The Computer Games Journal (2019) 8:215–239
https://doi.org/10.1007/s40869-019-00087-z

RESEARCH

MAuto: Automatic Mobile Game Testing Tool Using Image-Matching Based Approach

J. Tuovenen[1] · M. Oussalah[1] · P. Kostakos[1]

Received: 16 May 2019 / Accepted: 10 October 2019 / Published online: 19 October 2019

Abstract
The exponential increase in the speed of the mobile industry has led to a decreasing quality in many associated mobile apps. Besides, the number of distinct Android devices reached thousands. This challenged the development of universally accepted test applications that can run on all devices. This paper focuses on the development of a new mobile game testing framework, referred to, *MAuto*. MAuto records the user actions in the game and replays the tests on any Android device. MAuto uses image recognition, through AKAZE features, to record the test cases and the Appium framework to replay the user actions automatically. The feasibility of the developed tool has been demonstrated through testing on the Clash of Clans mobile game.

Keywords Mobile game · Testing · Image recognition

1 Introduction

In the era of mobile computing and internet technology, mobile gaming has seen a huge increase, gaining almost all population groups. According to SmartInsights (Chaffey 2016), there are about 1.7 billion mobile users worldwide who averagely spend 3 h per day on their mobile phones. In another market study (see, Iqbal 2019) showed that 33% of mobile phone users regularly play games on their phones where 50% of them download apps to their phones.

Besides, with the development of smartphones, fast mobile broadband and platform availability, mobile gaming has moved deeper into the broader culture of individuals and communities. According to Intelligence Blog Sonders (2017), 62% of smartphone users download game applications within a week after purchasing their phones, which is higher than any other downloaded applications. This generated more than \$60 billion of revenues, which are expected to exceed \$100 billion by 2021

✉ M. Oussalah
Mourad.Oussalah@oulu.fi

[1] Centre for Ubiquitous Computing, Faculty of Information Technology and Electrical Engineering, University of Oulu, PO Box 3500, 90114 Oulu, Finland

according to estimates raised by Newzo (2018). On the other hand, the ever-growing popularity of Android that constantly attract new developers and business, has unfortunately led to a wide discrepancy of distinct Android devices employed. As such, ensuring efficient and reliable testing of newly developed mobile game applications becomes of paramount importance and one of the toughest challenges faced by game developers, service providers as well as regulators. This is often referred to as quality assurance (QA) testing, which focuses on identifying technical problems with the games. Although QA techniques appear almost at every stage of the software development lifecycle, starting from requirement eliciting to product deployment and maintenance where a special interest is attributed to testing automation. The latter is an integral part of a continuous integration pipeline (Novak 2008) where simple automated tests are used for basic program elements, such as individual class methods or separate functions. Especially, test automation reduces the time, cost and resources, while enhancing the reliability through exposure to a large number of test cases that cannot be performed solely by human interaction in practice.

Compared to traditional desktop-applications, test automation for mobile applications bears additional challenges. First, through sandboxing, only limited access to internal processes is provided, which challenges the developers to optimize resource allocation. Second, the general user interface navigation of mobile apps is vulnerable and hard to control due to uncertainty pervading the response time of the interface (s). This includes, for instance, *grab* and *hold* like interactions. This problem is also referred to as the *fragile test* issue pointed out by Meszaros (2007). That is why it is recommended that the application functional logic should not be tested via the application's user-interface, although such rules are often violated by developers themselves. Third, mobile devices are often in a steady movement, which can cause the currently executed test-case for an app to break down. Fourth, often the complexity of the allocation task together with resource limitations cause a change in size and resolution of the screen, which, in turn, makes any user-interface based test likely to fail. Fifth, despite the effort to harmonize the software-hardware configuration in mobile platforms, the number of distinct configurations is sharply increasing, which makes the application of a single testbed very difficult. For instance, the number of distinct Android devices is exponentially increasing (e.g., more than 24,000 distinct (Android) devices were reported in 2015[1]). Therefore, it is nearly impossible to test an application on every distinct device in a real environment and, at the same time, provide the best user experience where the underlined application works flawlessly on all other devices. Sixth, mobile games bear additional inherent features that add extra difficulties. For instance, games involve a lot of graphics and other assets, which substantially increase the loading time. This, in turn, challenges the efficiency of the resource allocation policy. Besides, games have inherent hooks that are intended to make the player play the game again and again (Novak 2008). This makes it difficult to automate the process of accessing the various passes of the game. Finally, games bear a psychological factor referred to as *fun factor* (Novak 2008). Indeed, even in the case of a bug-free scenario, the game can fail because

---

[1] https://qz.com/472767/there-are-now-more-than-24000-different-android-devices/.

the players do not feel the fun factor so that their actions are random and not comply with game rules. Because of its inherent subjectivity and vulnerability from one player to another, it is almost impossible to automate the fun factor in testing.

Due to the above challenges and the lack of effective fully automated testing platforms, mobile app testing is still performed mostly manually, costing the developers and the industry significant amounts of effort, time, and money (Novak 2008; Choudhary et al. 2015; Kochhar et al. 2015). This requires attention from both the research community and practitioners. Although, setting up the test automation scheme would imply an additional investment, sometimes referred to as the "hump of pain" learning curve, the expected benefits gained from this process will return back such investment sooner or later (Crispin and Gregory 2011).

In this perspective, we present in this paper a new take on a mobile game application testing called *MAuto*. The latter aims to help the tester to create tests that work with Android games. The tests can then be re-run on any other Android device. The tool records tests from user-interactions and exports them to *Appium* framework (Appium 2012) for playback. MAuto belongs to the class of image-based recognition tests where *AKAZE* features (Alcantarilla et al. 2013) were used to recognize the objects from the screenshots. When the user performs the recording, MAuto generates a test script that reproduces the recorded events. MAuto then uses Appium framework to perform the replay of the test script task. To validate the developed MAutol, tests are created with the tool for Hill Climb Racing mobile game and successfully executed. The rest of this paper is organized as follows. Section 2 reviews the state of the art in the field of mobile testing. The description of the developed MAuto system is reported in Sect. 3 of this paper, while experimentation and exemplification using Hill Climb Racing game are examined in Sect. 4. Section 5 summarizes the key findings and ways forward.

2 State of Art

2.1 Test Automation Pyramid

The traditional test automation pyramid introduced by Cohen (2006) is highlighted in Fig. 1. It consists of a three layer-pyramid corresponding to End-to-End (E2E) test, an Integration test and a Unit test at the top of the hierarchy. The width of the pyramid reflects the number of tests to be written in each layer (Knott 2015). Usually, manual testing is not part of Cohen's test automation pyramid so it was drawn as a cloud on the top of the pyramid of Fig. 1 for illustration purposes only.

Mobile test automation tools are not yet good enough to support the traditional test automation pyramid. Besides, mobile devices are armed with a variety of sensors (e.g., camera, accelerometer, gyroscope, infrared, GPS) and other distinguished features (e.g., memory and CPU resources and various embedded software that accommodate current and future installed apps), which restrict the development of universally accepted testing tools (Knott 2015). We primarily focus on E2E testing because of its criticality.

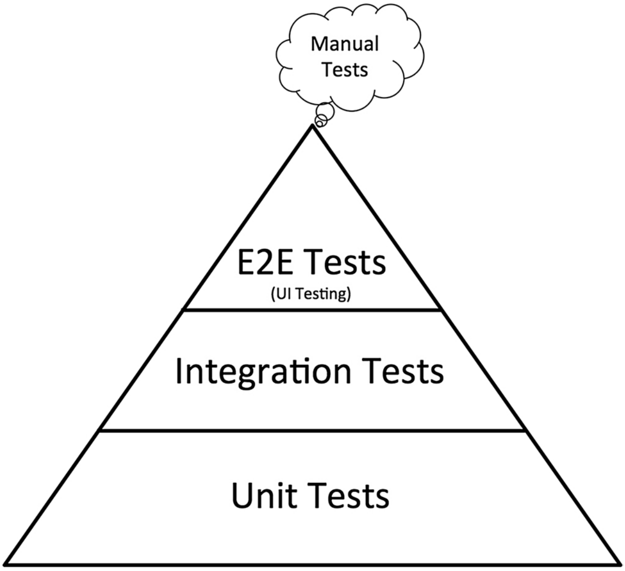


Fig. 1 Traditional test automation pyramid (Knott 2015)

2.2 Types of Mobile Test Automation Tools

We distinguish five types of test automation modes: image-based, coordinate-based, OCR/text recognition, native object recognition, gesture record and replay (Knott 2015).

2.2.1 Image-Based Tools

The key in this testing mode is to determine the type and location of icons/objects on the screen to be matched with a set of predefined graphical elements of the game taking into account the user's actions and status of the game. More specifically, the application objects and controls are stored as images, which are then compared to the objects displayed on the screen to identify any potential matching. Once a match is found, the pre-defined step can be re-executed. These frameworks can also associate specific actions, such as clicks and text entry, to the controls. Besides, every user-interface (UI) object, which includes buttons, text boxes, selection lists, icons, among others, has a set of properties that can be used to identify, define or validate the underlying object (MacKenzie 2012). This provides the tester with useful and powerful tools for GUI-testing. As a result, the automation engineer achieves a high reusable and good maintainable low-cost script development. Such a method is widely accepted in the field and recognized as the best practice according to test automation.

