Comparative Analysis of Fully Automated Testing Techniques for Android Applications

Ndukwe-Oke Eke
Department of Computer Science
Nile University of Nigeria
Abuja, Nigeria
ndukwe.eke@nileuniversity.edu.ng

Ibrahim Anka Salihu
Department of Software Engineering
Nile University of Nigeria
Abuja, Nigeria
ibrahim.salihu@nileuniversity.edu.ng

Asmau Usman
Department of Computer Science
Abdu Gusau Polytechnic
Tatata Mafara, Nigeria
asmee08@gmail.com

2023 2nd International Conference on Multidisciplinary Engineering and Applied Science (ICMEAS) | 979-8-3503-5883-4/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICMEAS58693.2023.10429901

***Abstract*—The software testing community has continued to research and develop new ways of testing Android mobile applications to ensure an application function as expected and serves its purpose. Given that every testing technique/tool has its strengths and weaknesses. This study aims to present a comparative analysis of fully automated techniques which automatically generate and execute test cases concurrently during runtime. We selected 10 fully automated techniques published from 2013 to 2023 techniques to identify the similarities and differences that exist among them. We clearly define the comparative criteria (such as the exploration method, systematic method, termination criterion, extraction criterion, and scheduling method) used for comparing the selected techniques. The analysis shows that most fully automated techniques adopt an active learning exploration approach for exploring the application under test (AUT). We also observed that only the techniques utilizing the active learning approach are capable of modelling and abstracting the graphic user interface (GUI) of the AUT, others randomly select events to be fired into the AUT.**

Keywords— Android Application, Mobile Application Software Testing, Testing Technique

I. INTRODUCTION

Computer technology for the past few years has gone from the desktop to the laptop and now to the handheld mobile device. Life seems to be far easier for the users of these devices as they have literarily built their world around it. Consequently, they have continued to patronize mobile applications to meet their day-to-day business and social needs. This transformation has made the job of software developers and testers more difficult to ensure these devices function as expected [1].

In the year 2021, about 1.54 billion smartphones were sold to users worldwide and over 371 million smartphones were purchased in the fourth quarter of 2021 [2], [3]. The high demand for smartphones and tablets led to a corresponding growth in the creation of mobile applications for these devices to address the computational requirements of their users [4], [5]. A recent study shows that over 230 billion mobile applications (Mobile Apps) were downloaded worldwide in 2021 alone [6]. The rate at which these mobile applications are being published into the marketplace poses a great challenge for software engineering researchers (particularly software testers) in determining the quality of these apps (that is, their functionality, behaviour, and performance under certain conditions) [5], [7]. Some of the challenges that hinder or limit the abilities of the software testers to effectively verify the quality of application software are the fragmentation of device models, variety of operating system (OS) platforms, fast release cycles, a huge number of mobile network operator, time and language barriers (such as time zone variation and targeted users), data security breach, etc. [8]–[10].

Mobile applications are "software programs designed to operate on mobile devices like smartphones, tablet PCs, and other mobile gadgets" [11], these applications can execute network-based functions through a cellular or satellite data connection [1], [11]. Due to the competitive nature of the mobile industry, mobile application developers now pay close attention to the quality of the applications they develop, responding to any form of negative reviews, and constantly updating their apps in other to rectify any bug-related issues reported by the users of these applications. Therefore, testing of these applications has become necessary and considered the best mechanism to ensure the quality of software programs in functionality, behaviours, performance, and features [12].

According to Morgado et al. [13], testing is essential in improving the quality of a product, hence, a crucial part of the mobile development process. The goal of testing software applications is to discover any form of defect that may affect the general functionality of the system. Testing can be manual – requiring some level of human intervention to generate test cases and interpret test results or automatically generate and execute test cases.

II. BACKGROUND

Testing of mobile applications to assess or verify their quality is manually or automatically implemented. Generally, test cases are important testing inputs that are generated from some kind of software artefact used as reference inputs for generating test cases such as the "program source code, software specification or design models" [14] as well as the information that is dynamically obtained from the program execution. Researchers have explored and documented various kinds of testing techniques in the cause of identifying and implementing a test strategy that is more efficient and cost-effective. Some of these testing techniques include "Script-based, Capture/Replay, Random walk, Model-based, Search-based, Symbolic/Concolic Execution, and Combinatorial-based" [15], [16].

979-8-3503-5883-4/23/$31.00 ©2023 IEEE

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.

