Pragmatic Approach to Automated Testing of Mobile Applications with Non-Native Graphic User Interface

Maxim Mozgovoy, Evgeny Pyshkin
School of Computer Science and Engineering, Division of Information Systems
University of Aizu
Aizu-Wakamatsu, Japan
E-mail:{mozgovoy, pyshe}@u-aizu.ac.jp

***Abstract*—This article addresses the problem of automated smoke testing for mobile applications with hand-drawn non-native graphic user interface (GUI) within the context of continuous integration pipeline. In such applications the traditional approach to define and test situations triggered by appearance of certain GUI elements accessed programmatically does not work, so we need to apply image recognition and pattern matching algorithms to testing both the application interface and its major functional features. We introduce one example, which is a Unity-based mobile game "World of Tennis: Roaring '20s". Our idea is to classify GUI elements (including buttons, game control elements, static and movable objects) with respect to their appearance in different type of game scenes, as well as to find pattern recognition methods providing the best similarity values to increase GUI element recognition quality and, therefore, to suggest a reliable support for test script writers.**

Keywords-software testing; GUI; image recognition; pattern matching; similarity; mobile game; continuous integration.

I. INTRODUCTION

This post-conference article is an extended revision of our paper presented at the UBICOMM-2017 conference [1].

Automated testing is an integral element of software development pipeline, frequently discussed in literature. Though many specialists agree that automated tests could not completely substitute careful manual testing [2], the combination of automated tests with manual quality assurance procedures is one of the central tenets of established software development methodologies, such as test-driven development [3] and behavior-driven development [4]. In addition, testing frameworks assure better communication between developers and customers: they allow developers rediscovering the customer context better and can be used to improve acceptance testing practices and procedures, which, in turn, are essential parts of iterative software development processes [5][6][7].

In practice, however, maintaining an adequate set of tests can be a challenging and time-consuming task: surveys show that the majority of professional developers are not satisfied with their current testing suites or do no automatic testing at all, complaining that the tests are difficult to write and maintain [8]. A pragmatic approach to testing suggests prioritizing testing strategies, and keeping at least the most useful tests well maintained. Some authors suggest giving the priority to smoke tests that check basic functions of the whole software system [9]. Let us recall that, according to [10], smoke tests represent a subset of all defined/planned test cases that cover the main functionality of a component or system. Their goal is to ascertain that the most crucial functions of a program work correctly, without checking more fine-grained aspects of software's functional specification. By definition, smoke tests do not cover the code under testing completely.

Humble and Farley believe that smoke tests (considered as elements of software deployment process) are probably the most important tests to write [11]. In turn, Mustafa et al. advise to "stick to smoke testing" in case of severe time and cost pressure [12]; MSDN documentation calls smoke testing "the most cost effective method for identifying and fixing defects in software" after code reviews [13].

Thus, smoke tests are aimed at performing some basic checkups: whether the program runs at all, is it able to open required windows, does it react properly to user input, etc. Automated user interface (UI) smoke tests should be able to access applications in the same way as users do, so they need to manipulate application's user interface. Specifically, testing graphical UI (GUI) provides an interesting and nontrivial case of testing automation [14].

While a smoke test can be as simple as "launching the application and checking to make sure that the main screen comes up with the expected content" [11], it can also evolve into a complex suite of tests checking core application functionality. Complex testing scenarios may require the use of specialized smoke testing frameworks. One interesting and widespread example of such scenario is mobile application testing automation. Mobile apps are hard to test due to several factors:

1. All supported platforms and a wide range of devices should be used in tests;
2. The apps should be tested on real devices rather than on emulators/simulators;
3. The tests should reveal both bugs and problems such as battery drain and low performance;
4. Non-native (hand-drawn) GUI requires specialized handling.

