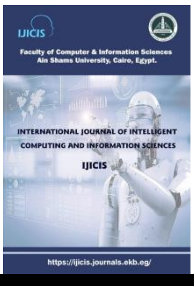
IJICIS, Vol.22, No.2, 126-136
DOI: 10.21608/ijicis.2022.98138.1124

International Journal of Intelligent Computing and Information Sciences

https://ijicis.journals.ekb.eg/



A SURVEY ON AUTOMATED USER INTERFACE TESTING FOR MOBILE APPLICATIONS

Amira Samir [*]
Department of Information Systems, Faculty of Computer and Information Sciences, Ain Shams University
Cairo 11566, Egypt
amira_samir@cis.asu.edu.eg
https://orcid.org/0000-0003-4641-0585

Huda Amin Maghawry
Department of Information Systems, Faculty of Computer and Information Sciences, Ain Shams University
Cairo 11566, Egypt
huda_amin@cis.asu.edu.eg
https://orcid.org/0000-0001-5550-5717

Nagwa Badr
Department of Information Systems, Faculty of Computer and Information Sciences, Ain Shams University
Cairo 11566, Egypt
nagwabadr@cis.asu.edu.eg
https://orcid.org/0000-0002-5382-1385

Received 2021-09-26; Revised 2022-04-27; Accepted 2022-04-28

Abstract: *Nowadays, smartphones play a remarkable role in our lives. Testing mobile applications is significant to guarantee their quality. Automated testing is applied to minimize the cost and the interval of time instead of manual testing. There are different testing levels which are unit testing, integration testing, system testing and acceptance testing. Automated mobile application testing type methodologies are categorized into white-box testing, black-box testing and grey-box testing. Besides, there are several testing types such as functional testing and non-functional testing. Most of the existing studies focus on user interface testing which is type of functional testing. In this paper, testing approaches for user interface testing through different existing studies from 2013 to 2021 have been surveyed. Those approaches are classified into model-based testing, model learning testing, search-based testing, random-based testing, and record & replay testing. Several essential issues related to those approach such as the optimization and redundancy for generation of test suites have been mentioned. Finally, challenges in automated mobile applications user interface testing have been discussed.*

Keywords: *mobile application testing, user interface testing, functional testing, system testing, black-box testing*

[*] Corresponding author: Amira Samir
Department of Information Systems, Faculty of Computer and Information Sciences, Ain Shams University Cairo 11566, Egypt
E-mail addresses: amira_samir@cis.asu.edu.eg
ORCID: https://orcid.org/0000-0003-4641-0585

1. Introduction

Today, technology has a great influence on the world. It has helped in the prediction and detection of severe diseases as lung cancer [1, 2] and the diagnosis of Corona virus disease 2019 (COVID-19) [3]. It has also helped in connecting the world through smartphones that have played a vital role in our daily life. Their applications are used in different fields to make life easier for users. For example, mobile payment is one of the quickest ways of electronic payment [4]. Another example is mobile learning that helps in the effectiveness of education through simplifying the connection between teachers and students [5]. Testing is essential when constructing a mobile application to guarantee its quality. It requires too much time to be performed manually. That is the reason of the presence of automated testing for mobile applications. It minimizes the cost and time interval of testing [6]. Mobile application testing could be performed using testing environments which are emulators and real devices. Emulator is a virtual machine simulating real device [7]. The same testing approach could use both emulator and device. The studies at mobile application testing have focused on several points: Optimization of the generated test suites to increase the performance to testing [8–10], redundant exploration of the Application Under Test (AUT) [11–14], generation of user interface events only without support to system events [12, 15].

In this paper, the automated user interface testing approaches for mobile applications have been surveyed through different existing studies from 2013 to 2021. In the next section, the testing types have been presented, they are divided into functional testing and non-functional testing. Then, existing studies of the mobile testing levels have been surveyed, specifically at unit testing and system testing. In section 4, the three testing type methodologies are presented, which are white-box testing, black-box testing and grey-box testing. Then, the challenges in automated mobile applications User Interface (UI) testing have been presented. Finally, the conclusion of this survey is presented.