1) Script-based

The script-based technique requires the software tester to manually write the test scripts that exploit the features of the application under test (AUT). A recent study conducted by Pan et al., proposed METER (Mobile Test Repair), a script-based technique and tool for repairing graphical user interfaces "test scripts for mobile apps that leverage computer vision techniques" [17]. Imparato suggested a fused approach involving GUI Ripping methodology with input perturbation testing, a method that systematically investigates the functionality of Android apps, constructing a model of the explored graphical user interface (GUI), and subsequently employing it to generate modified test inputs [18]. The application of this technique is implemented in GUI Ripper. Jiang et al., introduce a script-based method integrated into the MobileText tool, employing a "sensitive-event-based" approach to streamline case design, thereby improving both its efficiency and reusability [19].

2) Capture/Replay

The Capture/replay technique records entries during a manual test and creates automated test scripts that can be reused. This technique is considered the early transition stage from manual to automated testing, it captures a series of user actions within an application and transforms them into automated test scripts for subsequent replay. Capture/Replay can otherwise be referred to as Capture and Playback or Record and Replay.

3) Random Testing

In the random testing technique, the AUT is tested by randomly generating independent inputs and test cases. This method of testing like the one proposed by [20] is mostly implemented in Monkey Tool, although they are considered very scalable and easy to use, its effectiveness can be questionable [21] because Monkey Tool can be modified to allow certain adjustments to different classes of events i.e., a tester can decide to send "click events on the GUI more frequently than the gesture ones" [22].

4) Model-based

In the model-based testing approach, test cases originate from a model outlining the functional aspects of the tested application. All model-based techniques are automated; they include any approach that automatically generates and executes test cases without any human intervention.

5) Systematic/Active learning-based

Model learning addresses the shortcomings of model-based testing by learning a model of the app during testing, it directs the generation of user input sequences according to the learned model [23].

6) Search-based

The search-based testing approach utilizes a meta-heuristic method which is a higher-level procedure designed for finding, generating, or selecting a heuristic. The meta-heuristic search aims to optimize the search technique (like Genetic Algorithm) to fully or partially automate the generation of test data [24], [25].

7) Symbolic/Concolic Execution

In the symbolic testing technique, symbolic values represent test input values instead of using concrete (actual) data, and program variables are expressed symbolically. Consequently, output values are generated as a result of the input symbolic values. In contrast, concolic execution automates test input generation by using both concrete (actual data) and symbolic values for inputs, executing the program in both concrete and symbolic modes [26],[27].

8) Combinatorial-based

Combinatorial testing tests the application under test with all the required parameter value combinations; this type of testing technique can detect failure triggered as a result of the interactions among parameters in SUT [28], [29].

III. METHODOLOGY

We start by conducting a mini survey to identify the existing Android mobile app testing tools/techniques published in reputable platforms such as ACM, Web of Science, Google Scholar, and EBSCOhost. We categorize the identified techniques into two groups mainly, *partial* and *fully* automated based on their degree of automation (i.e., test execution and test case generation). Since some technique automates only the test execution step while requiring some level of human intervention at the test case generation stage. We classify this set as partial or semi-automated, example of this type of technique include the *script-based* and *capture/replay* techniques.

Our study is based on fully automated techniques which automatically generate and execute test cases concurrently during runtime. We selected 10 fully automated techniques published from 2013 to 2023 and then explored the characteristics and features that exist among them.

A. Selected Techniques/Tools

Techniques within this category can generate and execute test cases automatically at runtime. The subsections below summarize the 10 fully automated testing techniques/tools identified in this study as follows:

1) Dynodroid

Dynodroid is used for the automated testing of Android applications [30], it is specifically designed for exploring and testing Android applications by generating and executing a diverse set of inputs including various events like taps, gestures, and system calls.

2) iMPAcT

iMPAcT [31] uses a random technique that randomly fires an event to a running application and dynamically analyses the AUT. It recognizes the app's distinct UI patterns and verifies their correct implementation. The testing process concludes upon the execution of a predefined number of events.

3) BBoxTester

BBoxTester utilizes a random approach that relies solely on the Monkey Tool for generating and executing test cases [32].

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.

4) MobiGUITAR

MobiGUITAR is an automated testing tool for Android applications that operates on a GUI-driven approach, relying "on the observation, extraction, and abstraction of the runtime state of GUI widgets" [33]. It dynamically explores the GUI of the AUT creating its state-machine model, and then uses this model to obtain the sequence of events for execution.

5) A3E

A3E employs two systematic testing methods, Targeted Exploration and Depth-First Exploration [34]. The Targeted exploration technique focuses on the specific areas or functionalities of an application under test. While the Depth-First exploration technique emphasizes testing a single feature or functionality of the AUT before moving on to others.