The idea of hiding platform-specific UI automation frameworks behind a universal interface was recently implemented in the tools such as Appium [15] and Calabash [16]. However, these frameworks can only interact with user interfaces based on native GUI components of an

underlying operating system (such as widgets exposed by standardized GUI libraries like Qt or WinForms). Therefore, additional efforts are required to recognize and interact with non-native GUI elements, referenced in test scripts. For example, cross-platform mobile games often rely on such hand-drawn GUI elements. These widgets might look slightly different on different devices with different resolutions; their onscreen positions often are not fixed. Many interactions also have to be performed with "active" game objects, such as buildings, game characters, map elements, etc. Technically, an operating system "sees" non-native GUI elements as graphical primitives drawn on a canvas, and, therefore, cannot manipulate them via standard object-oriented API.

Consequently, a UI automation framework also recognizes the main window of a non-native GUI based application as a plain graphical image containing no UI elements. Similar problems might appear in other multimedia projects, such as text recognition applications (where texts are represented as images), applications based on interactive electronic maps, etc.

Figure 1. Actual screen of *World of Tennis: Roaring '20s*.

The example we use in this article is the mobile game project "World of Tennis: Roaring '20s", where we are involved in [17] (see Figure 1). This game is made with Unity, and its GUI is represented with hand-drawn components. This setup makes difficult to develop standard automated GUI tests and basic functional smoke tests, since all screen elements are in fact plain graphical images that we cannot easily access programmatically in test scripts [18]. Hence, testing automation requires integrated use of image recognition and pattern matching capabilities.

The basic goal of this paper is to show how standard pattern recognition tools can be used as a universal aid for GUI testing (primarily, for applications with non-native user interface). We list a number of practical challenges associated with this approach, and discuss how one can fine-tune the settings of the pattern recognition procedure to ensure smooth operation in a variety of scenarios.

The paper has the following structure. In Section II, we describe our approach within the context of existing research in the area. In Section III, we examine a number of problems to be resolved while implementing test scripts using pattern recognition methods. Section IV describes how our experiments were organized. Section V introduces a discussion on applicability of the suggested approach for wider range of mobile applications. In Section VI, we briefly summarize the current state of this project and introduce the tasks for future work.

II. Approach and Related Work

In our previous work, we described the process of deployment of smoke testing infrastructure using Appium as a testing automation framework and continuous integration setup using TeamCity as a build server [14] (see Figure 2). We also demonstrated that identifying objects of interest on the screen, such as GUI elements or game characters, could not be completely reduced to the task of perfect matching of a bitmap image inside a screenshot [1]. It happens due to several reasons:

- Onscreen objects may be rendered differently with different GPUs or rendering quality settings;
- Screens vary in dimensions, so patterns might need scaling;
- Onscreen objects often intersect with each other, so one object might partially hide another object.

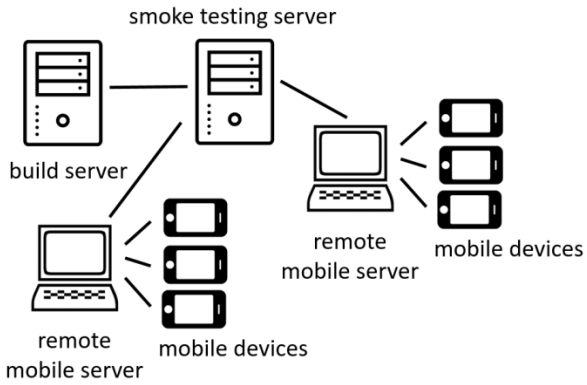

Figure 2. Mobile application testing infrastructure.

Thus, the most straightforward way to recognize such elements is to rely on approximate pattern matching. There are several tutorials where an idea of using image matching in creating test scripts is discussed [19][20]. OpenCV library [21] provides a number of methods for pattern recognition and can serve as a typical tool used for searching and finding the occurrences of the given pattern in a larger image. Basic OpenCV pattern matching methods can be accessed using *matchTemplate()* function with a parameter defining a specific method among the variety of supported pattern matching methods [22][23]:

1. CV_TM_SQDIFF: square difference matching minimizing the squared difference between the pattern and the image area;
2. CV_TM_SQDIFF_NORMED: normalized version of the square difference matching (normalized methods are typically used when the effects of lighting difference between a pattern and an image should be reduced [24]);

3. CV_TM_CCORR: correlation matching method multiplicatively matching a template against the image and then maximizing the matched area;
4. CV_TM_CCORR_NORMED: normalized version of the correlation matching method;
5. CV_TM_CCOEFF: correlation coefficient matching method that matches a template against the image relative to their means and generates a matching score ranging from –1 (complete mismatch) to 1 (perfect match); and
6. CV_TM_CCOEFF_NORMED: normalized version of the correlation coefficient matching method.

