Automated Testing of Mobile Applications Using a Robotic Arm

Demian Frister
Karlsruhe Institute of Technology (KIT)
AIFB - BIS
Karlsruhe, Germany
Demian.frister@kit.edu

Andreas Oberweis
Karlsruhe Institute of Technology (KIT)
AIFB – BIS
Research Center for Information Technology (FZI)
Karlsruhe, Germany
Andreas.oberweis@kit.edu

Aleksandar Goranov
Research Center for Information Technology (FZI)
Karlsruhe, Germany
goranov@fzi.de

***Abstract*—— Recent developments in mobile testing have raised the importance of black-box testing and the usage of automated test procedures. In order to secure a flawless user experience, developers are required to develop error free applications. The realization of mobile tests with the support of robotic equipment ensures new possibilities to run tests without further human interaction. In this paper we discuss different approaches for testing mobile applications with robotic arms and additionally share our insights on a prototype suited to automatically test mobile applications. The implemented prototype can perform black-box tests by utilizing an algorithmic approach based on tree-search.**

Keywords: robotic testing, mobile application testing, black-box-testing, testing automation

I. INTRODUCTION AND MOTIVATION

Despite the ever-growing share of mobile devices on the world wide web traffic [1], mobile applications are often still tested by traditional verification and testing methods [2]. These traditional methods do not take into consideration that mobile devices are used quite differently than applications on stationary systems. Whereas traditional applications are used with mouse and keyboard, mobile devices are operated by touch inputs on a touch screen. Most current testing systems available for developers, lack the ability to test the GUI of a mobile application fully autonomously and with the replicated authenticity and realism of touching the device screen of the device under test (DUT). To test the software behavior of a wet device or with gloves, requires the use of special test series to stay reproducible and consistent. Furthermore, the variety of device sizes, types and touch sensibility makes it difficult to produce consistent tests. Testing the GUI in environments hostile to humans (at prolonged exposure) might be impossible to achieve. This prevents tests under extreme conditions like high humidity or temperature. We found two promising systems which presented testing of mobile applications with a robot, but they mostly use specialized tools and hardware. Due to these limitations, testing of mobile applications is based on manual testing in most cases which is inefficient and costly compared to autonomous testing [2, 3]. This paper presents an approach using a standard industrial robotic arm for testing mobile Android apps. Furthermore, this approach enables black box testing of applications without ownership of the source code. In addition, we propose a testing sequence to automate the process. Unlike other testing methods for Android, this method does not need code or system access besides the Android Debug Bridge (ADB) between a computer and the DUT. This approach focuses on opening all activities an Android app contains and logging the whole procedure with screenshots. Additionally, a generated execution tree makes it possible to retrace every activity on the device and every action that was performed by the robot. By using a robotic arm and simulating the touch on the application UI errors (frontend dependent) can be found, which depend on using real haptic touch. The prototype was implemented by using open source frameworks. Due to their usage, the developed toolset could be built hardware agnostic and easily adaptable to different robots or algorithms for further development. The prototype was built for and tested on Android applications because of their open nature. To evaluate the implementation a simple Android application was created, utilizing the commonly used elements of a typical Android application.

This paper is structured as follows: Section II gives an overview of testing with a focus on GUI testing. Section III introduces related work with two similar robotic test systems and one automated testing tool for Android devices. In section IV the paper presents the tools used for the prototype and their interaction. Section V displays the implementation and architecture of the prototype and Section VI gives a short evaluation of the advantages and disadvantages of using the prototype. The paper closes with Section VII, which discusses future work and possible extensions of the prototype.

II. BACKGROUND

Testing in Software engineering is defined as a systematic approach to examine a program to gain trust in its correct implementation of all requirements and its reaction to errors. It's intention is to find errors [4]. To find errors behavior, the program is challenged into situations where the program deviates from its requirements. An error is defined as the failure of a function or action to perform its intended purpose. To find errors, a test must be systematically planned and executed. Afterwards the results need to be evaluated and documented to give practical results [5].