However, it is acknowledged that the image recognition-like technique runs on elaborate and time-consuming pixel-comparison algorithms. Image-recognition automation is also infeasible if application objects are dynamic. On the other hand, such tests can be fragile if the predefined graphical elements are not carefully chosen. For instance, badly chosen algorithms or algorithm parameters can lead to flaky tests (Knott 2015). Therefore cautious analysis of the context is needed before the application of image like technique.

2.2.2 Coordinate-Based Recognition

In this approach, user actions are captured and automated based on their associated *x–y coordinates* on the screen. This allows interactions with UI elements such as buttons and images present at specific, pre-defined locations in the application UI to be reproduced. However, if the screen orientation or object layout changes, scripts need to be rewritten. Indeed, the test just blindly executes a given action on a given coordinate so that whenever the screen size varies between devices under testing, the test can be broken down easily (Knott 2015). Therefore, the approach is rarely applied in practice, and only very few tools provide coordinate-based identification.

2.2.3 Optical Character/Text Recognition

The key in this approach is to identify the characters displayed on screens, e.g., “login” or “logout” button, by matching the text with the correct object on the screen, to determine the relevant application control (s).

However, OCR technology is dependent on the ability to identify the visible text, so that any blurring or screen resolution change may have a negative effect on the identification accuracy. Also, such tools are not suitable to test user-interface elements that are not visible or are continuously changing. Untestable elements for OCR and text matching would include a list of options that are not visible such as application controls that might have hidden text or dynamic text such as account balances or clocks that live-update. OCR recognition tools tend to be slower than other types of tools because they need to scan the whole screen for the text (Knott 2015). Therefore such techniques experience significant limits and, thereby, are commonly used in tandem with image-based recognition tools.

2.2.4 Native Object Recognition

Native object identification is based, first, on recognizing application object properties in the application code, such as *ID*, *XPath*, and, second, testing those elements. Especially, native object recognition is one of the most widely used types of mobile test automation tools where the UI objects are identified using the UI element tree. There are many ways to access the UI elements, such as XML Path Language (XPath), Cascading Style Sheet (CSS) locators or the native object ID of the element. With native object recognition, the developer can define the IDs or the locators properly and build very robust tests. The biggest advantage of this approach is that it does not depend on changes in the UI, orientation, resolution or the device itself (Knott 2015). The identification of programmatic objects makes this technique the most resilient to changing code, and hence quite reliable, although, it requires more effort and programming knowledge.

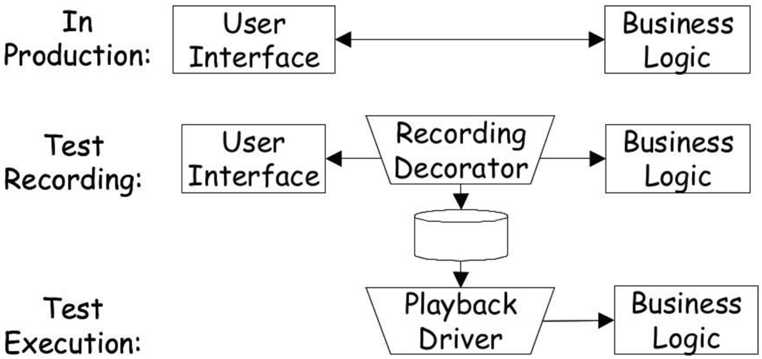


Fig. 2 Record and replay using a recording decorator adapted from MacKenzie (2012)

2.2.5 Gesture Record and Replay (GRR)

The basis of this approach is to record screen interactions during manual testing, including every mouse movement, keystroke, and screenshot to be replicated later on. This utility usually comes bundled as a record and a playback tool to enable testers with no programming skills to record and replay the flow of a test case. The test is primarily used for repetitive testing across various platforms and device models. Since each recording is unique, this automation technique is only meaningful in case of stable applications that do not involve important UI modifications. This concerns mainly quick and easy automation of unchanging flows. However, whenever the environment becomes dynamic with external interruptions such as incoming text and calls notifications or changes in orientation/layout, this approach has shown serious limitations. Many tools such as UFT and Perfecto[2] have capture-and-replay capabilities.

Figure 2 describes the basic principle of R&R like technique (MacKenzie 2012). In the record stage, the UI is connected to the business logic directly. When the test is recorded, the signals from the UI are intercepted by the Recording Decorator. Once the signals are stored, the decorator sends the signals to the business logic and the AUT will continue as it would without the decorator. During the execution phase of the test, the UI is put on the sleep mode. Playback Driver reads the signals from the container and sends them to the business logic.

Besides, in practice, many test automation tools are a combination of the aforementioned types and they are not usually locked into a single object recognition type. Every type has its pros/cons, so the developer has to choose the best approach that fits his needs and constraints based on the mobile platform employed and the mobile game properties (Knott 2015). Adapted from Linares-Vásquez et al. (2017), Table 1 summarizes the main tools employed in Automation framework APIs, Record & Replay Tools, Automated GUI-input generation tools.

---

[2] https://www.aspiresys.com/WhitePapers/QTPvsSelenium.pdf.

Table 1 Overview of automation frameworks and APIs

Name	GUI-automation	OS API automation	Black box	Test-case recording	Cross-device support	Natural language test cases	Open source
Automation frameworks and APIs							
UIAutomator (2018)	Yes	No	Either	No	Limited	No	Yes
UIAutomation (iOS) (2015)	Yes	No	No	Yes	Yes	No	Yes
Espresso (2015)	Yes	No	No	No	Limited	No	Yes
Appium (2012)	Yes	No	Yes	Yes	Limited	No	Yes
Robotium (2010)	Yes	No	Yes	Yes	Limited	No	Yes
Roboelectric (2012)	No	Yes	No	No	Yes	No	Yes
Ranorex (2017)	Yes	No	Yes	Yes	Yes	No	No
Calabash (2012)	Yes	No	No	No	No	Yes	Yes
Quantum (2012)	Yes	N/A	No	N/A	N/A	Yes	No
Qmetry (2016)	Yes	N/A	No	N/A	N/A	Yes	No
	GUI support	Sensor support	Root access required	Cross-device	High-level test cases	Open source	
Record and replay tools							
RERAN (Gomez et al. 2013)	Yes	No	Yes	No	No	Yes	
VALERA (Hu et al. 2015)	Yes	Yes	Yes	No	No	No	
Mosaic (Halpern et al. 2015)	Yes	No	Yes	Limited	Yes	Yes	
Barista (Fazzini et al. 2016)	Yes	No	No	Yes	Yes	No	
Robotium Recorder (2014)	Yes	No	No	Limited	Yes	No	
Xamarin Test Recorder (Xamarin 2018)	Yes	No	No	Yes	Yes	No	
ODBR (Moran et al. 2017a, b)	Yes	Yes	Yes	Limited	Yes	Yes	
SPAG-C (Lin et al. 2014)	Yes	No	N/A	N/A	No	No	
Espresso Recorder (2015)	Yes	No	No	Limited	Yes	Yes	

Table 1 (continued)

Tool name	Instrumentation	GUI exploration	Types of events	Replayable test cases	NL crash reports	Emulators, devices	Open source
Automated GUI-input generation tools							
Random-based input generation							
Monkey (2016)	No	Random	System, GUI, Text	No	No	Both	Yes
Dynodroid (Machiry et al. 2013)	Yes	Guided/random	System, GUI, Text	No	No	Emulators	Yes
Intent Fuzzer (Sasnauskas and Regehr 2014)	No	Guided/random	System (Intents)	No	No	N/A	No
VANARSena (Ravindranath et al. 2014)	Yes	Random	System, GUI, Text	Yes	No	N/A	No
Systematic input generation							
AndroidRipper (Amalfitano et al. 2012)	Yes	Systematic	GUI, Text	No	No	N/A	Yes
ACTEve (Anand et al. 2012)	Yes	Systematic	GUI	No	No	Both	No
A3E Depth-First (Azim and Neamtiu 2013)	Yes	Systematic	GUI	No	No	Both	Yes
CrashScope (Moran et al. 2017a, b)	No	Systematic	GUI, text, system	Yes	Yes	Both	No
Google RoboTest (2015)	No	Systematic	GUI, text	No	Yes	Devices	No
Model-based input generation							
MobiGUItar (Amalfitano et al. 2014)	Yes	Model-based	GUI, text	Yes	No	N/A	Yes
A3E Targeted (Azim and Neamtiu 2013)	Yes	Model-based	GUI	No	No	Both	No
Swifthand (Choi et al. 2013)	Yes	Model-based	GUI, text	No	No	Both	Yes
QUANTUM (Zaeem et al. 2014)	Yes	Model-based	System, GUI	Yes	No	N/A	No
ORBIT (Yang et al. 2013)	No	Model-based	GUI	No	No	N/A	No
MonkeyLab (Linares-Vásquez et al. 2017)	No	Model-based	GUI, text	Yes	No	Both	No

Table 1 (continued)