2. Testing types

There are two types of testing. They are functional testing and non-functional testing. Functional testing is concerned with the application's functionalities. For mobile application testing, it is concerned with testing application's UI, service functionalities and Application Programming Interface (API) [7, 16]. UI testing is a type of functional testing. It executes events through application's UI for testing the application's behavior [16]. Most of the studies focus on UI testing as there are lots of libraries supporting UI mobile application testing, this could help researchers working at this point [17]. Service is a functionality that could work in the application's background. Service testing is concerned with appropriate management of its life cycle [18]. API is a set of software functionalities that could run by another application. API testing ensures that there is no error could happen as result of this integration [19].

Non-functional testing is concerned with testing the application's non-functional requirements. There are several testing types of non-functional testing for mobile applications as performance testing, stress testing, security testing, compatibility testing, usability testing and accessibility testing. Performance testing is concerned with checking the application's performance behavior when it is used by several users. It checks its response, time it could take for executing, the usage of mobile resources such as memory, network, and power [16, 18]. Stress testing is concerned with checking the application's performance behavior when it is used by a huge number of users or at high usage of mobile resources. It checks when the application could crash and its capability for restoration [18]. Security testing tests the application's capability of rejecting access of unauthorized users and protecting the user's data. It assures that the application is invulnerable to any attacks could happen. It also detects the weak points of the application that could result into security breach in order to be solved [18]. Compatibility testing finds out the mobile devices that could support the AUT. There are several configuration profiles at the market, so it is important to check the devices that would fail to run the AUT. Emulators have decreased the cost of compatibility testing [18]. Usability testing assures that the AUT is easy to be used, that the user will not face any difficulties while dealing with AUT. The AUT should respond quickly to user's actions. However, this could be hard when there are more than one activity running at the same time [18]. It checks that AUT's UI design is usable to prevent any confusions for users [16, 20]. Accessibility testing is a type of usability testing. It tests the application's behavior with disabled users. There are some features at mobile operating systems that could help, as text-to-speech in android [18].

3. Testing Levels

Testing levels are classified into unit testing, integration testing, system testing and acceptance testing [16, 21]. Unit testing is concerned with verifying the smallest component of AUT. It focuses on finding out any error at the application's code [16, 21]. Integration testing targets revealing the bugs that happen after combining different application's units [18]. It could be performed using different ways such as integrating independent components at the beginning as database then combining other system functionalities which is bottom-up integration. Top-down integration is the opposite of the bottom-up integration as the components are tested and integrated to the system dependent components. For many applications those two ways are mixed and applied together [18, 22]. System testing is concerned with checking that the application has completed all its requirements after integration [18]. Acceptance testing checks that the application has reached the user's trust by completing all his requirements. It also checks that it does not have any problems using real data by the user. This testing is performed by the testers and users [21, 22].

Most of the studies focus on unit testing and system testing. The following subsections review the approaches proposed for each testing level of them.

3.1 Unit testing

Unit testing is usually done by the application's developers. It is also called component testing. It could assure the application's behavior by testing the response to different types of events. Moreover, it could be used for checking that the life cycle events are properly managed [18], as at [23, 24] and the user's events as at [25].

3.1.1 Model-based / Model learning testing

Several studies have focused on checking that the management of life cycle events. At MobiGUITAR [23] some errors have been found as result of the inappropriate life cycle management of AUT's activities. It is a model-based / model learning approach that has enhanced its previous model that worked on desktop applications. It could not work with mobile applications as it has no state and did not consider security. The application's UI state has been modelled through a reverse engineering approach. Model-based testing approaches create test cases based on the extracted model of AUTs [26]. Model learning testing approaches learn the model of AUT while testing [27]. At [24] a model-based approach has been proposed for generating application's life cycle test cases. The problem of ensuring the quality of mobile application's data has been addressed through different states of application's activity life cycle. A model of application's activity life cycle has been constructed using source code, parsed, and analyzed. Then, a graph has been created for each activity. Moreover, automated test cases have been generated and executed using the constructed model. The status of each system resource has been checked that it is either acquired or released in the suitable life cycle method, to detect any failure could happen at system resources.

3.1.2 Record & replay testing

Record & replay testing is an approach for recording the interactions by testers with application's UI components to create a script. This script could be automatically replayed later for the testing process [28]. ACRT [25] is a record & replay testing approach. It is implemented for minimizing the testing effort. It captures the user's events and input. Then, a script is created by obtaining the coordinates of the event at the screen or the code of the pressed key. The script is replayed by Robotium [29] which is an open source automated testing framework. Assertions are used to find out if there are any errors at the application's UI components.