One way to set apart testing methods is to distinguish between white-box and black-box testing. White-box tests are tests where the internal structure of the program is known to the tester. Usually an approach like this is used to test the formal requirements and the internal data flow. Unlike white-box tests, black-box tests do not necessarily know about the internal program structure. Black box tests obtain their test data from the specified functional requirements lacking the final program structure [6]. This enables testing of the documented requirements rather than the code implementation. Since it is not always feasible to test all possible test scenarios, finding the right test cases (i.e. cases that throw an error) is more important [4]. One of the requirements of the prototype created was the implementation of a test method for black-box testing. This requirement facilitates testing of applications not written or owned by the tester.

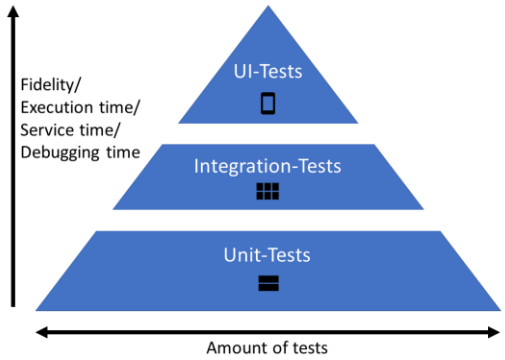

Figure 1 Testing pyramid [7]

A useful classification for mobile application tests is the testing pyramid shown in Fig. 1 [8]. With each layer of the pyramid, the test effort increases due to higher complexity and dependencies. The top of the pyramid consists of UI-Tests because they depend on all integrations below and each functional unit must work for itself and in integration with others before they are possible. Furthermore, UI-Tests cannot be performed alone. Therefore the amount of testcases increases exponentially in proportion to the amount of widgets on the UI [9]. Due to this errors found during integration-testing, typically cost five times more than errors found by Unit-Tests [10]. As a result of these limitations, it is recommended to develop an application with an iterative approach and check whether all basic unit tests pass after every added function [11]. Therefore, automating GUI-Tests could reduce the costs for software development.

GUI-Testing consists of tests which assess multiple functions working together in a closed workflow. According to Googles testing fundamentals they should make up around 10% of the full test count [11]. On the one hand they should follow a sensible sequence, and on the other hand, they should cover a wide area of functionalities. This means that they must follow a special sequence of steps to get to a certain screen. With every screen to reach, the amount of potential test cases and GUI states increases [9].

GUI-Tests are often prepared and executed through *Capture-and-replay* methods. They first capture a potential test sequence, which can be automatically replayed whenever a change in the application occurs. Then the as-is-condition is compared to the defined reference. Testing like this takes less time than testing everything by hand but still requires a capture phase and it is possible a developer might overlook or forget to test parts of a program.

A. *Comparable testing methods for Android applications*

We identified three main methods to test Android applications and especially the GUI. The first method is testing an application on an emulator or on a device by manually testing all functionalities. This can be cheaper than other test methods if done in the beginning of a software project [12]. To use this method the software should be given to a person not involved in developing the application. This test method is not necessarily complete and can only determine the existence of errors, not their nonexistence. For Android this is done either on an emulator that can be used to cost-effectively test different screen sizes and specifications or directly on a physical device.

The second frequently used method involves "*espresso tests*" [13, 14]. Espresso is a test framework from Google which allows the writing of GUI-Tests for Android. A test case must be written for each GUI element which then can be executed automatically after each new implementation. It is also possible to use espresso tests with the capture-and-replay method by recording a sequence of interactions with an application. The corresponding test code is generated automatically and can then be replayed afterwards. Espresso tests can be executed on an emulator as well as on a physical device. In contrast to test users, espresso tests are more systematic in assessing an application.