As we know from different sources (such as [24]), the *matchTemplate()* function slides a template over the given area and computes similarity value in a range of [0..1] for each pixel location, thus maximizing pattern matching similarity. The function yields the best value as the final recognition similarity, so we are able to analyze the result from the viewpoint of GUI elements recognition quality.

An automated test consists of the following steps:

- Take a game screenshot.
- Detect the presence of a certain GUI element using image recognition.
- React properly.
- Check the expected application behavior or program state.
- Repeat the process.

Hereafter, we describe the core function of the automated smoke tests we developed for the "World of Tennis: Roaring '20s" mobile game. The Python test script presented below is responsible for checking application initialization and several actions performed in the beginning of the game. Initial game run requires several core subsystems to work properly. Thus, successful first run is more than just a smoke test; it is a good indicator of a stable game build. In general, the test script follows the same routine as described in the list above. In the current automated testing framework implementation, we match GUI elements with OpenCV *matchTemplate()* function called with a parameter TM_CCOEFF_NORMED.

Since the game may run on devices with different screen sizes, we scale the screenshots to match the dimensions of the original screen used to record graphical patterns. In Listing 1 we present a Python code for the *findByImage()* function. This function tries to find a template GUI element pattern *templateImg* in the screenshot *img* (highlighted line), and returns the similarity score we got from the corresponding OpenCV algorithm paired with the coordinates of the matched area center.

LISTING 1. FINDING A TEMPLATE WITH TM_CCOEFF_NORMED METHOD

```
import cv2 # OpenCV
import imutils

def findByImage(img, templateImg):
  img_h, img_w = img.shape[0:2]          # image dimensions
  template = cv2.imread(templateImg, 1) # read template
  h, w = template.shape[0:2]


  # rescale the template for the target device's screen
  # (here we assume that template image was taken
  # at 1920x1080 resolution)
  factor = float(img_w) / 1920
  template = imutils.resize(template,
      width = int(w * factor), inter = cv2.INTER_CUBIC)
  h, w = template.shape[0:2]

  res = cv2.matchTemplate(img, template,
                          cv2.TM_CCOEFF_NORMED)
  (_, maxVal, _, maxLoc) = cv2.minMaxLoc(res)
  result = ((maxLoc[0] + (w / 2),
            maxLoc[1] + (h / 2)), maxVal)
  return result
```

The function *waitFor()* (see Listing 2) waits for the given image or a list of images to appear on the screen (while the application is running). In case of failed recognition, an exception is thrown. This function uses the above presented *findByImage()* in order to recognize the GUI element. The similarity score returned by *findByImage()* function is then compared to the globally defined threshold. After that, it is possible to interact with a GUI element.

LISTING 2. WAITING FOR A GUI ELEMENT IMAGE ON THE SCREEN

```
def waitFor(driver, tries, interval, imagefiles):
  # convert a single image into a one-element list
  if type(imagefiles) is not list:
    imagefiles = [imagefiles]

  for i in range(tries):
    img = takeScreenshot(driver) # a wrapper for Appium
                                 # screenshot function
    for imagefile in imagefiles:
      (loc, simRatio) = findByImage(img, imagefile)
      if simRatio > Config.PicFoundThreshold:
        return (loc, simRatio)
    time.sleep(interval)

  raise RuntimeError("GUI element not found. Tried: " +
                                  str(imagefiles) + ".")
```

The fragment of function *testFirstRun()* (see Listing 3) presents the first part of the smoke test responsible for checking some initial actions during the game run:

1. Test whether the user name is properly entered and accepted by the application.
2. Test whether the screen with players appears after pushing OK button.
3. Test a possibility to go to the configuration screen for the selected player.
4. Test whether the tabs work properly in the configuration window.
5. Test how the selected configuration is applied after tapping Apply button.