Tool name	Instrumentation	GUI exploration	Types of events	Replayable test cases	NL crash reports	Emulators, devices	Open source
Zhang and Rountev (2017)	No	Model-based	GUI, text	N/A	N/A	Both	Yes
Other types of input generation strategies							
PUMA (Hao et al. 2014)	Yes	Programmable	System, GUI, text	No	No	Both	Yes
JPF-Android (Van der Merwe et al. 2014)	No	Scripting	GUI	Yes	No	N/A	Yes
CrashDroid (White et al. 2015)	No	Manual Rec/replay	GUI, text	Yes	Yes	Both	No
Collider (Jensen et al. 2013)	Yes	Symbolic	GUI	Yes	No	N/A	No
SIG-Droid (Mirzaei et al. 2015)	No	Symbolic	GUI, text	Yes	No	N/A	No
Thor (Adamsen et al. 2015)	Yes	Test cases	Test case events	N/A	No	Emulators	Yes
AppDoctor (Hu et al. 2014)	Yes	Multiple	System, GUI2, text	Yes	No	N/A	No
EvoDroid (Mahmoud et al. 2014)	No	System/Evo	GUI	No	No	N/A	No
Sapienz (Mao et al. 2016)	Yes	Search-based	GUI, text, system	Yes	Yes	Both	Yes
Jabbarvand et al. (2016)	Yes	Search-based	GUI, text, system	Yes	Yes	Both	Yes

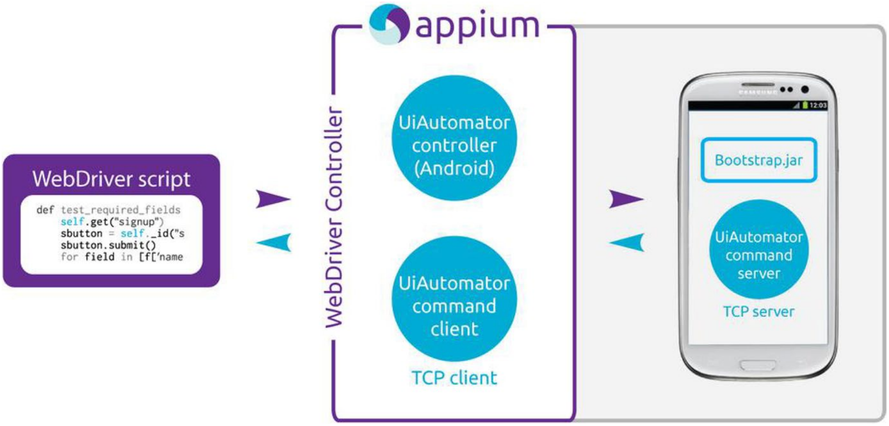


Fig. 3 Appium on Android architecture

On the other hand, one distinguishes noticeable tools that are of paramount importance for the developers: *Appium* (2012) is an open-source test automation framework that can test native, hybrid and mobile web applications on Android, iOS and Windows platforms. One special feature of Appium is that the developers do not have to modify the application binaries to test the application, because Appium uses vendor-provided automation frameworks. On the other hand, Appium uses WebDriver[3] protocol to wrap the vendor-provided framework into a single API. WebDriver specifies a client–server protocol (known as the JSON Wire Protocol[4]) for the communication. The clients have been written in many major programming languages like Ruby, Python, and Java.[5]

In terms of software implementation, Appium sets up a server into the host machine. The client, where the test logic is located, connects to the server. If the operating system of the device is Android, the server forwards the commands from the client to the device via UI Automator frame-work (see, Fig. 3). On older Android devices the server communicates with the device via Selendroid (Android API level < 17).

Close alternative candidates to image-based automation tools are summarized below.

SikuliX[6]: is a tool that automates everything on the screen. It is formally known as Sikuli.[7] It uses OpenCV image recognition to find the objects to click on the screen. SikuliX does not have support for mobile devices out of the box, but it is possible to make it work with simulators, emulators or VNC (virtual network computing)

---

[3] http://docs.seleniumhq.org/projects/webdriver/.

[4] https://code.google.com/p/selenium/wiki/JsonWireProtocol.

[5] http://appium.io/downloads.

[6] http://www.sikulix.com/.

[7] https://github.com/sikuli/sikuli.

Table 2 Review of main application testing tools

	Desktop support	Mobile support	Image recognition capabilities
SikuliX	X		X
Testdroid Recorder		X	
Robotium Recorder		X	
Appium GUI		X	
Jautomate	X		X

solutions where the mobile device screen can be accessed from the desktop (Yeh et al. 2009; Chang et al. 2010).

JAutomate (Alegroth et al. 2013): is a commercial tool combining image recognition with Record and Replay functionality. JAutomate does not support mobile devices out of the box, but it is possible to make it work with simulators, emulators or VNC solutions where the mobile device screen can be accessed from the desktop.

Also, in terms of record and replay capability, one shall mention *Testdroid Recorder*[8] a free plugin for Eclipse.[9] It is a record-and-replay tool that records user actions with the application under testing (AUT) and generates reusable Android JUnit, Robotium[10] and ExtSolo[11] tests. The generated tests can be replayed afterward.

Robotium Recorder[12]: is a commercial plugin for Android Studio, very similar to Testdroid Recorder and it can also record and replay Robotium tests.

Appium GUI[13]: is a project which provides a graphical user interface (GUI) for Appium. There is an inspector which can tell information about the objects on the screen and also a recorder that can record and replay Appium tests. Table 2 summarizes the aforementioned main application testing tools.

Nevertheless, despite the multiplicity of the mobile automation testing tools as highlighted in Tables 1 and 2, the effectiveness of such tools was limited in practice, especially when dealing with complex interactive mobile games as pointed out in [56–57], which motivates the proposed MAuto.

---

[8] https://www.testtoolreview.de/en/testtool-overview/tool-list/tooldetail/642-testdroid_recorder.

[9] https://eclipse.org/.

[10] https://github.com/RobotiumTech/robotium.

[11] https://github.com/bitbar/robotium-extensions.

[12] http://robotium.com/products/robotium-recorder.

[13] https://github.com/appium/appium-dot-app.

3 MAuto

3.1 Motivation and Rationality

Usually, the functionality of a mobile game is executed during the runtime stage in a graphic container, e.g., OpenGL, to provide better graphics and interaction capabilities to users. Often, the container wraps all the functionality of a game. Thus, it is not possible to access the wrapped functionality to test it. Several methods have been developed to overcome this problem. The most common and effective techniques are (1) programming the container in a particular way to expose functionality outside the container and (2) implementing image recognition approaches to identify functionality from the screen, which is then transmitted to the testing process.

Nevertheless, to use an image recognition-based technique, the user needs the graphical representation of the object to find, e.g., buttons or game characters. Sometimes the user can get those elements directly from the graphics designer, but this is not always the case. Also, the game might change the environment and the context where the object is presented, e.g., shadows and lighting, which, in turn, will affect the success ratio of image recognition. Therefore, it is better to use the actual context from the game and take screenshots while playing the game.

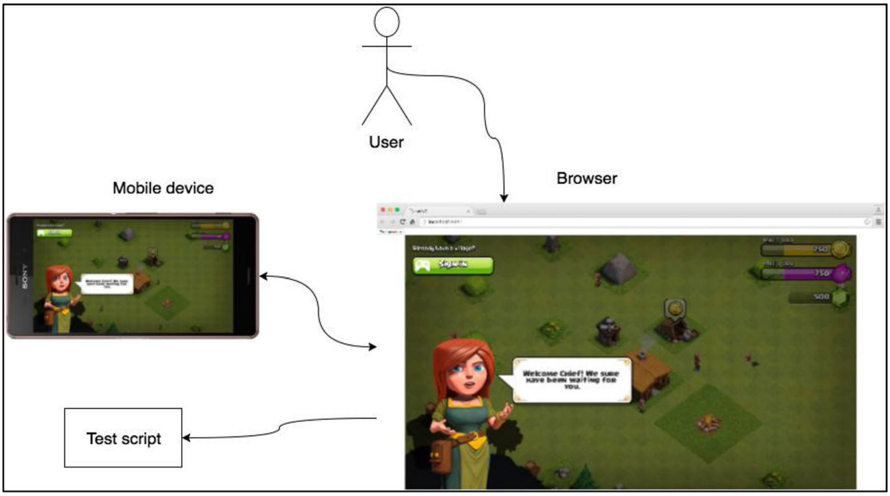
In this sequel, screenshots are taken while the game is running in the mobile device and objects are extracted in a real context. Indeed, the screenshots are stored in the memory of the mobile device so that the user needs to transfer the image to his/her machine. Once the screenshot becomes available in the user's machine, the object or partial image could be extracted from the screenshot. Finally, when every object required to run the game is automatically extracted, the user can utilize these objects to write the automation code that replays the sequence he played before.

To make the cycle above easier and faster for the user, we propose a tool called MAuto. Especially, MAuto will automatically take the screenshots and extract the objects from those screenshots while the user is playing the game. Once the sequence is ready, MAuto will generate the Appium test code to replay the sequence. The design and architecture of MAuto are detailed in the next section.