The third method that is increasingly used to test Android applications is Google's cloud-based test framework "Firebase Test Lab". By using the "Firebase Test Lab" developers eliminate the necessity to own a multitude of physical devices because they can test their applications on real devices in the cloud. It also allows to execute espresso tests as well as other pre captured test scenarios to be run on the device. One intriguing method of automatically testing is the *robo test* [15]. With *robo tests* an application can be tested by a software agent on multiple devices

automatically. The software agent tests different elements on the application without requiring the developer to define them. Nevertheless, these tests only simulate touches on the device screen without actually touching the device and therefore lack authenticity and realism.

III. Related Work

There are two systems considered for the implementation in this paper, where robotic systems with a multi-purpose approach were used to test mobile applications by touching the display instead of using digital methods. Both are black box testing methods. These systems were chosen for comparison because they both use a new approach to testing mobile applications with a hardware robot.

One is *Tappy* a low cost delta-robot [16]. Delta robots use a technology also used in 3D-printers. Delta-robots are characterized by high speed and high accuracy in a 2-dimensional plane but low adaptability to new contexts. They are usually used for pick and place activities. Tappy must be calibrated to the x-, and y-positions of the testing device. Then the system utilizes the *capture-and-replay* method. By making use of the ADB screenshots are taken, and the testing person selects elements in the app, that the system should test. The testing person must therefore enter the complete testing sequence which is saved and can be repeated by the robot. During this process elements can be forgotten by the tester due to lack of concentration or instructions, which is a disadvantage.

The second system examined is *Axiz*. Axiz is more than just the robotic system itself but a complete framework for automatically testing Android applications [3]. The goal of this system is to use physical robots to perform more realistic black-box tests. The framework consists of two main parts. An autonomous test generator and a test executor. As an additional abstraction layer, the test generator uses a camera without any further connection to the smartphone to detect possible testable elements. The test cases are generated by reusing and extending realistic test cases created by traditional test methods. The testing robot is a four-axis robotic arm also usable for tasks other than testing. The robot was built by using inexpensive commodity hardware.

For testing the UI of Android Applications on a device or an emulator with digital input (i.e. by simulating clicks on the device screen) there is a toolset called *GUIRipper* which uses the Android ripper technique [17, 18]. This tool and technique enable autonomous testing of GUIs by reading all elements and creating an ordered hierarchy of all elements. By comparing the old state of a GUI to the state after an event was fired (e.g. a button was clicked) possible interactions are found and put into a task-list. This task-lists is updated with every new event while exploring the application simultaneously and iteratively. The implementation introduced in this paper uses a similar approach for iterating over the application.

IV. Tools used

A. Franka Emika Panda 7-DOF Robotic Arm

Robotic arms show higher flexibility for testing mobile applications in closed space than for example Tappy. The human arm has 7-DOF [19]. Robotic arms with 7-DOF define the minimum to reproduce the human motion scope. It is still recommended to use more than 7-DOF for natural looking, humanlike movements [20]. This is due to their motor-powered joints instead of muscles [21]. For the implementation in this paper the robotic arm *Panda* from the company Franka Emika (shown in Fig 3 on the left) was used. This industrial robot offers a path deviation of around 0.1 mm which makes it possible to reach elements on a mobile device with a high accuracy and low deviation between test executions and thus high repeatability. This accuracy makes it possible to reach even tiny UI elements which would be difficult to reach consistently even for humans. Through its sufficiently large workspace and humanlike motion abilities, the arm is capable of testing mobiles devices without further preparations [22]. It can be adapted for a multitude of different use cases without needing a new robot or toolset. The robotic arm uses a parallel gripper as an end effector which allows the use of a capacitive pen for touching the mobile device. By utilizing the standard gripper and a capacitive pen, no additional retooling is required, and the robot remains flexible. Due to the panda's high force sensibility it is possible to detect if a touch on the DUT-surface was strong enough for a successful interaction. The Panda is a collaborative robot with many sensors for working with humans without the need for fences or additional security measurements.