3.2 System Testing

System testing is concerned with finding any errors that could happen when executing the complete application. It has three steps, first the application runs on an emulator. Then, the environment changes gradually towards real device. Finally, it runs on a real device [18]. Many applications have applied system testing, as with search-based testing [8–10, 14, 30], model-based testing [11, 12], model learning testing [15] and random-based testing [13, 31–33].

3.2.1 Search-based testing

Search-based testing approaches apply metaheuristic approaches for generating optimized test cases by evaluating some solutions with a fitness function [34]. EvoDroid [10] has applied an approach for solving the defects resulting from using genetic algorithm to find the best individuals affecting the performance of the search. The application's behavior has been analyzed and modelled for detecting the source code dependencies. Then, search approach has been applied on the detected independent code parts to find the appropriate crossing over between genes and going in depth at the code to increase the coverage. AGRippin [8] is a search-based approach for handling the inefficiency of generation of test suites by other approaches. Crossover has been applied to generate test cases (chromosomes). They have been ensured that they could be executed by applying some heuristic criteria. A mutation approach has been applied and ranked test cases based on fitness. Genetic algorithm has been merged with hill climbing algorithm. Sapienz [9] applies a genetic algorithm with randomness for optimization of test sequences and increasing coverage. If the input is an Android Application Package (APK), then it is instrumented, to find out the coverage. Monkey++ [14] generates control flow graph for AUT and traverses through it by using Depth First Search (DFS) strategy. The generated test cases achieved more coverage results with less execution time than Monkey. ADAPTDROID [30] performs adaptation for test cases of AUTs with common methods. It extracts their semantics though their APKs. Then, it matches between the events of AUTs through the implemented evolutionary algorithm. Finally, it produces effective related test cases with more similarity in semantics between the AUTs compared to other approaches.

3.2.2 Model-based testing

Different model-based approaches for UI testing were proposed in literature, they have focused on maximizing coverage and generating relevant events. However, the exploration strategy could be a challenging issue. Stoat [12] is a model-based approach for UI testing. It has addressed several problems, as the exhaustive derivation of tests from the models to validate an application, the redundancy of randomly generated tests that results from the previous point, there are no models for some applications and the generation of UI events only. Dynamic analysis for exploration and static analysis have been combined for building a stochastic model. A guided search for finding the best models has been applied. Generation of system events by using intents is also supported. CrawlDroid [11] is model-based approach for the redundant exploration of an application that could get to the same state multiple of times. A feedback exploration approach has been applied and scores have been given to actions based on their ability to get to a new state in the application. That leads to an increasing in the coverage and discovering more failures.

3.2.3 Model learning testing

As the limitations of previous approaches were the susceptibility of errors from manual testing, requirement of source code of application and the generation of UI events only GATS [15] has been proposed. It learns the model of AUT using finite-state machine. The first state is the application's installation. An unexplored transition is selected, then the model is updated. The target is to find more bugs in the least time. It generates a report when a crash happens. It supports system and UI events.

3.2.4 Random-based testing

Random-based testing approaches depend on the randomness of selecting the events. Several studies have focused on generating relevant and minimizing redundancy compared to monkey. Monkey [31] is a testing tool at Android Software Development Kit (SDK). It generates pseudo-random events. Monkey testing was used in the beginning for testing Macintosh (Mac) programs in 1983 [35]. Dynodroid [32] generates relevant events for testing an application. Three approaches have been implemented for selecting an event which are frequency, biased random and uniform random. Frequency selects the event with the minimum number of selections. Uniform random selects a random event. Biased random keeps a history of events' number of selections. Dynodroid's performance is better than monkey. At [33] an approach has been presented for handling the excessive time and effort required for generation of test cases. It is based on statistical analysis from mining users' usage. First, the executed events have been recorded to get the usage logs. Then, a behavioral model from those logs has been generated. An event will be randomly selected based on probabilistic calculations. MonkeyImprover [13] enhances AUT through refactoring the GUI without affecting the functionalities of AUT. It extracts GUI elements of AUT that a user could interact with. Then, it generates a weight for each functionality based on its complexity. Then, GUI components would be resized based on their weight. As the size increases, the chance of interacting with monkey increases.

4. Testing Type Methodologies

There are three testing type methodologies. They are white-box testing, black-box testing and grey-box testing.

White-box testing is also called structure-based methodology. It considers the internal structure of an application, as the testing process is based on the presence of the application's source code [21]. There are several white-box testing approaches as AGRippin [8], EvoDroid [10], MonkeyImprover [13], Monkey++ [14], ACRT [25] and [24]. Their input is source code of android application.