3.2 General Architecture

The developed MAuto mobile is a new mobile game automatic testing tool, which targets users without programming skills. Indeed, the tool enables generating an Appium test without a single line of code. From the mobile game categorization techniques highlighted in Sect. 2, MAuto makes use of two of the above techniques: image-based recognition and Record & Replay like technique. The image-based approach uses AKAZE features (accelerated KAZE features) (Alcantarilla et al. 2012).

From an input–output perspective, MAuto overall architecture involves three elements; namely, user, browser and mobile device, and next, it generates a test script that the user can run later on (see Fig. 4). Once the user has launched MAuto, all the



[Virtual environment] Appium

Fig. 4 System overview highlighting its components: user, mobile and browser

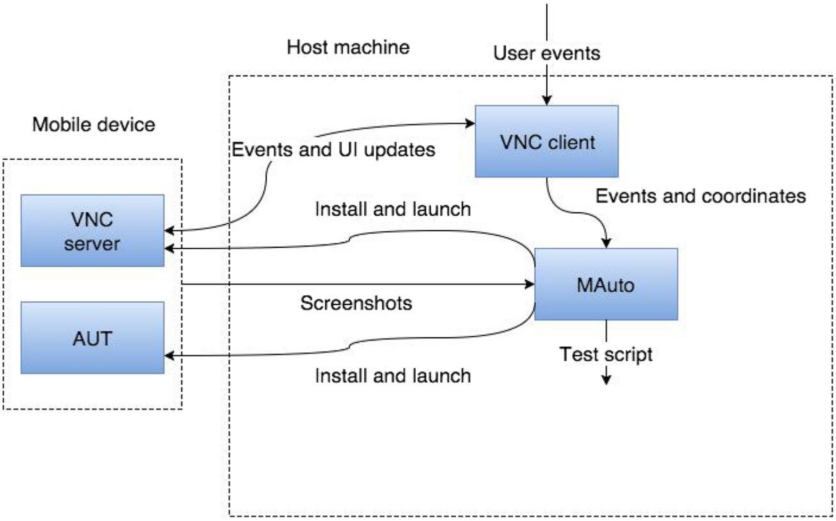


Fig. 5 Architectural design

interactions between the user and the tool make use only the browser input. MAuto takes care of the mobile device so that the user role is reduced to start MAuto and interact with the application via a web browser. When the user performed the recording task, MAuto will generate a test script that can reproduce the recorded events. MAuto itself is not able to replay the test, but the test script can be replayed with Appium.

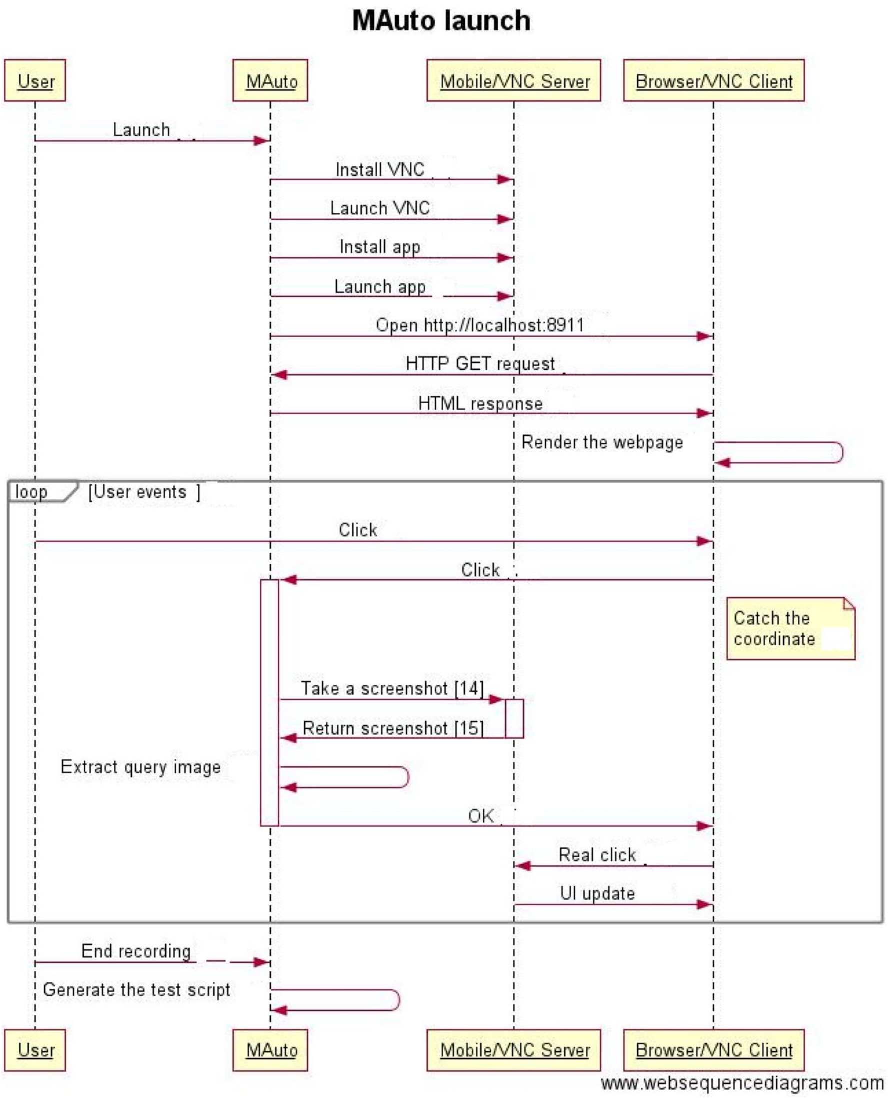


Fig. 6 MAuto sequence diagram

In Fig. 5 is described a more detailed view of the system. The two physical components are a mobile device and the host machine. In the beginning, MAuto installs and launches the Application Under Test (AUT) and the VNC (Virtual Network Computing) server to the device. Then MAuto initiates the connection between the VNC server and the client. If a user-related event occurs, the VNC client forwards this event together with its coordinates to MAuto. The latter captures the screenshot from the mobile device, saves them in a separate database, and gives VNC client permission to continue its processing. VNC client sends the same event to the VNC server. Next, the UI view from the mobile device will be updated to the VNC client. When the user has finished his manipulations, MAuto generates the test script from the screenshots and events.

In summary, MAuto acts as an R&R tool where the tool records the user interactions and Appium is employed to replay the tests. The recording decorator is a modified VNC viewer in the browser, while the replay driver is an Appium test together with the image recognition module.

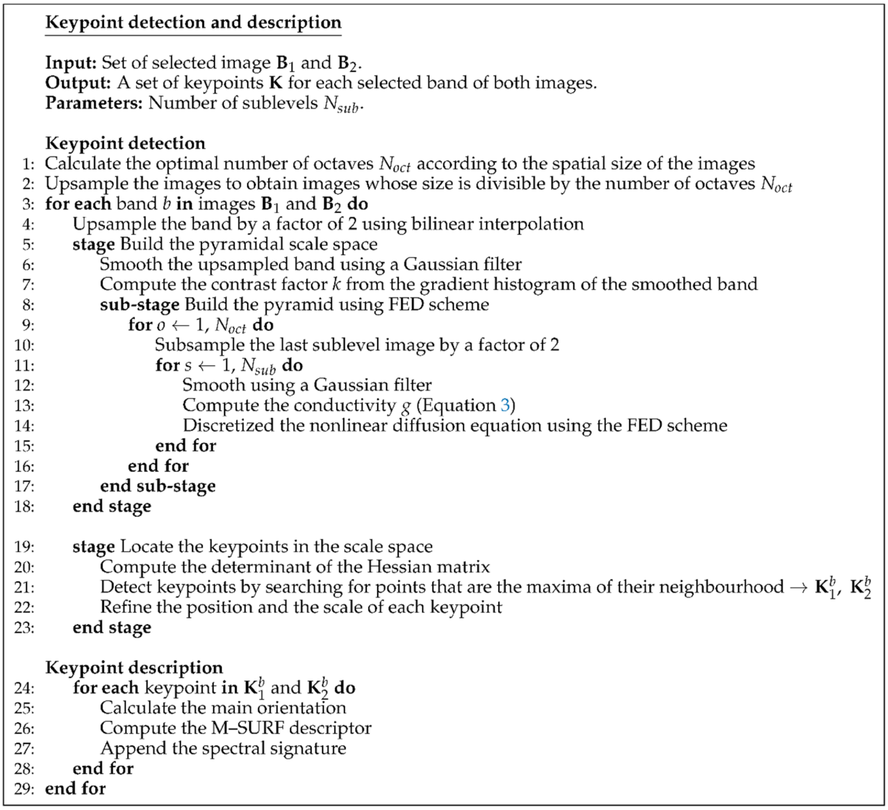
Figure 6 summarizes how the recording sequence interacts with MAuto tool. At first, the user launches the MAuto from the command line, which is also transmitted to AUT so that MAuto installs the VNC client in the mobile device. Then it installs and executes AUT. MAuto runs an accessible webserver such that whenever the recording task is ready, MAuto opens up the associated webpage. The user can then visualize and monitor various UIs of the device so that the events associated with the VNC client running on the browser can be monitored. Next, MAuto runs a modified VNC client which will send the event (s) together with the associated coordinates to MAuto. The latter saves the event, takes a screenshot from the screen of the mobile device and extracts the query image around the event coordinates.