B. Communication with the robotic arm

The robotic arm communicates over TCP/IP. It can connect with the robotic operating system (ROS). ROS is a software framework for robots. One of the main goals of ROS is to achieve a basic toolset that can be abstracted to different tasks [23]. It offers standard capabilities a robot could need in a modular and open fashion. With ROS it is possible to communicate with different systems or sensors on different hardware. The capsulated architecture of ROS allows every function to run as a *node* on its own system and if one system is substituted for another (e.g. another robotic arm) it requires less adaptions to achieve the same goal. Each node can communicate with other nodes via ordered message exchanges. Messages are defined and use an agreed upon format. One tool ROS offers is *MoveIt!*. MoveIt! presents the ability to convert a position in cartesian space (i.e. x-, y-, z-axis) to suitable joint angles. This allows sending a position in the real world, the end effector should move to. MoveIt! uses different inverse kinematic calculations to reach this point [24]. It can be adapted to different robotic arms with different joint counts.

C. *Extracting the GUI*

The different elements on a DUT GUI are extracted by utilizing the ADB-Bridge with a tool called *UIAutomator* and a python wrapper for it [25, 26]. By reading the UI-elements (visible and invisible on the DUT screen) this tool generates a hierarchy of the GUI and writes it in an XML file. The created XML allows easy iteration through the different surface elements with additional information about the elements (e.g. “clickable”, “scrollable” or “focused”). By iterating through the elements, it is possible to build a tree structure of an application and follow its activities. The use of a tree structure enables using tested algorithms such as depth-first search or breadth-first search to visit the different nodes (i.e. elements and activities of an application). This method is only able to import applications written in native Android. Non-native applications will not output any elements on the GUI. *UIAutomator* also allows the simulation of different User Interactions (e.g. Clicks, scrolls and swipes) on a DUT. With this feature the testing sequence algorithm was tested without the need to execute every test on the robotic arm. Furthermore, it can take screenshots of the DUT. Since *UIAutomator* only uses the Android debug bridge, it can execute black-box tests.

D. *Test Application*

To test the implementation, a simple Android application was created. In addition, several activities were written to simulate a typical Android application. An activity is an in itself completed set of functions that creates a window where different UI elements can reside [27]. Jumping between is possible through interaction with screen elements. The activities in the test application exist with different objectives. Each activity implements some typical elements Android uses, such as lists, sidebars, input fields, buttons and touch sensitive items. From the main activity different subcategories and activities can be opened. The applications elements were chosen because Android studio recommends them as the most commonly used elements.

V. IMPLEMENTATION

The proposed implementation of an application testing follows several (predefined) steps. It starts with the *teach-in* phase, during which the robotic arm has to be set up to interact with the DUT. In the second phase (*UI-reading-phase*) the display content of the DUT must be read. During the third phase *(selection-phase)* relevant, testable elements are to be found and added to a stack. In order to perform the test an efficient test sequence has to be calculated in the fourth phase *(test-phase)* and finally executed in the last fifth phase *(execution-phase)*.

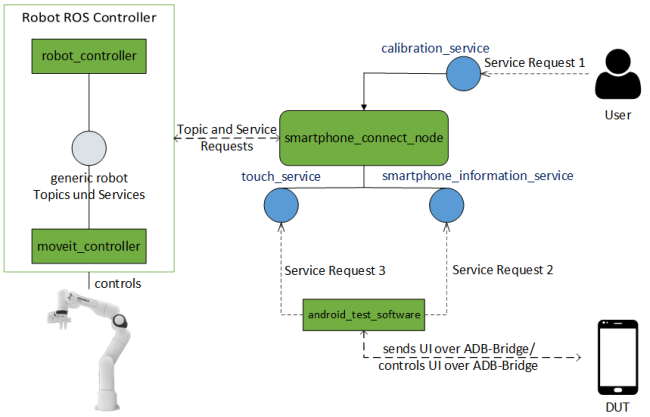

Figure 2 Components in the prototype

Fig. 2 shows a simplified version of the components relevant to the task at hand. The left block implements all modules necessary for controlling the robot. It uses modules created by the robot manufacturer. They are accessible over standard ROS topics. It also shows the usage of MoveIt! for calculating the inverse kinematic. Because of this modular architecture, the methods of this paper can be applied to any other robotic arm connected with ROS. The “*smartphone_connect_node*” can interact over ROS topics with other ROS controlled robotic arms and communicates over ROS topics with the “*Android_test_software*”.