LISTING 3. FIRST RUN SMOKE TEST (FRAGMENT)

```
# Test first run (test script fragment)
def testFirstRun(driver):
  print("Testing the first run")

  print("wait for the name input box, and tap it")
  tap(driver, waitFor(driver, 9, 3, 'name_input_box.png'))

  print("type the user name")
  # this function generates and types a random user name
  inputName(driver)

  print("tap OK button")
  tap(driver, waitFor(driver, 3, 3, 'ok_button.png'))
  time.sleep(4)

  print("wait for the player circle, and tap it")
  tap(driver, waitFor(driver, 10, 3,
          ['player_green_circle.png', 'finger.png']))
```

```
print("tap 'Character'")
tap(driver, waitFor(driver, 3, 3,
      ['character_tab_1.png', 'character_tab_2.png']))

print("tap on the player")
tap(driver, waitFor(driver, 3, 3,
                    'player_select_region.png'))

print("tap 'Apply'")
tap(driver, waitFor(driver, 3, 3,
        ['apply_button_1.png', 'apply_button_2.png']))

# ...
```

In *testFirstRun()* script we used a number of GUI elements (Figure 3) that we are trying to recognize on the screens presented in Figure 4.

Figure 3. GUI elements used in the first run test script.

Figure 4. Game screenshots used in the first run test script.

III. Towards Better GUI Element Recognition Reliability

As it follows from the observations mentioned in Section II, an important problem is to find optimal parameters of image recognition algorithms maximizing GUI elements recognition reliability. Such an approach would decrease the number of automated tests that might fail, not because of the software bugs, but due to the UI elements recognition defects.

Figure 5. Club view template examples that can be used for checking screen orientation.

Both types of recognition errors (false negatives and false positives) actually caused problems in our experiments. Let us provide some examples.

a) The first step in all our tests is to detect whether the device screen is rotated upside down (for the sake of brevity, this procedure was omitted in Section II). To do it, we try to match certain elements of the initial "club view" scene against a number of regular and flipped patterns (see Figure 5). Our experience shows that this step often provides false positives, as both types of patterns can be found by OpenCV algorothms.
b) When the game designers slightly change the buttons (in order to beautify them, make them slightly larger or smaller, change fonts, colors, etc.), our tests stop recognizing them. It may happen even with very simple elements like buttons shown in Figure 6.
c) When buttons have two states (enabled/disabled), visually shown with different colors, the tests often fail to recognize them accurately.
d) Additional elements shown on or next to GUI elements (such as checkboxes or numbers) might prevent the tests to recognize them properly.

The challenge is to make sure that we still can match changing GUI elements, while being able to distinguish them.

Figure 6. Static UI controls: buttons, tabs, check boxes, static images, etc.

There are also numerous moving objects on the screen. Suppose the test script needs to press on the character's model in the pictures shown in Figure 7.

Figure 7. Moving objects: the object view changes and the surrounding area might change as well.

An animated head might make a perfect match difficult not only because of changes in object view itself, but also because of possible changes in the adjacent screen area (e.g., an airplane appeared in the sky, in our case). Hence, it might be required to work with a set of different images related to the same UI element and to perform a matching process for all of them. We have to consider a possibility to work with a larger region providing necessary context to avoid false positive recognition results.

Our hypothesis is that experimenting with different pattern matching algorithms will allow us to provide a number of recommendations for test script developers. These recommendations will provide hints on what are the better algorithms to use in certain test contexts.

IV. Experiments with Different UI Elements

A large variety of UI components designed for the "World of Tennis: Roaring '20's" allows us to classify them in a number of classes including the following UI types:

- UI widgets: buttons, edit boxes, tabs, etc.
- Static images: player portraits, court fragments, popup message boxes.
- Dynamic objects: moving player figures, onscreen hints.