6) DROIDRACER

DROIDRACER [35] is a tool for testing the performance and reliability of Android applications. It's designed to automate the testing of Android apps under various conditions and scenarios to identify issues related to performance, security, and stability. Key features and capabilities of DroidRacer include automated testing, stress testing, race condition detection, randomized testing, coverage analysis, and crash reporting.

7) Swifthand

Swifthand is an active learning technique that utilizes "execution traces generated during the testing phase to learn an approximate model of the GUI" [23], and subsequently, uses it to select user inputs that navigate the application to states that have not been explored previously.

8) ORBIT

ORBIT [36] is a search-base technique that automatically obtains the GUI model of a mobile app by statically analyzing the application's source code allowing the extraction of a collection of user actions associated with each widget in the GUI. Subsequently, these events are systematically executed on the running app using the depth-first search strategy.

9) DroidCrawler

DroidCrawler [37] automatically explores the AUT utilizing the depth-first search approach, extracting the GUI Tree model from the graphical user interface. The exploration concludes once there is no other new interface to explore.

10) AppDoctor

AppDoctor [38] performs different exploration methods such as interactive, scripted, random, and systematic on the AUT. Interactive exploration shows the available list of actions to the tester, who then selects which action to perform. The scripted method allows the tester to write scripts to select actions to be executed by the AppDoctor. The random method allows AppDoctor to randomly select an action to perform, and lastly, the systematic method systematically explores the GUI for bugs using different search heuristics (such as breadth-first or depth-first search). The exploration terminates after a predefined number of events (for random testing) or after the entire interface has been explored (for systematic testing).

B. Comparative criteria

Having observed that all the fully automated testing techniques share similar iterative behaviour (such as the exploration, learning, extraction, scheduling, and termination method used) when exploring the application under test whereby a sequence of events is scheduled and executed in a given iteration, then, stops once a predefined termination condition is reached. We further explore these iterative behaviours in the following subsections and represent them in Table I.

1) Exploration method: this defines the approach used by the fully automated technique while exploring the AUT and for generating test cases. The potential values within the exploration method include – *random* and *active learning*. The random method does not rely on the AUT model, while the active-learning exploits the model of the AUT at runtime.

2) Systematic method: this criterion is valid if the exploration method relies on the model of the AUT. It is made up of two sub-parameters namely – The inferred model and the Abstraction method. The former states the type of GUI model inferred by the testing technique. The five main inferred models identified in the literature are the GUITree, DATG, SATG, and EDLTS. The latter describes the method employed to create an abstract representation of a GUI model, they include the Active Name (AN), a GUI state that belongs to the same "Activity class", the "Single Attribute Value" (SAV) refers to a scenario in which a GUI state within the model is linked to GUIs that share an identical set of widgets and possess the same value for a particular attribute while the "Multiple Attribute Values" (MAV) refers to a situation where a GUI state is associated with all the GUIs that have the same set of widgets and exhibit identical values for a subset of the widgets' attributes. SAV and MAV can be configured to allow the software tester the flexibility to select the specific type of widget and attribute to consider while describing the GUI.

3) Termination criterion: this defines the approach used to terminate or end an automated testing process. The common termination criterion values that can be assumed by any online testing technique include a Predefined Number of Events (PNE) that stops the iteration loop after executing a predefined number of events, a Predefined Length (PL) that stops the loop when the sequence of events generated reached a predefined length, a Predefined Time of Execution (PTE) stops the loop once a predefined time of execution is reached. However, in a situation where a random exploration method is used, the Sat value terminates the testing process when N (a given random technique) reaches the same testing adequacy i.e., the same code coverage. On the other hand, the MC value terminates the process when an expected coverage has been reached.

4) Extraction criterion: this defines the events to be extracted and sent to the AUT by the technique. We identified two values for this purpose: the Predefined Events (PE) and the Relevant Events (RE) values, the former considers a subset of the predefined type of event that could be

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.

potentially managed by any Android application, while the latter ensures that the technique extracts only relevant events that can be handle by the app in its present state.

5) Scheduling method: this explains or describes the method used to select the sequence events to be fired next during the test process. The two possible sets of values as identified in the literature are graph-based and random-based values, the former is common with fully automated techniques. They include L*, Targeted, Depth First, Breadth First, and L* - minimal restart. The latter involves a random selection of events such as Priority, Frequency, Adaptive, Biased, and Uniform.