To test and interact with the DUT, a teach-in must be done in the *teach-in*. This is because the robot does not have any information about the location of the DUT, nor (more importantly) the elements on the device screen in particular. With this teach-in, the robot must be calibrated to know the exact positions, every time a new device is introduced to the robotic arm. The teach-in is also implemented in the “smartphone_connect_node” and it is called over typical ROS service requests. Whereas the robot exists in a three-dimensional world, the DUT screen localizes its elements in a two-dimensional space. This is shown in Fig. 3 on the left picture. The left coordinate system denotes the robot world coordinates and the right one shows the DUT coordinates. There are various methods for adapting a robot system to a DUT, for example the method used by Tappy [16]. It requires an application to run on the DUT while the DUT must be placed at a specific point in front of the robot. Then the robot tests different points on the application and aligns itself according to the feedback it gets (i.e. which position on the screen translates to which robot end effector position).

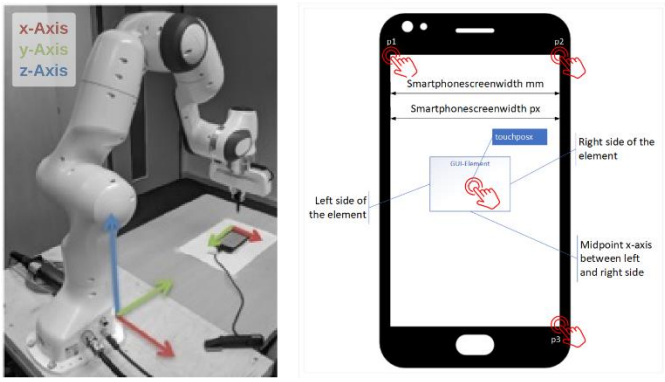

Figure 3 the robotic arm with the DUT in front and their coordinate systems. Right: A DUT with the initial teach-in points the robot learns

The *teach-in* proposed in this paper consists of leading the robot with a capacitive pen as its end effector to three different points (p1, p2 and p3) on the DUT screen shown in Fig. 3. Because the points consist of the left upper, right upper, and right lower position of the screen, they can be used to calculate the width and height of the DUT after saving them and applying the Pythagoras' theorem. This ensures that a device can even be in an inclined position. The robot is also shown the position of the "back" button of a device because they can vary among devices. The exact position of a GUI element is given over the ADB-interface in pixel coordinates for the left, right, upper, and lower side of an element. This can be transferred to the robot's world space in meters by calculating the center of an element. By utilizing the framework MoveIt! to calculate the inverse kinematic between the touch point and the robotic arm joints, it is possible to send Cartesian coordinates indicating the position on the DUT the robot should touch. This method does not require any additional software to run on the DUT and is easy to implement. It is also independent from display sizes or other screen characteristics.