As discussed in Section II, for the first implementation of test scripts, we used OpenCV *matchTemplate()* function and a number of built-in pattern matching methods. After experimenting with a number of test scripts, we realized that pattern matching reliability significantly depends on a recognition task. For example, simple button-like GUI elements (buttons, menus, tabs) can usually be recognized with high degree of similarity (0.90..0.98), according to OpenCV reports. Similarity score decreases to (0.63..0.65) for certain elements interfering with the background like menu items placed against the sky with moving clouds. This makes perfect template matching impossible in principle. Lower similarity values might occur even for the objects that are not graphically complex, but contain patterns distorted during rescaling (as Figure 8 shows).

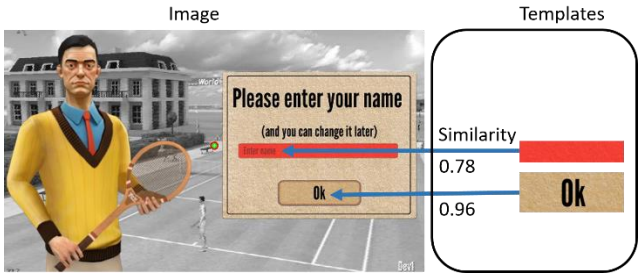

Figure 8. Template matching similarity varies for different UI elements.

Even in the simplest cases, the similarity scores might differ depending on the device where the code is running. As Table I illustrates, the scores for OK button range from 0.94 to 0.99 for different devices even if there is no any recognition complication such as "bad" background, surrounding or changing objects, etc. (apparently due to different screen resolutions and screen image scaling distortions).

Table I lists a selection of devices used in this experiment, their screen characteristics and reported similarity scores [1].

TABLE I. EXPERIMENTING WITH OK BUTTON USING TM_CCOEFF_NORMED ALGORITHM

Case	Description of the test case			
	Device	*Screen*	*Tap size*	*Similarity*
a	Xiaomi Redmi Note 3 Pro	1920x1080	1920x1080	0.99
b	iPad Air	2046x1536	1024x768	0.95
c	Doogee X5 Max Pro	1280x720	1280x720	0.94

In certain tests, the system reports the presence of UI elements that are actually not shown on the screen. Such false positive cases typically happen if some similar-looking graphical elements are confused with each other, especially when they are surrounded by moving objects or complex background. As noted in Section III, a possible way to struggle with such cases is to try to match larger regions in order to include more context into the pattern. For example, in our tennis game application, the Skip button is always placed next to a checkbox, so we can try to match the whole button/checkbox region.

One more approach that can be used to decrease interference with the complex background is to apply image transformations for both the grabbed screenshot and the pattern [25]. In particular, we were experimenting with a number of edge detection filters including Laplace algorithm and Canny edge detection algorithm [26]. Samples of image transformations are shown in Figure 9. We used the GNU Image Manipulation Program [27] for Laplace edge detection and the online tool "Imaging Web Demonstrations" [28] for Canny edge detection algorithm.

Figure 9. Samples of filtered templates.

Preliminary experiments show that the use of transformed images does not significantly improve recognition of UI elements when an element is present on the screen. However, such transformations may be helpful to avoid false positives.

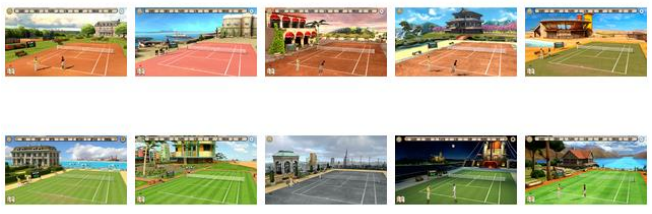

Figure 10. Scenes used for experimenting with false postive cases.

In Table II, we summarized our experiments with the TM_COEFF_NORMED algorithm, as described in details in our separate paper [25]. For the experiments, we used 10 game scene screenshots (Figure 10) and 11 test fragments. 10 fragments ware taken from the above-mentioned screenshots (hence, each fragment exists exactly in one scene). We also used one pattern fragment that does not belong to any of 10 screenshots. Thus, from 110 possible combinations only 10 correspond to true positive cases.

As we can see from the results listed in Table II, edge filtering does not affect true positive cases. However, the number of false positive cases with substantially high scores significantly decreases if the pattern matching method receives filtered images as input.

