aizu

Tsuruga, Ikki-Machi, Aizu-Wakamatsu, Fukushima, 965-8580, Japan
Email: {pyshe, mozgovoy}@u-aizu.ac.jp

Abstract—**The focus of this research is on improving a process of resource and time-consuming automated software testing. Particularly, we address the problem of testing mobile applications with rich non-native or hand-drawn graphical user interface, as well as resource-consuming dynamic applications, such as mobile games. We introduce an approach to creating a mobile testing farm, which is relatively easy to build with inexpensive components and open source software. This approach can be useful for supporting a product development cycle for a company on lean budget. It is suitable for a wide range of mobile applications with a high variety of human-computer interaction mechanisms.**

Keywords–Non-native GUI; mobile applications; testing framework; human factors.

I. Introduction

Though there are many existing frameworks supporting software development and testing automation, creating open testing platforms and sharable pragmatic solutions remains one of strategic parts of software quality assurance [1]. For the specific case of mobile testing (including mobile user interface (UI) testing automation), there is a gap between rapid evolution of mobile software and availability of comprehensive automated solutions focusing peculiarities of mobile applications development and testing [2]. One of such particular aspects of mobile software testing is the problem of creating flexible tools that would facilitate running automated tests of large-scale and resource-intensive on mobile applications [3].

One of obvious requirements for automated UI tests is that they should be able to access applications similarly as users do. Particularly, testing graphical UI (GUI) provides a nontrivial case of testing automation for both traditional and mobile applications [4][5]. Existing tools for testing automation (such as Jemmy library [6], Microsoft UI Automation [7], or Android UI Automator [8]) provide features for testing GUI applications in regular cases. They allow accessing programmatically many GUI elements and performing different operations such as pushing a button, scrolling a window, hovering an area, and so on. However, there are specific cases when testing process is time- and resource-consuming. Unlike to traditional applications rewritten to be runnable on mobile devices, applications developed primarily for mobile devices have significant particularities such as connectivity dependency, limitations in available computing resources, battery discharging, specufic GUI based touch screen gestures, rapid evolution and diversity of devices, as well as rapidly evolving new operating systems [9]. Many of these factors are connected and mutually dependent. For example, in mobile games, we might have to run the relatively long-lasting process and collect many screenshots necessary for reproducing the test cases and for making further fine-grain analysis of possible application failures. For arranging such time- and effort-consuming tests, device emulators (being a widely used solution allowing to decrease the testing costs) are often not enough. There are the following reasons:

1) We have to be sure that a program works properly on real devices (it is mentioned above that a wide diversity of mobile devices is one of the significant peculiarities of mobile applications).
2) An emulator could not help in testing applications with intensive CPU and GPU load required for revealing battery drain problems.
3) Test failures might be device-sensitive: a test might successfully pass on one device, while (often unexpectedly) crashing on the another one.
4) Testing on emulators makes difficult to reveal low performance problems.
5) It is hard to model connectivity-sensitive test cases.

In order to decrease testing complexity and save testing time, the developers often use the restricted test suites known as smoke tests, which are useful for some sanity checks: they are aimed at checking whether the whole application works, provides its basic functionality, and operates with user controls properly. Smoke testing is an important element of software deployment process, particularly in case of severe time and cost pressure [10][11].

Simple test scripts can check whether a program works in general, but also they can reveal many potential problems like lack of interaction with a server backend, incorrect processing of user requests, failures in user interactions with UI (probably containing non-standard hand-drawn elements), etc.

In the domain of mobile development including (with respect to the scope of our particular interests) virtualized environments, mobile games, learning environments, etc., arranging smoke tests is far from being a trivial problem. Complex testing scenarios might require the use of specialized smoke testing frameworks. As it has been mentioned above, mobile applications often do not have a platform-native GUI, but a set of hand-drawn elements built without using standard GUI libraries. One relevant project is the recently launched Unity-based mobile game “World of Tennis: Roaring ’20s” [12], which is a good example illustrating the complexity of testing gaming applications running primarily on mobile devices [13].