The API call from the VNC client to MAuto is enabled after the corresponding images have been processed. Then the VNC client passes the event to the device via VNC protocol. In turn, the UI in the VNC client will be updated. This will continue until the user decides to stop the recording. Finally, when MAuto gets the command to stop the recording, it generates the test script which can be used with Appium to replay the test on any given device.

MAuto stores the screenshots and query images to the session folder. The latter has a CSV file where the events and image names are saved. The test script generator loads the CSV file from the disk and transforms it into Appium compatible test file.

3.3 Image Recognition in MAuto

MAuto calculates AKAZE features in the query image and the current screenshot. Fast Explicit Diffusion schemes (FED) are used in AKAZE to speed up the feature detection in nonlinear scale-spaces. AKAZE also introduces Modified-Local Difference Binary (M-LDB) to preserve low computational demand and storage requirements. Once both features (at the original image model and current image) are calculated, thresholding is employed to compare those features to ascertain whether the query image is currently shown on the screen and ascertain its coordinates accordingly, see Fig. 7 for a detailed implementation description. Examples of experimental results using these features are reported in Sect. 4 of this paper, see, e.g., Fig. 12 where green circles are the calculated features and the red lines are the matching features in both images. Especially, once the matching features are identified, we calculate the average coordinate from the inliers to get the coordinate of the query image in the screenshot.



```
Keypoint detection and description

Input: Set of selected image $B_1$ and $B_2$.
Output: A set of keypoints $K$ for each selected band of both images.
Parameters: Number of sublevels $N_{sub}$.

Keypoint detection
Calculate the optimal number of octaves $N_{oct}$ according to the spatial size of the images
Upsample the images to obtain images whose size is divisible by the number of octaves $N_{oct}$
for each band $b$ in images $B_1$ and $B_2$ do
    Upsample the band by a factor of 2 using bilinear interpolation
    stage Build the pyramidal scale space
        Smooth the upsampled band using a Gaussian filter
        Compute the contrast factor $k$ from the gradient histogram of the smoothed band
        sub-stage Build the pyramid using FED scheme
            for $o \leftarrow 1, N_{oct}$ do
                Subsample the last sublevel image by a factor of 2
                for $s \leftarrow 1, N_{sub}$ do
                    Smooth using a Gaussian filter
                    Compute the conductivity $g$ (Equation 3)
                    Discretized the nonlinear diffusion equation using the FED scheme
                end for
            end for
        end sub-stage
    end stage

    stage Locate the keypoints in the scale space
        Compute the determinant of the Hessian matrix
        Detect keypoints by searching for points that are the maxima of their neighbourhood $\rightarrow K_1^b, K_2^b$
        Refine the position and the scale of each keypoint
    end stage

Keypoint description
    for each keypoint in $K_1^b$ and $K_2^b$ do
        Calculate the main orientation
        Compute the M–SURF descriptor
        Append the spectral signature
    end for
end for
```

Fig. 7 Pseudo-code implementation of AKAZE based image recognition algorithm

4 Exemplification and Evaluation

MAuto tool has been tested and validated using Clash of Clans Android mobile game (version 8.551.4) (available from Supercell[14]). Clash of Clans (CoC) is a mobile Massive Multiplayer Online Game (MMO/M-MOG) where the player builds a community, trains troops and attacks other players to earn assets. The game has a tutorial that the player should pass to play the game. The tutorial guides the player to click certain elements to continue the game. Therefore, if the tutorial can be passed without serious bugs, it is likely that the game works properly. Besides, since the variations are quite limited in the tutorial, this makes it a good test subject for MAuto. During the recording phase of the MAuto, the browser pops up indicating the readiness to start the recording task as seen in Fig. 8.

---

[14] http://supercell.com/.

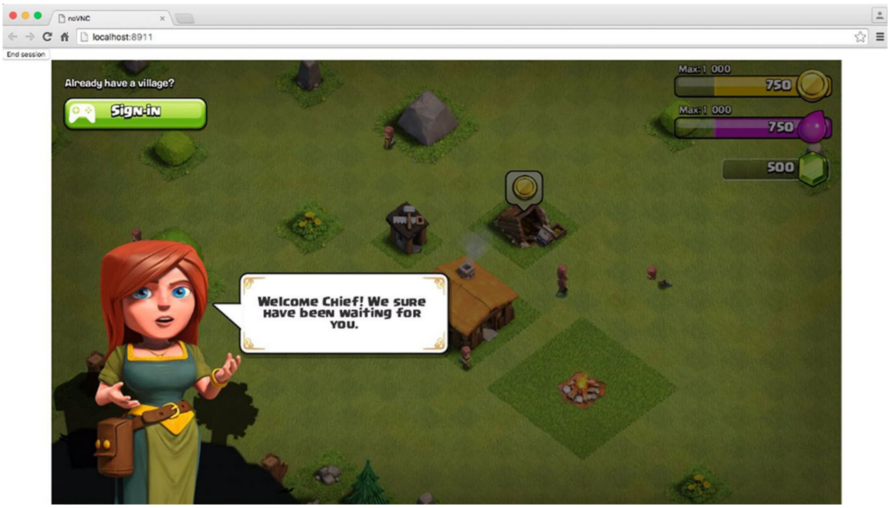
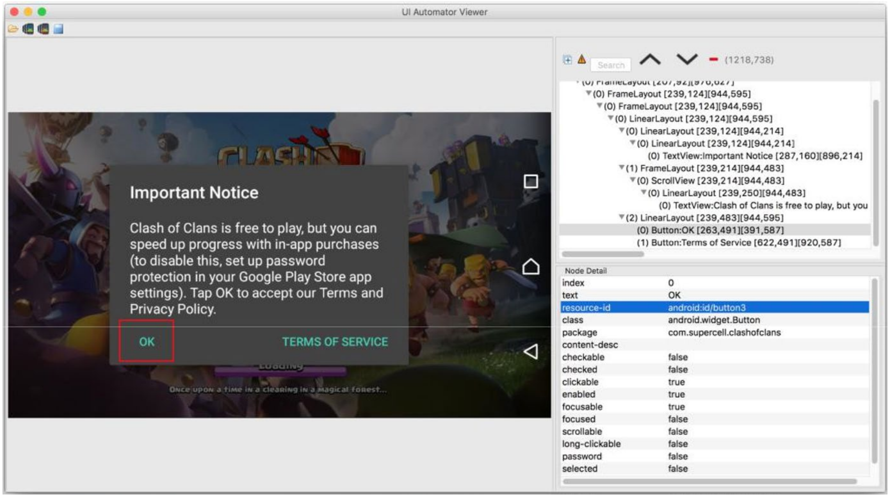


Fig. 8 Browser's view in case of Clash of Clans



Important Notice

Clash of Clans is free to play, but you can speed up progress with in-app purchases (to disable this, set up password protection in your Google Play Store app settings). Tap OK to accept our Terms and Privacy Policy.

OK TERMS OF SERVICE

```
 (0) FrameLayout [207,92][976,627]
  ▼(0) FrameLayout [239,124][944,595]
   ▼(0) FrameLayout [239,124][944,595]
    ▼(0) LinearLayout [239,124][944,595]
     ▼(0) LinearLayout [239,124][944,214]
      ▼(0) LinearLayout [239,124][944,214]
         (0) TextView:Important Notice [287,160][896,214]
     ▼(1) FrameLayout [239,214][944,483]
      ▼(0) ScrollView [239,214][944,483]
       ▼(0) LinearLayout [239,250][944,483]
          (0) TextView:Clash of Clans is free to play, but you
     ▼(2) LinearLayout [239,483][944,595]
        (0) Button:OK [263,491][391,587]
        (1) Button:Terms of Service [622,491][920,587]
```

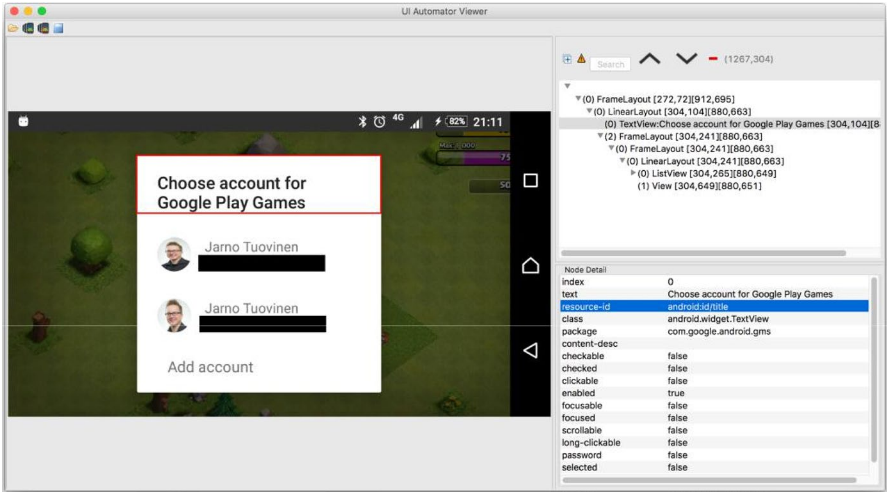
index	0
text	OK
resource-id	android:id/button3
class	android.widget.Button
package	com.supercell.clashofclans
content-desc	
checkable	false
checked	false
clickable	true
enabled	true
focusable	true
focused	false
scrollable	false
long-clickable	false
password	false
selected	false

Fig. 9 The first view which requires interaction in Clash of Clans

The first view which requires user interaction in Clash of Clans is the important notice view, see Fig. 9. This view is accessible using the native object recognition tool. The object's resource ID is android:id/button3 whose associated package is com.supercell.clashofclans.