Black-box testing is also called specification-based methodology, as it based on the application's requirements. It does not consider the presence of the source code of application. However, it considers interfaces [21]. There are several black-box testing approaches as at CrawlDroid [11], Stoat [12], GATS [15], MobiGUITAR [23], Monkey [31], Dynodroid [32], ADAPTDROID [30] and [33]. Their input is the application's APK.

Grey-box testing is a combination of both black-box testing and white-box testing. Testing is based on the application's structure and requirements [21]. At Sapienz [9], where the input is only application's APK, it is extracted to get the source code for instrumentation. However, the approach could also be performed if the source code is present.A summary for existing studies is presented at Table 1. They are sorted according to the year of publication ascendingly.

Table 1 - Summary of UI testing existing studies

Paper	Name	Year	Testing Type Methodology	Testing Level	Testing Approach	Testing environment	Advantages	Disadvantages
[31]	Monkey	1983	Black-box	System	Random-based	Mobile device and emulator	Fast execution time	Irrelevant events
[32]	Dynodroid	2013	Black-box	System	Random-based	Android 2.3.5 emulator	Relevant events	Slow execution time
[23]	MobiGUITAR	2014	Black-box	Unit	Model-based / Model learning	Not specified	Enhanced model of desktop applications	Simple exploration strategy
[10]	EvoDroid	2014	White-box	System	Search-based	Emulators in parallel	Fast execution time with increased coverage	Slow execution time with complex applications
[25]	ACRT	2014	White-box	Unit	Record & replay	Mobile device	Decreased time and effort	Not providing coverage
[8]	AGRippin	2015	White-box	System	Search-based	Android 2.3.3 emulator	Increased effectiveness than applying hill climbing solely	Same coverage at successive iterations
[9]	Sapienz	2016	Grey-box	System	Search-based	Android 4.4 emulator and mobile device	Optimization of test sequences	Tested applications may not cover all existing fields
[12]	Stoat	2017	Black-box	System	Model-based	Android 4.4.2 emulator	Relevant events	Incomplete exploration of UI
[33]	Random GUI Testing of Android Application Using Behavioral Model	2017	Black-box	System	Random-based	Android Nexus 5 emulator	Mining users' usage and increasing coverage	Complex events not handled
[11]	CrawlDroid	2018	Black-box	System	Model-based	Android 4.4 emulator and mobile device	Increasing coverage and discovering more failures	Tested applications may not cover all existing fields
[15]	GATS	2019	Black-box	System	Model learning	Android 5.1 emulator	Supporting system and UI events	Slow execution time
[24]	A Model for Generating Automated Lifecycle Tests	2020	White-box	Unit	Model-based	Android 11 mobile device	Detecting failures in system resources	Failed resources are released manually
[13]	MonkeyImprover	2020	White-box	System	Random-based	Not specified	Minimizing redundant events generated by Monkey	Needs more evaluation (one AUT only)
[14]	Monkey++	2021	White-box	System	Search-based	Not specified	Relevant events, increase coverage and fast execution time	Tested applications may not cover all existing fields
[30]	ADAPTDROID	2021	Black-box	System	Search-based	Android emulators	Semantic test cases	High cost of computation

5. Challenges in Automated Mobile Applications UI Testing

Several challenges in automated mobile applications testing approaches through UI testing have been surveyed.

- Selecting the AUTs covering all existing fields is a challenging issue. As the approach's performance could change between different applications [9, 11]. Sapienz [9] and CrawlDroid [11] could need to select more AUTs in different fields to ensure their effectiveness .
- The effectiveness UI testing at model-based approaches could be affected by applying simple exploration strategies such as using breadth-first search approach in MobiGUITAR [23] or the inability to extract the whole behaviors leading into incomplete exploration of UI of an application as in Stoat [12].
- The slow execution time at some approaches as GATS [15] and Dynodroid [32] compared to some existing tools as monkey. However, They have achieved higher performance results than monkey [31], as monkey generates irrelevant events to the application.
- Most of the studies as [33] focus on simple events such as to click, long-click, text and scroll. Since handling complex input events such as dragging is a challenging issue.
- Complex AUTs with many conditions could have lower performance than other AUTs when applying the same testing approach. By applying EvoDroid [10], complex applications got slower execution time.
- The need of providing information about testing metrics as code coverage when validating the performance of an approach to ensure that it has covered as much as possible of AUT's code. ACRT [25] does not provide information about code coverage. Maximizing the coverage is a challenging issue for any approach as for EvoDroid [10]. For search-based approaches as AGRippin [8], the ratio of crossover and mutation could lead into that some test suites could have the same coverage at successive iterations. This could lead to decreasing the coverage.
- Detecting errors in system resources when using an application could be challenging issue, as there are multiple resources that could be assessed. This approach [24] has focused on 3 resources only which are drive, camera and location.
- Extraction of GUI elements of AUT that a user could interact with could be a challenging issue. As there are different formats of implementation for event handlers either by using particular annotations or by declaring their names in layout files or by using specific functions as in MonkeyImprover [13]. Moreover, common callback functions names could lead to inability of detection of control and its related callback function as in Monkey++ [14].