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

Apart from mobile games, there are more application types that may be built with non-native UI components. For example, map-based travel applications often use GUI elements, which are not ordinary user controls supported by standard testing frameworks, but specially designed components integrated with an electronic map [14]. Hand-drawn GUI is also widely used in educational mobile applications, for example, in language learning applications with non-standard UI elements for representing language grammar structures [15].

The remaining paper is organized as follows. In Section II we describe the application context for our approach and briefly examine recent works in the domain of mobile software testing automation. In Section III we introduce our approach and discuss its current implementation, as well as the lessons learned from using this approach in a mobile development project. In Conclusion we summarize our current contribution and briefly describe the planned future steps.

II. Related Work

The standard approach to mobile application testing is to connect mobile devices to a "test server" running some special software, in order to execute test scripts on a remote machine (as shown in Figure 1).

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

For the server-side software, one can rely on existing solutions, such as Appium [16] and Calabash [17]: a test script is usually a set of instructions containing such activities as waiting, tapping screen location, asserting that an expected UI elements appears on the screen, pressing the button, tapping a certain GUI element within some screen area, etc.

In such test scripts for native GUI applications, user controls can be accessed programmatically: normally, this capability is supported by an operating system. However, in a case of applications that do not rely on the natively rendered GUI components of an underlying operating system and do not use standardized GUI libraries, this approach does not work. Various sources [18][19], including our own works [5][20], suggest to use pattern recognition methods to identify GUI elements on the screen. In a sense, human intelligence (e. g., constructing smoke test scripts) meets the algorithms of machine intelligence (e. g., using image recognition to find GUI elements on the screen).

This technique requires experimenting with the settings of pattern matching and image transformation algorithms (provided, e. g., by OpenCV library), and in general, slows down the testing process. It might be difficult to estimate "typical" duration of a test, since simple smoke tests can reveal the absence of crashes within seconds, while *stress tests*, designed to check the stability of an application in a prolonged time interval, can take hours. Furthermore, ideally every new build should be tested on a variety of mobile devices.

A common way to run automated tests on a selection of real mobile devices is to use cloud mobile farm providers (such as *Amazon Web Services*, or *Bitbar*). Such cloud farms have many advantages: they support many different mobile devices; they can be easily set up; they do not require specific client side equipment. However, there are significant drawbacks as well: most providers still do not support an adequate variety of devices or a selection of devices that can be particularly interesting for the mobile software developers. The testing cost can be quite high for a small team, freelance developer or startup company.

III. Our Approach and Current Implementation

A possible alternative to cloud mobile farms (which are easy to deploy, but expensive and often insufficient) is to build own farms that can be configured to fit exact developers' requirements and specific purposes of the testing process. The expected functionality of such farms includes support for the follwing processes: 1) getting builds from a build machine (such as *TeamCity* [21]); 2) running all tests on all connected devices; 3) generating HTML reports containing the application action logs and screenshots; 4) sending the reports and related data to the subscribed users.

A. Prototype mobile farm

Figure 2 shows the organization of the current prototype we use.