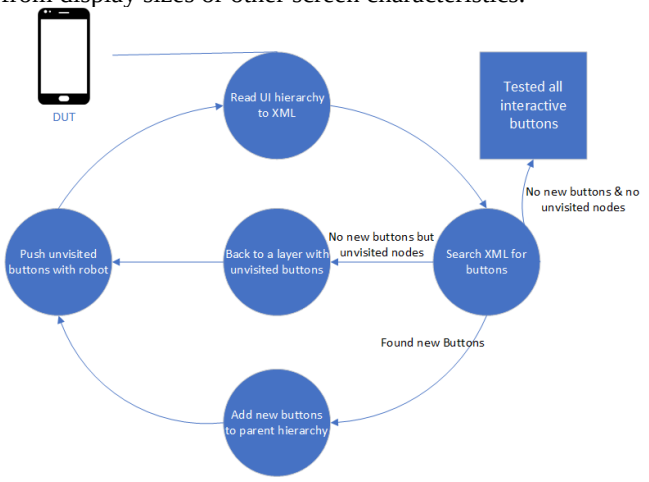

Figure 4 Sequence to build the test tree

The *UI-reading-phase, selection-phase and testing-phase* are done by the "Android_test_software" in the lower right corner of Fig. 2 and are completely autonomous. This node calculates the test sequence shown in Fig. 4. To start the sequence, the GUI is extracted as described in Paragraph C. The used method is based on a few principles shown by most Android applications [28]. Many elements of the type *button* lead to new activities. To go back one step in the activity hierarchy, the *back button* of the DUT is used. Elements other than buttons often create a new state in the same activity instead of leading to a new activity. On this basis, the average flow through an application can be described as an acyclic graph where each button leads to a new node. A tree is a special kind of acyclic, connected graph. Every node can be accessed from the trees root and there are no cycles inside. This ensures that the robot does not end up in a loop. It is not possible to connect activities that are not in a parent-child relationship, which is not necessary, because every activity can still be accessed from its parent. With every touched element (by the robotic arm), the application either opens a new activity with an entirely new GUI or new elements appear. The created prototype only registers buttons on the screen, but it would require little additional work to expand its capabilities to gather different elements. The robot can simulate swipes as well. The gathered hierarchy is scanned for new unvisited nodes (e.g. new buttons which were not touched before) and adds them to a tree. If there are no more unvisited nodes, the whole application has been tested and the test is concluded. If new elements are found, they are added to the current node and the next unvisited node is tested. If the current layer in the tree has no unvisited nodes, the back button is pushed until the application is in an activity with unvisited nodes. This is repeated until there are no untested elements. To test all saved tree nodes (i.e. activities) and simultaneously discover new ones, a search algorithm based on tree search algorithms is used. This is shown by the example test application in Fig. 5. Two algorithms are possible to search through a tree. Depth-first and breadth-first. Since not all nodes are known at the beginning of the search, a mixture between the two algorithms was chosen. Compared to depth-first, breadth-first is not complete for traversing a finite graph without testing for cycles. For finite graphs breadth-first is complete compared to depth-first, which (if not tested for cycles) is not. This ensures every node is found. To save time by not going back too many times, all buttons in an opened activity are tested first, before going back. In Fig. 5, the step in which a node was found is annotated in red and the step in which it as tested is blue.

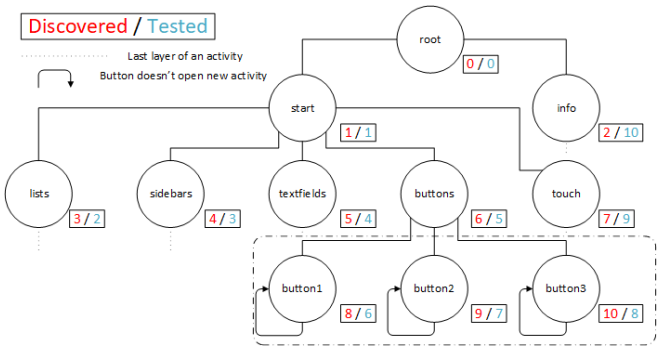

Figure 5 Example algorithm testing the Test application

After each test step a screenshot is taken, and a visual cyclic graph is generated to document the test procedure. This makes it possible to retrace which element leads to a program error or unexpected behavior. Additionally, the full tree is saved as a graph to follow the test sequence.