TABLE II. RECOGNIZING A SELECTION OF SAMPLE FRAGMENTS WITH TM_CCOEFF_NORMED METHOD IN PLAIN CASE AND WITH EDGE DETECTION FILTERS

	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
True positive (normal images)	0	0	0	0	0	0	1	0	1	8
True positive (Canny)	0	0	0	0	0	0	1	0	1	0
False positive (normal images)	0	0	14	19	32	22	13	0	0	0
False positive (Canny)	58	37	5	0	0	0	0	0	0	0

Figure 11 provides visual demonstration of the fact that the scores for false positive cases are shifting to the lower ranges after applying an edge detection filter to both scene and template images.

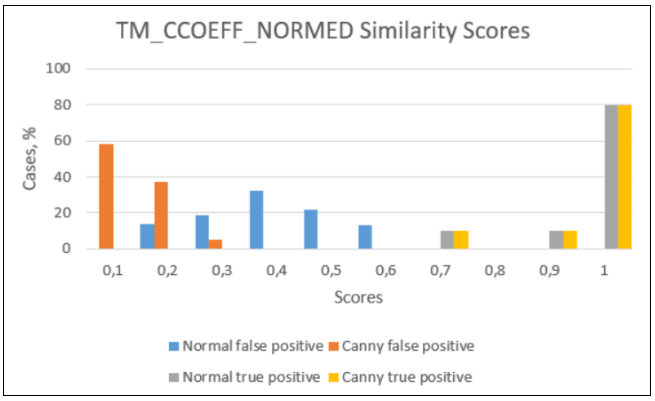

Figure 11. Struggling with false positive cases using Canny edge filtering.

In principle, there is no necessity to strive for the highest similarity scores in the situations where the GUI elements are present on the screen. Our primary goal is to minimize the number of both false positive and false negative errors: what we need is an algorithm that separates the scenes *with* and *without* target GUI elements reliably.

We have to run pattern-matching algorithms for significantly different usage contexts: in player settings window, club selection window, game selection window,

ongoing game window, etc. We expect that for every combination *(algorithm, UI element, usage context)*, the reported similarity values could give us better understanding how to improve the quality of test scripts and, therefore, how to make the next steps towards building a testing automation framework for mobile applications based on hand-drawn or non-native GUI components.

V. Discussion

The approach to GUI testing we describe above is applicable to a large variety of applications implementing custom (non-native) user interface. They include desktop and mobile games, HTML5 Canvas-based web applications, and many specialized instruments such as graphical or audio editors or mapping software. Accessing UI elements via the API of the underlying operating system is the preferable way for interacting with application for most tests. However, it is not available if the application UI is not based on standard widgets.

One may argue that using pattern recognition leads to fragile tests, since any changes in application GUI might break test scripts. While this is certainly the case, we have to note that all kinds of GUI tests are prone to fragility, and might fail due to reorganization, renaming, or addition of new UI elements. Writing robust GUI tests is a challenging task regardless of the approach used.

Reliance on screenshot processing slows down the testing process considerably. There are three major reasons. First, the process of screenshot generation on a mobile platform might take several seconds depending on a particular device. Second, the screenshot needs to be transferred to the machine running the test scripts, which requires fast and reliable connection. Third, pattern recognition is also computationally intensive, though in our experiments with five concurrent test processes the largest performance bottlenecks were still caused by the procedure of taking and transferring the next screenshot to the testing machine.

However, let us note that the screenshots provide a useful graphical log of the testing process, and we often resort to it for debugging purposes. So one might consider taking screenshots regardless of the method of accessing GUI controls. Furthermore, accessing UI elements with conventional methods (such as using XPath locators in Appium) can also be slow, and might take several seconds per query depending on a particular situation.

VI. Conclusion and Future Work

Let us note that image recognition algorithms are rarely discussed within the scope of software testing, so we believe that advancing and improving the quality of the proposed approach will provide a feasible solution to be used as a part of integration pipeline in software development and testing.