6. Conclusion

Every day, the importance of smartphones increases. Testing mobile applications has become essential to assure their quality. In this paper, a survey about automated UI testing for mobile applications has been presented. Mobile testing types, testing type methodologies and testing levels have been discussed. Existing studies have focused on the several issues such as discovering more failures in AUTs, generating relevant events with minimized redundancy, the non-generation of system events when testing AUT and optimization of test suites. Different existing UI mobile application testing studies from 2013 to 2021 have been presented. Those studies have been classified according to their year, testing type methodology, testing level, testing approach, testing environment, their advantages, and disadvantages. Testing approaches that have been presented in these studies are random-based testing, model-based testing, model learning testing, search-based testing, and record & replay testing. Most of the existing mobile testing studies are about UI testing, which is a type of functional testing, due to the presence of libraries supporting it. Moreover, the challenges that could happen at automated mobile applications UI testing have been discussed. As future direction, researchers can start by tackling those challenges.

References

1. Numan N., Abuelenin S., Rashad M.: Prediction of Lung Cancer Using Artificial Neural Network Int. J. Intell. Comput. Inf. Sci., 16, pp. 1–19 (2016)
2. Aouf M.: Application of K-MTh Algorithm for Accurate Lunge Cancer Detection Int. J. Intell. Comput. Inf. Sci., 19, pp. 1–20 (2019)
3. Ukaoha K., Ademiluyi O., Ndunagu J., Daodu S., Osang F.: Adaptive Neuro Fuzzy Inference System for Diagnosing Coronavirus Disease 2019 (COVID-19) Int. J. Intell. Comput. Inf. Sci., 20, pp. 1–31 (2020)
4. Nasr M., Farrag M., Nasr M.: E-Payment Systems Risks, Opportunities, and Challenges for Improved Results in E-Business Int. J. Intell. Comput. Inf. Sci., 20, pp. 1–20 (2020)
5. Amasha M.: Using Actionscript 3.00 To Develop an Android Application for Mathematics Course Int. J. Intell. Comput. Inf. Sci., 16, pp. 67–79 (2016)
6. Kaur A.: Review of Mobile Applications Testing with Automated Techniques Int. J. Adv. Res. Comput. Commun. Eng., 4, pp. 503–507 (2015)
7. Gao J., Bai X., Tsai W.-T., Uehara T.: Mobile Application Testing: A Tutorial Computer (Long. Beach. Calif)., 47, pp. 46–55 (2014)
8. Amalfitano D., Amatucci N., Fasolino A.R., Tramontana P.: AGRippin: A novel search based testing technique for android applications 3rd Int. Work. Softw. Dev. Lifecycle Mobile, DeMobile 2015 - Proc., pp. 5–12 (2015)
9. Mao K., Harman M., Jia Y.: Sapienz: multi-objective automated testing for Android applications Proc. 25th Int. Symp. Softw. Test. Anal. - ISSTA 2016, pp. 94–105 (2016)
10. Mahmood R., Mirzaei N., Malek S.: Evodroid: Segmented evolutionary testing of android apps Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 599–609 (2014)