Clash of Clans requires the user to select one Google Play account for the game, accessible using the native object recognition as well (see Fig. 10).

The subsequent views are no longer accessible using native object recognition, therefore, the use of image recognition is needed. See an example of this



Choose account for Google Play Games

Jarno Tuovinen
[redacted]

Jarno Tuovinen
[redacted]

Add account

```
▼
  ▼(0) FrameLayout [272,72][912,695]
    ▼(0) LinearLayout [304,104][880,663]
       (0) TextView:Choose account for Google Play Games [304,104][8
      ▼(2) FrameLayout [304,241][880,663]
        ▼(0) FrameLayout [304,241][880,663]
          ▼(0) LinearLayout [304,241][880,663]
            ▶(0) ListView [304,265][880,649]
             (1) View [304,649][880,651]
```

Node Detail

index	0
text	Choose account for Google Play Games
resource-id	android:id/title
class	android.widget.TextView
package	com.google.android.gms
content-desc	
checkable	false
checked	false
clickable	false
enabled	true
focusable	false
focused	false
scrollable	false
long-clickable	false
password	false
selected	false

Fig. 10 Google Play account selection

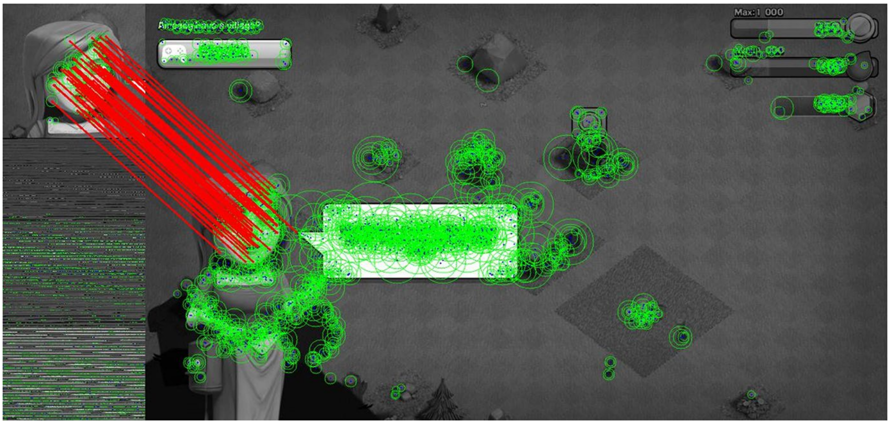
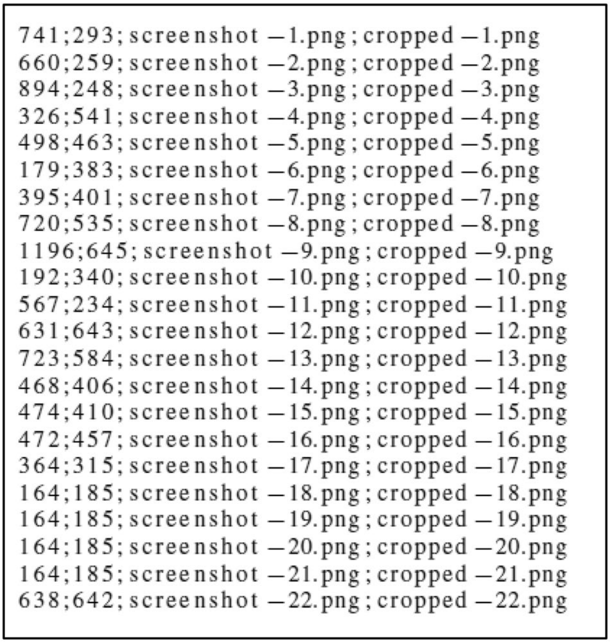


Fig. 11 Example of image recognition using AKAZE features—query image is correctly matched from the screen

task in Fig. 11. The green circles are the calculated features and the red lines are the matching features in both images. Once we have the matching features, we calculate the average coordinate from the inliers to get the coordinate of the query image in the screenshot.

An example of a file created by MAuto in order to save the click coordinates, screenshots and query images is shown in Fig. 12, while an example of Appium script in MAuto is reported in the "Appendix" of this paper.

Next, the Record and Replay phase is carried out using Appium test script that takes into account the image recognition based approach. An example of Appium test script for this purpose is shown in "Appendix".



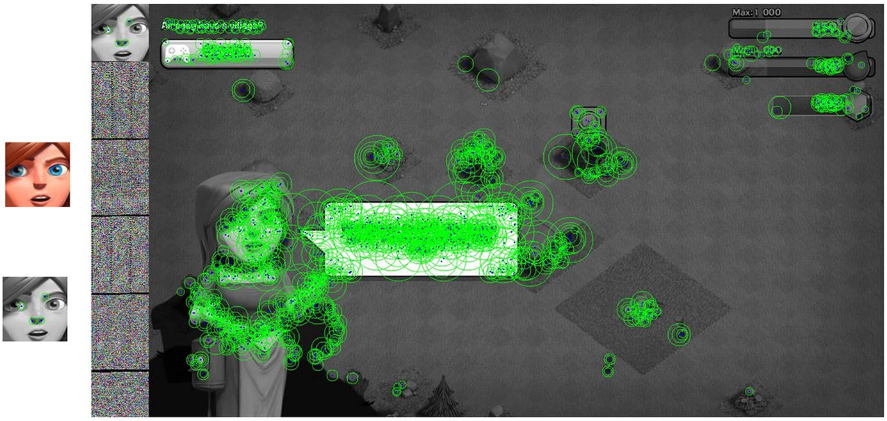
```
741;293;screenshot −1.png;cropped −1.png
660;259;screenshot −2.png;cropped −2.png
894;248;screenshot −3.png;cropped −3.png
326;541;screenshot −4.png;cropped −4.png
498;463;screenshot −5.png;cropped −5.png
179;383;screenshot −6.png;cropped −6.png
395;401;screenshot −7.png;cropped −7.png
720;535;screenshot −8.png;cropped −8.png
1196;645;screenshot −9.png;cropped −9.png
192;340;screenshot −10.png;cropped −10.png
567;234;screenshot −11.png;cropped −11.png
631;643;screenshot −12.png;cropped −12.png
723;584;screenshot −13.png;cropped −13.png
468;406;screenshot −14.png;cropped −14.png
474;410;screenshot −15.png;cropped −15.png
472;457;screenshot −16.png;cropped −16.png
364;315;screenshot −17.png;cropped −17.png
164;185;screenshot −18.png;cropped −18.png
164;185;screenshot −19.png;cropped −19.png
164;185;screenshot −20.png;cropped −20.png
164;185;screenshot −21.png;cropped −21.png
638;642;screenshot −22.png;cropped −22.png
```

Fig. 12 Example of MAuto output file

5 Discussion

It is very time consuming and prone to errors for human testers to take regular screenshots from the device, transfer them to the host machine and crop appropriate query image. However, such a repetitive task can more efficiently be automated. MAuto is designed to do so and, thereby, decrease the amount of required manual work. It can take the screenshots and transfer the images to the host machine automatically. More specifically, MAuto crops the images properly and creates reusable tests through the appropriate use of Appium. To demonstrate its feasibility and technical soundness, MAuto was used to create automated test scripts for Clash of Clans mobile game. Strictly speaking, although MAuto does not automate everything, still it can significantly improve the speed of test automation script creation. Nevertheless, the selected query images have a huge impact on test stability on other devices. Indeed, the query image must have a good layout for the AKAZE features to be matched appropriately on the screen. Figure 13a highlighted an example of the query image of click where MAuto and AKAZE found only 4 features so that most likely this query image cannot be found from the screen when the test is run (see Fig. 13b).

The current version of MAuto has a predefined box to crop the query image from the click-coordinate and, sometimes, the box size becomes too small to contain a usable number of features. To circumvent this limitation, the user needs to manually crop a better query image from the screenshot to expect enhanced results.



(a) (b)

Fig. 13 Example of bad AKAZE matching

Initially, the user should use native object recognition whenever possible. Indeed, native object recognition is found to be relatively stable, thereby, once available, such an approach should be privileged over MAuto. The latter often takes shortly after the native object recognition step.

When creating the tests, the screenshot operation is quite slow. It can last few seconds to take the screenshot. This means the usability of the application in the browser is not the same as in the device without MAuto. It is also harder to play games through the browser than in the device.