Current frameworks such as Appium [29] allow running smoke tests on real mobile devices. However, they do not provide built-in capabilities for managing multiple-device tests and for integration of smoke testing into continuous delivery pipeline. Several companies (such as Bitbar and Amazon) offer “mobile test farm” services. They are also used and evaluated by academic researchers [30][31]. However, they are expensive for small developers, offer a limited range of mobile devices, and lack flexibility. There is also an open Smartphone Test Farm initiative [32], but its primary goal is to provide remote control options for Android devices rather than to build an automated cross-platform smoke testing facility. We also have to mention a number of commercial service providers, such as Amazon, Xamarin, and Bitbar supporting heterogeneous multiple-device farm facilities

Our further goal is to create an open source framework for small-scale mobile farms [33]. The aim of this framework is to let anyone to quickly connect their own iOS or Android devices into a fully functional mobile farm, and integrate it into existing continuous delivery pipeline. Our software will make smoke testing of mobile apps easier to set up for the developers, and, therefore, will increase the popularity of automated testing methods, and, consequently, will contribute to the improved quality of software. We believe that a practical testing framework should implement the following capabilities:

1. Concurrent tests on several mobile devices.
2. Automated detailed reports of test results with app screenshots and activity logs.
3. Health monitoring of the devices for early detection of battery drain or device malfunction.
4. Manual and event-driven test runs.

We expect that the approach can be advanced towards designing the architecture of a farm as a distributed system allowing geographically dispersed teams to use their devices effectively. We specifically address the tasks of automating apps with non-native GUIs, such as apps written in Unity (a popular instrument for cross-platform development of games and multimedia software). This project can be considered a transdisciplinary human-centric research [34] that requires applying the solutions achieved in areas such as pattern recognition, intelligent interfaces and usability to a distinct application domain (mobile software testing), rather than straightforward integrated use of available tools and methods.

References

[1] M. Mozgovoy and E. Pyshkin, “Using Image Recognition for Testing Hand-drawn Graphic User Interfaces,” 11th International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies (UBICOMM 2017), Barcelona, Spain, November 12-16, 2017, IARIA, pp. 25-28.

[2] D.M. Rafi, K.R. Moses, K. Petersen, and M. Mäntylä, “Benefits and limitations of automated software testing: systematic literature review and practitioner survey,” In Proceedings of the 7th International Workshop on Automation of Software Test (AST '12), IEEE Press, Piscataway, NJ, USA, 2012, pp. 36-42.

[3] K. Beck, Test-Driven Development by Example. Addison-Wesley Professional, 2002, 240 p.

[4] S. Bellware, “Behavior-Driven Development,” Code Magazine, 2008, vol. 9(3).

[5] E. Pyshkin, M. Mozgovoy, and M. Glukhikh, “On requirements for acceptance testing automation tools in behavior driven software development,” In Proceedings of the 8th Software Engineering Conference in Russia (CEE-SECR), 2012, accessed: November 13, 2018. [Online]. Available:

http://2012.secrus.org/2012/presentations/pyshkin-mozgovoy-glukhikh_80_article.pdf.

[6] Boehm. B. Software Engineering Economics. Prentice Hall, Englewood Cliffs, NJ, 1981.

[7] E. Pyshkin and M. Glukhikh, "Teaching program flow validation: A case study of branch coverage testing," In Ari Lindeman (Ed.), Studies in social sciences, humanities and engineering, The second joint research publication of Peter the Great St. Petersburg Polytechnic University and Kymenlaakso University of Applied Sciences, Kymenlaakso University of Applied Sciences, Kouvola, Finland, 2015, Series A, No. 71, pp. 72-80.

[8] E. Daka and G. Fraser, "A Survey on Unit Testing Practices and Problems," 25th IEEE International Symposium on Software Reliability Engineering (ISSRE), 2014, pp. 201-211.

[9] S. McConnell, "Daily build and smoke test," IEEE software, 1996, vol. 13(4), p. 144.

[10] E. van Veenendaal, "Standard glossary of terms used in software testing," International Software Testing Qualifications Board, 2010, pp. 1-51.

[11] J. Humble and D. Farley, Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Addison-Wesley Professional, 2010, 512 p.