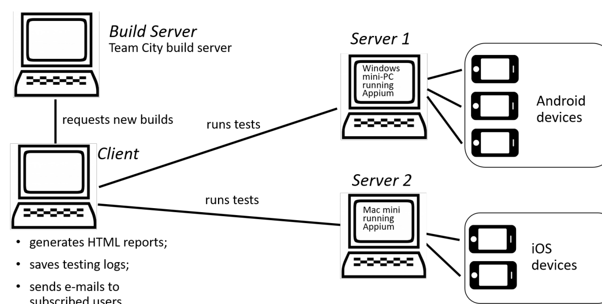


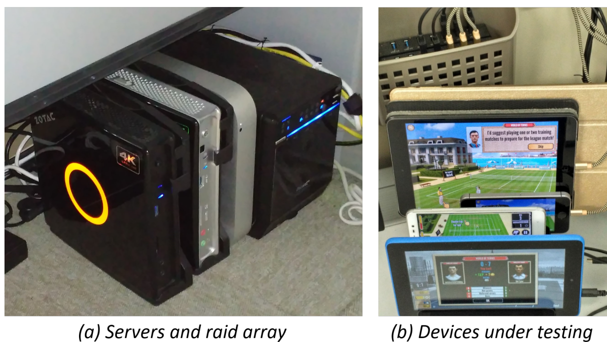
Figure 2. Mobile farm organization: major components.

In our implementation, *Client* testing server is a Windows-based mini-PC, used to run Appium test scripts. There are two servers supporting tests on connected devices: *Server 1* is a Windows-based mini-PC, running Appium server software for Android devices mostly (but Windows devices can also be connected to this server); *Server 2* is a Mac mini computer, running Appium server software for iOS devices mostly. The second server is required, since it is not possible to run iOS tests on the devices connected to non-macOS machines.

Testing devices (where the mobile software under testing is running) are connected with the computers via *Plugable* USB hubs that support simultaneous data transfer and charging with charging rate up to $1.4A$ depending on the device. Figure 3 demonstrates a working mobile farm prototype with three servers, a RAID array based storage and a variety of connected mobile devices under tests.

Though the current implementation is a relatively simple compact solution, it helped us to analyze many difficulties

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2



(a) Servers and raid array (b) Devices under testing

Figure 3. A prototype mobile farm.

that a quality assurance engineer might face while building a reliable and convenient infrastructure for automated testing of mobile applications on real devices, including (but not limited to) the following problems:

- How to support a representative variety of devices with regards to their operating and eco-systems (e. g., solutions which are perfectly deployed for Android devices might not be working for iOS-based devices).
- How to find appropriate hardware for connecting a reasonable number of devices (with respect to battery draining, difference in charging/connection interfaces, charging rate and time, capabilities to charge and transfer data at the same time, etc.).

B. Case Study: Using Appium and Image Recognition for Testing Non-Native GUI of Mobile Applications

Appium is a test automation framework designed to assist functional testing of compiled native (iOS, Android or Windows) and hybrid applications [22]. By accessing an application from Appium scripts, we can simulate different user interactions. Appium is responsible for the following activities: 1) receiving connections from a client; 2) listening for commands; 3) passing received commands to the application under testing (the application is run on the same machine as Appium or on a USB-connected mobile device); 4) sending responses from the application back to the client.

Thus, Appium just provides a client-server layer that has testing script on the client side and application on the server side. All the work of test scheduling, interaction with TeamCity, storing and retrieving test logs and other things has to be done in our own code, so there is much work for test engineers.

According to the above described process, remote clients connect to Appium servers and run test scripts that send commands for execution. Native application GUI elements can be accessed using a specialized API. However, in order to access non-native or hand-drawn GUI elements, one needs to recognize them on the screen first. From one's first look, identifying objects of interest on the screen (such as non-native GUI controls or game characters) can be reduced to the task of perfect matching of a requested bitmap image inside a screenshot. However, there is a reasonable number of factors making such a naïve approach insufficient for reliable GUI element recognition and thus requiring approximate matching:

- Onscreen objects may be rendered differently (because of GPUs or rendering quality settings.
- Screens vary in dimensions, thus, game scenes might have to be rescaled before rendering. Such a transformation might cause significant distortions.
- Game designers might slightly change the UI elements (fonts, colors, background, etc.).
- Onscreen objects might interfere with complex background or with other objects.
- Many interactions are performed with non-GUI onscreen game objects (such as game characters).