MAuto cannot work with fast-paced mobile games, because it is too slow. It takes relatively too much time to transfer the screenshot from a mobile device to the host. Therefore, it is almost impossible to play fast-paced games with MAuto, because the game can end up in a couple of seconds without new inputs.

Many recording tools do far better when native object recognition can be used in the application. If a native object recognition cannot be used, then MAuto will take over. However, we should notice that it is not possible to give inputs to mobile device sensors through MAuto. This means that it is not possible to test directly games that use sensor data. Although Appium has some support for sensor inputs, MAuto cannot record those inputs.

6 Conclusion

This paper focused on mobile game testing. We reviewed the motivation, key milestones and challenges pervading the development of automated mobile game testing tools. Especially, we highlighted why mobile game testing and test automation are harder than testing traditional mobile applications. One of the key reasons lies in the fact that native object recognition is less applicable to games where the use of additional object recognition methods, like image recognition, is necessary. Besides, the acknowledged *fun factor* renders traditional sequential like approaches quite inefficient. A review of existing technologies revealed five key approaches for mobile game testing: image-based, coordinate-based, OCR/text recognition, native object recognition, and gesture record/replay. Image-based recognition test has shown increased performance, although with limited scope.

Tools to create image-based recognition test scripts have not reached maturity yet and still are under development. This paper has introduced a testing tool, MAuto, to make it easier to create automated mobile game tests. The approach is based on a fruitful combination of AKAZE features, Appium, record and replay, and native object recognition. Evaluation and testing have been conducted using the Clash of Clans game. It is stressed that a good choice of query images is required to make the test stable for production use.

MAuto is a very raw approach to solve challenging mobile automatic testing problems. With some polishing, MAuto would work on slow games, but it does not work with fast games that require rapid user interactions.

MAuto allows the user to create Appium tests with image recognition without coding a single line of code. The main target is to help the developers to test mobile games, but the tool can be used for other application types as well.

As a perspective work, it would be a good idea to make the cropped image size dynamic. At the moment it is a static 10 pixel square around the click-coordinate. When cropping the query image we could calculate the number of features in the image and if there are fewer features than 20 for example, then the algorithm should increase the query image size and calculate the features again until the image has a good amount of features. This would decrease the manual work the user has to do to fix the low-quality query images. It should be quite easy to add iOS support to MAuto as well. Appium works al-ready for Android and iOS. The corner problems are to find a way to take a screenshot from the iOS device and to find a quality VNC client for iOS. The image recognition solution MAuto can therefore easily be extended to an iOS environment.

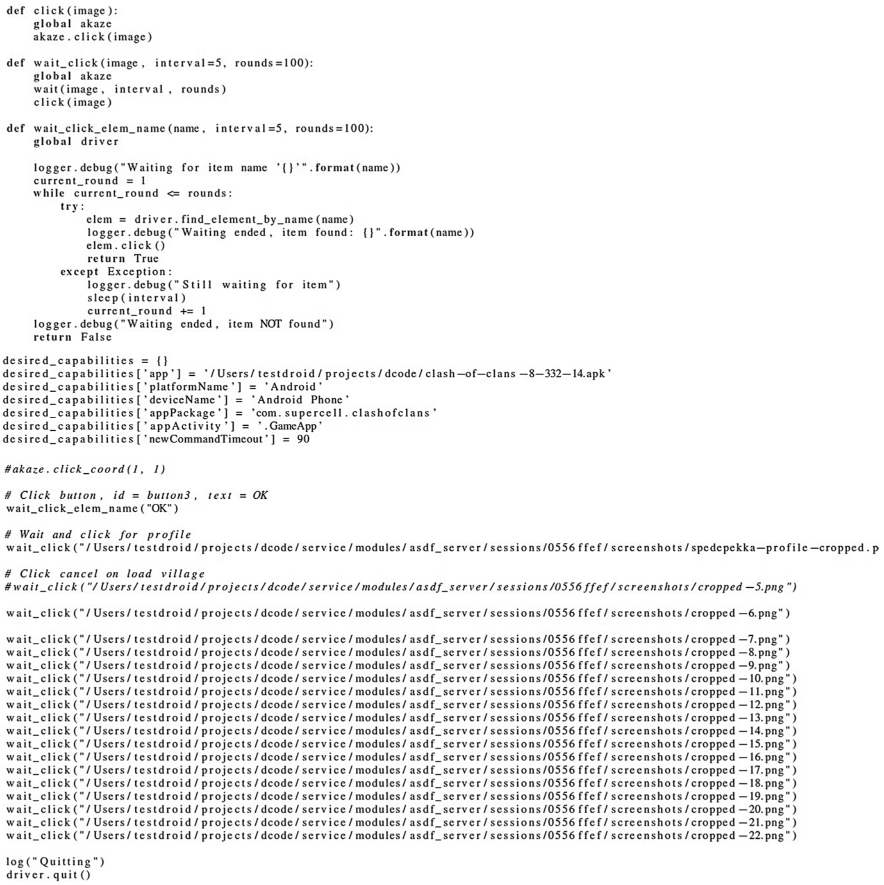
To overcome the slowness of MAuto, especially in the recording phase, one solution could be to compress the image in the mobile device and then send it to the host machine. Another solution consists of tapping into the Android operating system and remove the VNC solution completely. From an input–output perspective, MAuto takes the user inputs from a desktop browser, which is not an ideal way to interact with a mobile device. It would be better for instance to trap the inputs directly from the screen of the mobile device and transfer the clicks and images to the host machine after the test has been recorded. Indeed, if the test recording would be in the mobile device, MAuto might be able to trap the sensor inputs and write those inputs to tests as well.

Acknowledgements Open access funding provided by University of Oulu including Oulu University Hospital. This work is partly supported by EU Horizon 2020 project Cutler (ID # 770469) on Coastal Urban Development through the Lenses of Resiliency and EU YoungRes (#823701) project on Youth polarization.

Compliance with Ethical Standards

Conflict of interest No conflict of interest.

Appendix: Example of Appium Script in MAuto



```
def click(image):
    global akaze
    akaze.click(image)

def wait_click(image, interval=5, rounds=100):
    global akaze
    wait(image, interval, rounds)
    click(image)

def wait_click_elem_name(name, interval=5, rounds=100):
    global driver

    logger.debug("Waiting for item name ’{}’".format(name))
    current_round = 1
    while current_round <= rounds:
        try:
            elem = driver.find_element_by_name(name)
            logger.debug("Waiting ended, item found: {}".format(name))
            elem.click()
            return True
        except Exception:
            logger.debug("Still waiting for item")
            sleep(interval)
            current_round += 1
    logger.debug("Waiting ended, item NOT found")
    return False

desired_capabilities = {}
desired_capabilities[’app’] = ’/Users/testdroid/projects/dcode/clash—of—clans —8—332—14.apk’
desired_capabilities[’platformName’] = ’Android’
desired_capabilities[’deviceName’] = ’Android Phone’
desired_capabilities[’appPackage’] = ’com.supercell.clashofclans’
desired_capabilities[’appActivity’] = ’.GameApp’
desired_capabilities[’newCommandTimeout’] = 90

#akaze.click_coord(1, 1)

# Click button, id = button3, text = OK
wait_click_elem_name("OK")

# Wait and click for profile
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/spedepekka—profile—cropped.p

# Click cancel on load village
#wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—5.png")

wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—6.png")

wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—7.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—8.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—9.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—10.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—11.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—12.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—13.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—14.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—15.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—16.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—17.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—18.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—19.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—20.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—21.png")
wait_click("/Users/testdroid/projects/dcode/service/modules/asdf_server/sessions/0556ffef/screenshots/cropped—22.png")

log("Quitting")
driver.quit()
```

References

Adamsen, C. Q., Mezzetti, G., & Møller A. (2015). Systematic execution of android test suites in adverse conditions. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA) ISSTA'15*, pp. 83–93.

Alcantarilla, P. F., et al. (2013). Fast explicit diffusion for accelerated features in nonlinear scale spaces. In *British Machine Vision Conference*, Bristol, BMVC.

Alcantarilla, P., Bartoli, A., & Davison, A. (2012). Kaze features. In A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, & C. Schmid (Eds.), *Computer vision—ECCV 2012. Lecture Notes in Computer Science* (Vol. 7577, pp. 214–227). Berlin, Heidel-berg: Springer.

Alegroth, E., Nass, M., & Olsson, H. (2013) JAutomate: A tool for system-and acceptance-test automation. In *IEEE sixth international conference on software testing*, *verification and validation (ICST)*, pp. 439–446.

Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., & Memon, A. M. (2012). Using GUI ripping for automated testing of android applications. In *Proceedings of the 27th IEEE/ACM international conference automated software engineering (ASE'12)*, pp. 258–261.

Amalfitano, D., Fasolino, A. R., Tramontana, P., Ta, B. D., & Memon, A. (2014). Mobiguitar—A tool for automated model-based testing of mobile apps. *IEEE Software, 32,* 53–59.

Anand, S., Naik, M., Harrold, M. J., & Yang, H. (2012). Automated concolic testing of smartphone apps. In *Proceedings of the 20th ACM international symposium on the foundations of software engineering (FSE'12)*, North Carolina.