[12] G. Mustafa, A. Shah, K. Asif, and A. Ali, "A Strategy for Testing of Web Based Software," Information Technology Journal, 2007, vol. 6(1), pp. 74-81

[13] "Microsoft Corp. Guidelines for Smoke Testing," MSDN Library for Visual Studio 2008, accessed: November 13, 2018. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms182613(v=vs.90).aspx.

[14] M. Mozgovoy and E. Pyshkin, "Unity application testing automation with Appium and image recognition," In Itsykson V., Scedrov A., Zakharov V. (eds) Tools and Methods of Program Analysis. TMPA 2017. Communications in Computer and Information Science, vol 779. Springer, Cham, pp. 139-150.

[15] Appium, project homepage, accessed: November 11, 2018. [Online]. Available: http://appium.io.

[16] Calabash, project homepage, accessed: November 11, 2018. [Online]. Available: http://calaba.sh.

[17] "World of tennis: Roaring '20s," project homepage, accessed: November 11, 2018. [Online]. Available: http://worldoftennis.com/.

[18] "Automating user interface tests," accessed: November 13, 2018. [Online]. Available: https://developer.android.com/training/testing/ui-testing/index.html.

[19] V. V. Helppi, "Using opencv and akaze for mobile app and game testing," (January 2016), accessed: November 13, 2018. [Online]. Available: http://bitbar.com/using-opencv-and-akaze-for-mobile-app-and-game-testing.

[20] S. Kazmierczak, "Appium with image recognition," (February 2016), accessed: November 13, 2018. [Online]. Available: https://medium.com/@SimonKaz/appium-with-image-recognition-17a92abaa23d\#.oez2f6hnh.

[21] "OpenCV Library," accessed: November 11, 2018. [Online]. Available: http://opencv.org.

[22] "OpenCV: Template Matching", accessed: November 11, 2018. [Online]. Available: http://docs.opencv.org/master/de/da9/tutorial_template_matching.html.

[23] G. Bradski and A. Kaehler, Learning OpenCV: Computer vision with the OpenCV library. O'Reilly Media, Inc., 2008.

[24] R. Laganière, "OpenCV Computer Vision Application Programming Cookbook," 2nd ed., Packt Publishing, 2014.

[25] M. Yamamoto, E. Pyshkin, and M. Mozgovoy, "Reducing False Positives in Automated OpenCV-based Non-Native GUI Software Testing," In Proceedings of the 3rd International Conference on Applications in Information Technology (ICAIT'2018), N. Bogach, E. Pyshkin, and V. Klyuev (Eds.). ACM, New York, NY, USA, 2018, pp. 41-45.

[26] J. Canny, "A computational approach to edge detection," IEEE Transactions on pattern analysis and machine intelligence, no. 6, 1986, pp. 679-698.

[27] GIMP, project homepage, accessed: November 11, 2017. [Online]. Avaliable: https://www.gimp.org/.

[28] "Imaging Web Demonstrations," Biomedical Imaging Group, accessed: November 13, 2018. [Online]. Available: http://bigwww.epfl.ch/demo/ip/demos/edgeDetector/.

[29] N. Verma. Mobile Test Automation with Appium. Packt Publishing, 2017, 231 p.

[30] C. Tao and J. Gao, "Cloud-Based Mobile Testing as a Service," International Journal of Software Engineering and Knowledge Engineering, 2016, vol. 26(1), pp. 147-152.

[31] M. Linares-Vásquez, K. Moran and D. Poshyvanyk, "Continuous, Evolutionary and Large-scale: A New Perspective for Automated Mobile App Testing," 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), 2017.

[32] Smartphone Test Farm, project homepage, accessed: November 11, 2018. [Online] Available: https://openstf.io/.

[33] M. Mozgovoy and E. Pyshkin, "Mobile Farm for Software Testing," In Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct (MobileHCI '18), ACM, New York, NY, USA, 2018, pp. 31-38.

[34] E. Pyshkin, "Designing human-centric applications: Transdisciplinary connections with examples," In Proc. of 2017 3rd IEEE International Conference on Cybernetics (CYBCONF), Exeter, UK, Jun 21-23, 2017, pp. 455-460.