VI. Discussion

The proposed approach is suited for automated testing of mobile applications by using a robotic arm with minimal human supervision and interaction during tests. This approach offers several advantages over other test methods for mobile applications. The initial resources and expenses needed to run the testing of mobile applications with a robotic arm contain higher acquisition costs, however the long-term variable costs (e.g. labor) are lower compared to other methods. After the robot has learned how to handle a specific DUT it is able to test any (written in native Android) application it receives. In contrast to cloud based autonomous testing tools (e.g. Firebase Cloud Testing) the method can only test one application and DUT at a time but compared to manual testing the robot can test an application in a consistent and tireless manner. It is also able to test a multitude of different devices of different shapes without changing the hardware. A large part of the software costs are labor costs for developers and test users. If the costs and work required for tests can be lowered, the developers are free to do the more creative software developing work than testing. Additionally, specially trained labor necessary for e.g. espresso tests, can be avoided. Using the robot can also enable testing scenarios that were not possible before (e.g. in hazardous environments) or which required additional arrangements. Furthermore, the tests are also more realistic compared to digital testing scenarios. Whereas other testing methods might offer more additional information why an application crashed, the information given by the proposed prototype should give a developer enough information to retrace the steps that lead to it, reproduce it and rectify it. In addition, it does not rely on code access to give this information, unlike other methods.

VII. Future Work

Using an industrial robotic arm when testing mobile applications, offers the opportunity to access known and tested robotic functions via ROS. Additionally, it allows for readjustment of the testing environment for different DUTs without much additional work. The tests executed are carried out under highly realistic circumstances, as opposed to merely digital tests, and can be carried out with various environmental requirements.

Future work could implement the exploration strategy of the GUIRipper tool [17, 18] for further capabilities. This would also allow tests to search for additional UI elements and objects. Whereas other papers looked at using deep learning to gain a higher understanding of the GUI [29] we plan to implement a solution based on reinforcement learning [30]. A software agent should achieve "curiosity" about an application and try to see all possible actions and screens [31]. With further development, it should also be possible to process already written espresso tests on the robot. It is also planned to use a robotic arm for testing traditional software running on a computer with a touch screen.

References

[1] StatCounter. "Percentage of all global web pages served to mobile phones from 2009 to 2018: Digital in 2018." Statista. https://www.statista.com/statistics/241462/global-mobile-phone-website-traffic-share/ (accessed 10.09.2020, 2020).

[2] J. Gao *et al.*, "Mobile Application Testing: A Tutorial," *Computer*, vol. 47, no. 2, pp. 46-55, 2014, doi: 10.1109/mc.2013.445.

[3] K. Mao *et al.*, "Robotic Testing of Mobile Apps for Truly Black-Box Automation," *IEEE Software*, vol. 34, no. 2, pp. 11–16, 2017, doi: 10.1109/ms.2017.49.

[4] G. J. Myers *et al.*, *The art of software testing*, 3 ed. Hoboken, NJ: Wiley (in eng), 2012, p. 240.

[5] P. Liggesmeyer, *Software-Qualität: Testen, Analysieren und Verifizieren von Software*, 2. ed. Heidelberg: Spektrum Akademischer Verlag (in ger), 2009.

[6] W. E. Perry, *A Standard for Testing Application Software*. Auerbach Publishers, 1989.

[7] M. Fowler, "The Practical Test Pyramid: The Test Pyramid," ed, 2018.

[8] M. Cohn, *Succeeding with agile: Software development using Scrum* (The Addison-Wesley signature series A Mike Cohn signature book). Upper Saddle River, NJ: Addison-Wesley (in eng), 2010, p. 475.

[9] A. M. Memon *et al.*, "Using a goal-driven approach to generate test cases for GUIs," in *International Conference on Software Engineering: ICSE 99* New York, N.Y, B. Boehm, D. Garlan, and J. Kramer, Eds., 1999: ACM, pp. 257–266, doi: 10.1145/302405.302632.

[10] J. O. Paul Ammann, *Introduction to Software Testing*, 2 ed. Cambridge University Press, 2016.

[11] Google. "Fundamentals of Testing." https://developer.android.com/training/testing/fundamentals (accessed 19.03, 2020).