Appium. (2012). *Appium testing framework*. Retrieved September 2019, from http://appium.io.

Azim, T., & Neamtiu, I. (2013). Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the ACM SIGPLAN international conference on object oriented programming systems languages & applications*, *OOPSLA'13*, pp. 641–660.

Calabash. (2012). Calabash testing framework. http://calaba.sh.

Chaffey, D. (2016). *Mobile marketing statistics compilation.* Retrieved September 2019, from Smart Insights: http://www.smartinsights.com/mobile-marketing/mobile-marketing-analytics/mobile-marketing-statistics/.

Chang, T. H., Yeh, T. & Miller, R. C. (2010). GUI testing using computer vision. In *Proceedings of the SIGCHI conference on human factors in computing systems*, *CHI'10* (pp. 1535–1544). New York, NY: ACM. http://doi.org/10.1145/1753326.1753555.

Choi, W., Necula, G., & Sen, K. (2013). Guided GUI testing of android apps with minimal restart and approximate learning. In *OOPSLA'13*, pp. 623–640.

Choudhary, R. S., Gorla, A., & Orso A. (2015). Automated test input generation for android: Are we there yet? (E). In *Proceedings of the ACM/IEEE international conference on automated software engineering*, pp. 429–440.

Cohen, M. (2006). *Agile estimation and planning*. Upper Saddle River: Prentice-Hall.

Crispin, L., & Gregory, J. (2011). *Agile testing A practical guide for testers and agile teams* (7th ed.). Boston: Addison-Wesley.

Esspresso. (2015). *Esspresso testing framework.* https://google.github.io/android-testing-support-library/docs/espresso/.

Fazzini, M., Freitas, E. N., Choudhary, S. R., & Orso, A. (2016). Barista: A technique for recording, encoding, and running platform independent android tests. In *Proceedings of the IEEE international conference on software testing*, *verification and validation (ICST).*

Gomez, L., Neamtiu, I., Azim, T., & Millstein, T. (2013). Reran: Timing- and touch-sensitive record and replay for android. In *Proceedings of the 33th international conference on software engineering (ICSE)*, pp. 72–81.

Google Robo Test. (2015). *Google firebase test lab robo test*. Retrieved September 2019, from https://firebase.google.com/docs/test-lab/robo-ux-test.

Halpern, M., Zhu, Y., Peri, R., & Reddi, V. G. (2015). Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem. In *Proceedings of the IEEE international symposium on performance analysis of systems and software*, pp. 215–224.

Hao, S., Liu, B., Nath, S., Halfond, W., & Govindan, R. (2014). Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys'14*, pp. 204–217.

Hu, Y., Azim, T., & Neamtiu, I. (2015). Versatile yet lightweight record-and replay for android. In *OOPSLA'15*, *ser. OOPSLA 2015* (pp. 349–366). New York, NY: ACM.

Hu, G., Yuan, X., Tang, Y., & Yang, J. (2014). Efficiently, effectively detecting mobile app bugs with app doctor. In *EuroSys'14*, pp. 18:1–18:15.

Iqbal, M. (2019). *App downloaded and usage statistics*, *Business of Apps*, August 7th 2019, https://www.businessofapps.com/data/app-statistics/. Accessed September 2019.

Jabbarvand, R., Sadeghi, A., Bagheri, H., & Malek, S. (2016). Energy-aware test-suite minimization for android apps. In *Proceedings of the international symposium on software testing and analysis (ISSTA)*, pp. 425–436.

Jensen, C. S., Prasad, M. R., & Moller, A. (2013). Automated testing with targeted event sequence generation. In *Proceedings of the international symposium on software testing and analysis (ISSTA)*, pp. 67–77.

Knott, D. (2015). *Hands-on mobile app testing: A guide for mobile testers and anyone involved in the mobile app business* (1st ed.). New York: Addison-Wesley Professional.

Kochhar, P. S., Thung, F., Nagappan, N., Zimmermann, T., & Lo, D. (2015). Understanding the test automation culture of app developers. In *Proceedings of the 8th IEEE international conference on software testing*, *verification and validation*, Austria, ICST'15.

Lin, Y., Rojas, J. F., Chu, E., & Lai, Y. (2014). On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering, 40*(10), 2014.

Linares-Vásquez, M., Moran, K., & Poshyvanyk, D. (2017). Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *Proceedings of the 33rd IEEE conference of software maintenance and evolution (ICSME).*

Machiry, A., Tahiliani, R., & Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 9th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pp. 224–234.

MacKenzie, B. (2012). *Top 10 mobile application testing automation tool requirements*. Retreived April 7, 2018 from http://northwaysolutions.com/blog/top-10-mobile-application-testingautomation-tool-requirements/.

Mahmood, R., Mirzaei, N., & Malek, S. (2014). EvoDroid: Segmented evolutionary testing of android apps. In *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'14)*, pp. 599–609.

Mao, K., Harman, M., & Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th international symposium on software testing and analysis*, pp. 94–105.

Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.

Mirzaei, N., Bagheri, H., Mahmood, R., & Malek, S. (2015). Sig-droid: Automated system input generation for android applications. In *ISSRE'15*, pp. 461–471.

Monkey Testing. (2016). *Android ui/application exerciser monkey*. Retrieved September 2019, from http://developer.android.com/tools/help/monkey.html.

Moran, K., Bonett, R., Bernal-Cárdenas, C., Otten, B., Park, D., & Poshyvanyk, D. (2017). On-device bug reporting for android applications. In *Proceedings of the 4th IEEE/ACM international conference on mobile software engineering and systems (MobileSOFT).*

Moran, K., Linares-Vásquez, M., Bernal-Cárdenas, C., Vendome, C., & Poshyvanyk, D. (2017). Crashscope: A practical tool for automated testing of android applications. In *ICSE'17 Companion*, pp. 15–18.

Newzo. (2018). *Global Game Market Report*, see sample by Tom Wijman at https://newzoo.com/insights/articles/global-games-market-reaches-137-9-billion-in-2018-mobile-games-take-half/. Accessed June 2018.

Novak, J. (2008). Game development essentials: An introduction. *Game development essentials series*, Thomson/Delmar Learning.

Qmetry. (2016). *Qmetry test automation framework*. https://qmetry.github.io/qaf/.

Quantum. (2012). *Quantum*. https://community.perfectomobile.com/posts/1286012-introducing-quantum-framework.

Ranorex. (2017). *Ranorex testing framework*. http://www.ranorex.com.

Ravindranath, L., Nath, S., Padhye, J., & Balakrishnan, H. (2014). Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on mobile systems*, *applications*, *and services*, *MobiSys'14* (pp. 190–203). New York, NY: ACM. http://doi.org/10.1145/2594368.2594377.

Roboelectric, (2012). *Roboelectric testing framework*. Retrieved September 2019, from http://roboelectric.org.

Robotium. (2010). *Robotium testing*. Retrieved September 2019, from https://github.com/RobotiumTech/robotium.

Robotium Recorder, (2014). *Robotium recorder*. https://github.com/RobotiumTech/robotium.

Sasnauskas, R., & Regehr, J. (2014). Intent fuzzer: Crafting intents of death. In *Proceedings of the Joint 12th International Workshop on Dynamic Analysis (*WODA*) and Workshop on Software and System Performance (WODA & PERTEA)*, pp. 1–5.

Sonders, M. (2017). *New mobile game statistics every game publisher should know in 2016.* Retrieved from Survey Monkey: https://www.surveymonkey.com/business/intelligence/mobile-game-statistics/. Accessed August 2017.

Ui-Automation iOS. (2015). *Apple ui-automation documentation*. Retrieved September 2019, from https://web.archive.org/web/20140812195854/https://developer.apple.com/library/ios/documentation/DeveloperTools/Reference/UIAutomationRef/_index.html.

Uiautomator. (2018). *Android uiautomator*. http://developer.android.com/tools/help/uiautomator/index.html.

Van der Merwe, H., Van der Merwe, B., & Visser, W. (2014). Execution and property specifications for jpf-android. *SIGSOFT Software Engineering Notes, 39*(1), 1–5.

White, M., Linares-Vásquez, M., Johnson, P., Bernal-Cárdenas, C., & Poshyvanyk, D. (2015). Generating reproducible and replayable bug reports from android application crashes. In *ICPC'15*.

Xamarin. (2018). Retrieved September 2019, from https://www.jimbobbennett.io/ui-testing-your-xamarin-apps/.

Yang, W., Prasad, M., Xie, T. (2013). A grey-box approach for automated GUI-model generation of mobile applications. In *Proceedings of the 6th international conference on fundamental approaches to software engineering (FASE)*, pp. 250–265.

Yeh, T., Chang, T. H., & Miller R. C. (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* (pp. 183–192). New York, NY: ACM. http://doi.org/10.1145/1622176.1622213.

Zaeem, R. N., Prasad M. R., & Khurshid S. (2014). Automated generation of oracles for testing user-interaction features of mobile apps. In *ICST'14*, pp. 183–192.

Zhang, H., & Rountev, A. (2017). Analysis and testing of notifications in android wear applications. In *Proceedings of the 39th international conference on software engineering (ICSE)*, May 2017.