11. Cao Y., Wu G., Chen W., Wei J.: Crawldroid: Effective model-based gui testing of android apps Proceedings of the Tenth Asia-Pacific Symposium on Internetware. pp. 1–6 (2018)
12. Su T., Meng G., Chen Y., Wu K., Yang W., Yao Y., Pu G., Liu Y., Su Z.: Guided, Stochastic Model-Based GUI Testing of Android Apps Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. pp. 245–256. Association for Computing Machinery, New York, NY, USA (2017)
13. Paydar S.: Automated GUI Layout Refactoring to Improve Monkey Testing of Android Applications Proc. RTEST 2020 - 3rd CSI/CPSSI Int. Symp. Real-Time Embed. Syst. Technol., (2020)
14. Doyle J., Saber T., Arcaini P., Ventresque A.: Improving mobile user interface testing with model driven monkey search Proc. - 2021 IEEE 14th Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2021, pp. 138–145 (2021)
15. Chen T., Song T., He S., Liang A.: A GUI-based automated test system for android applications Adv. Intell. Syst. Comput., 760, pp. 517–524 (2019)
16. Mostefaoui G.K., Tariq F.: Mobile apps engineering: design, development, security, and testing, CRC Press, (2018)
17. Tramontana P., Amalfitano D., Amatucci N., Fasolino A.R.: Automated functional testing of mobile applications: a systematic mapping study Softw. Qual. J., (2018)
18. Amalfitano D., Fasolino A.R., Tramontana P., Robbins B.: Testing android mobile applications: Challenges, strategies, and approaches Advances in Computers. vol. 89. pp. 1–52. Elsevier (2013)
19. Jorgensen A., Whittaker J.A.: An api testing method Proceedings of the International Conference on Software Testing Analysis \& Review (STAREAST 2000) (2000)
20. Bruegge B., Dutoit A.H.: Object-Oriented Software Engineering. Using UML, Patterns, and Java, (2009)
21. Homès B.: Fundamentals of Software Testing, (2013)
22. Sommerville I.: Software Engineering, Pearson Education, (2006)
23. Amalfitano D., Fasolino A.R., Tramontana P., Ta B.D., Memon A.M.: MobiGUITAR: Automated Model-Based Testing of Mobile Apps IEEE Softw., 32, pp. 53–59 (2014)
24. Motan M., Zein S.: Android App Testing: A Model for Generating Automated Lifecycle Tests 4th Int. Symp. Multidiscip. Stud. Innov. Technol. ISMSIT 2020 - Proc., (2020)
25. Liu C.H., Lu C.Y., Cheng S.J., Chang K.Y., Hsiao Y.C., Chu W.M.: Capture-replay testing for android applications Proc. - 2014 Int. Symp. Comput. Consum. Control. IS3C 2014, pp. 1129–1132 (2014)
26. Dias Neto A.C., Subramanyan R., Vieira M., Travassos G.H.: A survey on model-based testing approaches: A systematic review Proc. - 1st ACM Int. Work. Empir. Assess. Softw. Eng. Lang. Technol. WEASELTech 2007, Held with 22nd IEEE/ACM Int. Conf. Autom. Softw. Eng., ASE 2007, pp. 31–36 (2007)
27. Choi W., Necula G., Sen K.: Guided gui testing of android apps with minimal restart and approximate learning Acm Sigplan Not., 48, pp. 623–640 (2013)
28. Börjesson E., Feldt R.: Automated system testing using visual GUI testing tools: A comparative study in industry Proc. - IEEE 5th Int. Conf. Softw. Testing, Verif. Validation, ICST 2012, pp. 350–359 (2012)
29. GitHub - RobotiumTech/robotium: Android UI Testing, https://github.com/robotiumtech/robotium
30. Mariani L., Pezze M., Terragni V., Zuddas D.: An Evolutionary Approach to Adapt Tests across Mobile Apps Proc. - 2021 IEEE/ACM Int. Conf. Autom. Softw. Test, AST 2021, pp. 70–79

(2021)
31. UI Application Exerciser Monkey - Android Developers, https://developer.android.com/studio/test/monkey
32. Machiry A., Tahiliani R., Naik M.: Dynodroid : An Input Generation System for Android Apps Proc. 9th Jt. Meet. Found. Softw. Eng. ACM,pp. 224–234., (2013)
33. Muangsiri W., Takada S.: Random GUI Testing of Android Application Using Behavioral Model Int. J. Softw. Eng. Knowl. Eng., 27, pp. 266–271 (2017)
34. Afzal W., Torkar R., Feldt R.: A systematic review of search-based testing for non-functional system properties Inf. Softw. Technol., 51, pp. 957–976 (2009)
35. What is Monkey Testing_ Features, Types With Examples - Testbytes, https://www.testbytes.net/blog/monkey-testing/