The idea of using OpenCV-supported approximate image matching in Appium is discussed in several tutorials [23][24]. We rely on OpenCV function *matchTemplate()* called with the parameter *TM_CCOEFF_NORMED*. This parameter defines the pattern matching algorithm used by *matchTemplate()*. The pattern matching function allows us to get image similarity coefficients and analyze testing results from the viewpoint of UI elements recognition quality. Unfortunately, *matchTemplate()* function is unable to match scaled patterns. Since a mobile application may run on devices with different screen sizes, we have to scale the screenshots to match the dimensions of the original screen used to record graphical patterns.

Image processing functions (including operations with a large number of screenshots) slows down the testing procedures significantly and makes the whole testing process resource- and time-consuming.

C. Assessment and Lessons Learned

Our current experience to use the suggested approach is based on two prototype farm implementations for testing the large scale software project, which is the above mentioned Unity-based mobile game "World of Tennis: Roaring '20s". Our experiments taught us a number of interesting facts about mobile farms.

Due to very intensive application usage in test runs, mobile devices quickly discharge while testing. Unfortunately, it is not enough to plug a device into a computer or a USB hub to keep the level of battery at an acceptable level: typical USB charging rates are inadequate. Our experiments demonstrated that even powered USB hubs can be insufficient, hence, one might need a hub supporting simultaneous charge and intensive data transfer. However, we realized that even if one uses special powered hubs, there are devices charging very slowly or refusing to charge in such conditions.

Though Appium is a mature project with a significant user base, there are still some unresolved issues that can lead to unreliable test execution. However, we have to admit that there is a visible progress in this project, since many problems (that we faced in the past) have been already fixed.

A device may have its own oddities. While testing, unexpected behavior may be conditioned by a particular version of the operating system or firmware, or even default onscreen keyboard. For example, we tested one device that was randomly crashing until we installed CyanogenMod. Another device reported the lack of available memory space after several

Copyright (c) IARIA, 2018. ISBN: 978-1-61208-668-2

dozens of installation-uninstallation cycles of the application under testing. The problem was resolved by installing an alternative Android version.

IV. Conclusion

In this contribution, we demonstrated that building a mobile testing farm is not a trivial task. We introduced an approach, which includes a process, a working system, and a set of sample applications using this testing infrastructure. Some primary evaluation results of testing professional software products proves the applicability of the suggested method to the practical cases of mobile software testing.

Particularly, our contribution include a mobile testing infrastructure; a working prototype supporting testing of Windows, Android, and iOS devices; Appium extensions (for handling application distribution across a number of connected devices, load balancing and supporting additional types of UI interaction, which were not included to the Appium implementation we used; and pattern matching-based technique for recognition of non-native GUI elements in test scripts. All above mentioned elements of our solution can be considered as parts of continuous integration process.

We do not argue that creating a farm is always better than renting through the alternatives. However, it is important to note that our approach is not only about implementing smoke tests for a particular case: it should be considered as a stage of a continuous integration pipeline, similar to automated unit testing and automated builds.

We believe that the proposed approach provides a practical solution for real world problems of software analysis and verification automation. Our primary experiments show feasibility of the suggested process for testing automation with the combined use of several technologies including traditional automated unit tests, functional testing frameworks, and image recognition algorithms.

As a future work, we expect to create an open source framework for small-scale mobile farms that would allow users to use facilities of users' own computers and connected devices as a part of the whole testing framework. We expect that such an approach will make smoke testing easier to set up, encourage mobile software developers to extend their testing automation practices, and, therefore, improve mobile software quality. As a result of our efforts, paraphrasing on the the famous Glenn Gould's conceptual composition "So you want to write a fugue" [25], we believe to be able to say: "So you want to build a farm – so go ahead and build a farm".

References

[1] D. Amalfitano, A. R. Fasolino, P. Tramontana, and B. Robbins, "Testing android mobile applications: Challenges, strategies, and approaches," in Advances in Computers. Elsevier, 2013, vol. 89, pp. 1–52.

[2] M. Linares-Vásquez, K. Moran, and D. Poshyvanyk, "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, 2017, pp. 399–410.