[12] R. Ramler and K. Wolfmaier, "Economic perspectives in test automation," in *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, H. Zhu, J. R. Horgan, S. C. Cheung, and J. J. Li Eds. New York, NY, USA: ACM Press, 2006, p. 85.

[13] T. Lämsä, "Comparison of GUI testing tools for Android applications," University of Oulu, 2017.

[14] Google. "Espresso | Android Developers." https://developer.android.com/training/testing/espresso/ (accessed 11.09, 2020).

[15] Google. "Firebase Test Lab Robo Test | Firebase." https://firebase.google.com/docs/test-lab/android/robo-ux-test (accessed 11.09, 2020).
[16] Testdevlab. "How we built a robot for automated manual mobile testing." https://www.testdevlab.com/blog/2017/07/how-we-built-a-robot-for-automated-manual-mobile-testing/ (accessed 11.09, 2020).
[17] D. Amalfitano *et al.*, "A toolset for GUI testing of Android applications," in *International Conference on Software Maintenance (ICSM)*. Piscataway, NJ: IEEE, 2012, pp. 650–653.
[18] D. Amalfitano *et al.*, "Using GUI ripping for automated testing of Android applications," in *International Conference on Automated Software Engineering - ASE 2012*, 2012-01-01 2012: ACM Press, doi: 10.1145/2351676.2351717. [Online]. Available: http://www.cs.umd.edu/~atif/papers/AmalfitanoASE2012.pdf
[19] M. Benati *et al.*, "Anthropomorphic robotics," *Biological Cybernetics*, vol. 38, no. 3, pp. 125–140, 1980, doi: 10.1007/bf00337402.
[20] T. Asfour and R. Dillmann, "Human-like motion of a humanoid robot arm based on a closed-form solution of the inverse kinematics problem," in *International Conference on Intelligent Robots and Systems: (IROS 2003)*, Piscataway, NJ, 2003: IEEE, pp. 1407–1412, doi: 10.1109/iros.2003.1248841.
[21] M. J. Matarić, *The robotics primer* (Intelligent robotics and autonomous agents series). Cambridge, Mass: MIT Press (in eng), 2007, p. 306.
[22] FrankaEmika. "Datasheet Panda." https://s3-eu-central-1.amazonaws.com/franka-de-uploads/uploads/Datasheet-EN.pdf (accessed 10.09., 2020).
[23] M. Quigley *et al.*, "ROS: an open-source Robot Operating System," in *ICRA workshop on open source software*, vol. 3, 2009.
[24] S. Chitta *et al.*, "Moveit![ROS topics]," *IEEE Robotics & Automation Magazine*, vol. 19, no. 1, pp. 18–19, 2012.
[25] *uiautomator*. (2018). Accessed: 11.09.2020. [Online]. Available: https://github.com/xiaocong/uiautomator
[26] Google. "UI Automator | Android Developers." https://developer.android.com/training/testing/ui-automator (accessed 10.09, 2020).
[27] Google. "Activity | Android Developers." https://developer.android.com/reference/android/app/Activity (accessed 11.09, 2020).
[28] Google. "Principles of navigation." https://developer.android.com/guide/navigation/navigation-principles (accessed 11.09, 2020).
[29] T. Zhang *et al.*, "Deep Learning-Based Mobile Application Isomorphic GUI Identification for Automated Robotic Testing," *IEEE Software*, vol. 37, no. 4, pp. 67-74, 2020, doi: 10.1109/ms.2020.2987044.
[30] D. Adamo *et al.*, "Reinforcement learning for Android GUI testing," 2018: ACM Press, doi: 10.1145/3278186.3278187. [Online]. Available: https://dx.doi.org/10.1145/3278186.3278187
[31] S. Still and D. Precup, "An information-theoretic approach to curiosity-driven reinforcement learning," *Theory in Biosciences*, vol. 131, no. 3, pp. 139-148, 2012/09/01 2012, doi: 10.1007/s12064-011-0142-z.