[3] T. Ki, A. Simeonov, C. M. Park, K. Dantu, S. Y. Ko, and L. Ziarek, "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, 2017, p. 185.

[4] K. Moran, M. L. Vsquez, and D. Poshyvanyk, "Automated gui testing of android apps: From research to practice," in 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), May 2017, pp. 505–506.

[5] M. Mozgovoy and E. Pyshkin, "Unity application testing automation with appium and image recognition," in Tools and Methods of Program Analysis, V. Itsykson, A. Scedrov, and V. Zakharov, Eds. Cham: Springer International Publishing, 2018, pp. 139–150.

[6] "Jemmy library," retrieved: Aug 1, 2018. [Online]. Available: https://jemmy.java.net/

[7] "Ui automation," retrieved: Aug 1, 2018. [Online]. Available: https://docs.microsoft.com/en-us/windows/desktop/WinAuto/entry-uiauto-win32

[8] "Automate user interface tests," retrieved: Aug 1, 2018. [Online]. Available: https://developer.android.com/training/testing/ui-testing/index.html

[9] H. Muccini, A. Di Francesco, and P. Esposito, "Software testing of mobile applications: Challenges and future research directions," in Proceedings of the 7th International Workshop on Automation of Software Test. IEEE Press, 2012, pp. 29–35.

[10] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation (Adobe Reader). Pearson Education, 2010.

[11] G. Mustafa, A. A. Shah, K. H. Asif, and A. Ali, "A strategy for testing of web based software," Information Technology Journal, vol. 6, no. 1, 2007, pp. 74–81.

[12] "World of tennis: Roaring 20's," retrieved: Aug 1, 2018. [Online]. Available: http://worldoftennis.com/

[13] K. Haller, "Mobile testing," ACM SIGSOFT Software Engineering Notes, vol. 38, no. 6, 2013, pp. 1–8.

[14] E. Pyshkin and M. Pyshkin, "Towards better requirement definition for multimedia travel guiding applications," in Computational Intelligence (SSCI), 2016 IEEE Symposium Series on. IEEE, 2016, pp. 1–7.

[15] M. Purgina, M. Mozgovoy, and V. Klyuev, "Developing a mobile system for natural language grammar acquisition," in Dependable, Autonomic and Secure Computing, 14th Intl Conf on Pervasive Intelligence and Computing, 2nd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), 2016 IEEE 14th Intl C. IEEE, 2016, pp. 322–325.

[16] "Appium: Automation for apps," retrieved: Aug 1, 2018. [Online]. Available: http://appium.io

[17] "Calabash: Automated acceptance testing for mobile apps," retrieved: Aug 1, 2018. [Online]. Available: http://calaba.sh

[18] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using gui screenshots for search and automation," in Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, ser. UIST '09. New York, NY, USA: ACM, 2009, pp. 183–192.

[19] T.-H. Chang, T. Yeh, and R. C. Miller, "Gui testing using computer vision," in Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, ser. CHI '10. New York, NY, USA: ACM, 2010, pp. 1535–1544.

[20] M. Mozgovoy and E. Pyshkin, "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, Nov 2017, pp. 25–28.

[21] M. Mahalingam, Learning Continuous Integration with TeamCity. Packt Publishing Ltd, 2014.

[22] M. Hans, Appium Essentials. PACKT, 2015, retrieved: Aug 1, 2018. [Online]. Available: https://www.packtpub.com/application-development/appium-essentials/

[23] S. Kazmierczak, "Appium with image recognition," February 2016, retrieved: Aug 1, 2018. [Online]. Available: https://medium.com/@SimonKaz/appium-with-image-recognition-17a92abaa23d\#.oez2f6hnh

[24] V.-V. Helppi, "Using opencv and akaze for mobile app and game testing," January 2016, retrieved: Aug 1, 2018. [Online]. Available: http://bitbar.com/using-opencv-and-akaze-for-mobile-app-and-game-testing

[25] G. Gould, So you want to write a fugue? G. Schirmer, 1964.