TABLE I. ANALYSIS OF THE TESTING TOOL

Criteria		*Testing Tools*									
			[30]	[31]	[32]	[33]	[34]	[35]	[23]	[36]	[37]	[38]
		Classification	Dynodroid	iMPAcT	BBoxTester	MobiGUITAR	A3E	DROIDRACER	SwiftHand	ORBIT	DroidCrawler	AppDoctor
Exploration Method		Random	✓	✓	✓	✓			✓			✓
		Active Learning				✓	✓	✓	✓	✓	✓	✓
Systematic Method	Inferred Model	GUITree				✓		✓			✓	✓
		DATG					✓					
		SATG					✓					
		EDLTS							✓			
		FSM								✓		
	Abstraction Method	AN				✓	✓					
		SAV				✓						
		MAV				✓		✓	✓	✓	✓	✓
Termination Criterion		PNE	✓	✓	✓	✓						✓
		PL						✓				
		PTE							✓			
		Sat				✓						
		MC				✓	✓		✓	✓	✓	✓
Extraction Criterion		PE			✓	✓					✓	
		RE	✓	✓		✓	✓	✓	✓	✓		✓
Scheduling Method	Graph-based	Breadth-first				✓						✓
		Depth-first				✓	✓	✓		✓	✓	✓
		Targeted					✓					
		L*							✓			
		L* Minimal							✓			
	Random-based	Uniform	✓			✓			✓			✓
		Frequency	✓	✓								
		Biased	✓									
		Priority			✓							

IV. DISCUSSION

Among the identified tools presented in this study, [30-32] employ only a random exploration approach while exploring the application under test. Techniques [34 - 37], use an active learning exploration approach, while [23], [33], and [38] adopt both random and active learning approaches for exploring the AUT. We also observed that only the techniques utilizing the active learning approach are capable of modelling and abstracting the graphic user interface (GUI) of the AUT, others randomly select events to be fired into the AUT. Among these techniques utilizing the systematic method, [33], [35], [37], and [38] inferred a GUITree; [34] adopt both the Dynamic Activity Transition Graph (DATG) and Static Activity Transition Graph (SATG) to model the behaviour of the AUT; [23] and [36] use the Extended Deterministic Labeled Transition Systems (EDLTS) and Finite State Machine (FMS) respectively.

For the abstraction method used, [30], [23], and [36 - 38] all utilized the multiple attribute values (MAV); [34] uses the single attribute value (SAV), only technique [33] adopts the activity name (AN), (SAV), (MAV) abstraction values.

Test execution must terminate at some point during testing, techniques that terminate after a pre-defined number of events (PNE) is fired into the AUT include [30 - 33], and [38]. *DROIDRACER* [35] terminates once the sequence of events generated reaches a predefined length (PL), and *SwiftHand* [23] stops executing after a predefined time of execution (PTE). In the event where a random exploration is used, the Saturation value (Sat) terminates the testing process once a given technique (*n*) reaches the same code coverage.

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.

If a technique adopts the systematic exploration approach, the multiple coverage (MC) value terminates the process after an expected code coverage is reached.

As for the extraction method used, only the techniques [32], [33], and [37] adopted the predefined events (PE) for events to be extracted and sent to the AUT, others adopted relevant event values. At the event scheduling stage, the techniques either explored a graph-based or random-based method for scheduling events. We observed that all the active learning techniques adopt a graph-based scheduling method, while their random-based counterpart used either the uniform, frequent, biased or priority approach in scheduling events fire on the AUT.

V. CONCLUSION

Various kinds of testing techniques have been proposed and incorporated into testing tools that implement them. App developers/testers now have the opportunity to decide which technique (i.e., tool) to adopt while testing their app. In this study, we have identified these fully automated testing techniques and further explored the similarities and differences that exist among this set of techniques to provide the software testing community and researchers a quick guide to help them make informed decisions as to which testing technique best suits their testing needs. Our future work will include comparing these fully automated testing techniques to ascertain their strength and weaknesses.

REFERENCES

[1] G. J. Myers, T. Badgett, C. Sandler, G. J. Myers, T. Badgett, and C. Sandler, *Art of Software Testing*, 3rd edition. 2012.

[2] Statista, “Smartphone market share by vendor 2009-2023.,” 2023, [Online]. Available: accessed: August 12, 2023. [Online]. Available: https://www.statista.com/statistics/271496/global-market-share-held-by-smartphone-vendors-since-4th-quarter-2009/

[3] Counterpoint Research, “Global Smartphone Market Share: By Quarter,” 2023.

[4] J. Gao, X. Bai, W. T. Tsai, and T. Uehara, “Mobile application testing: A tutorial,” *Computer (Long. Beach. Calif).*, vol. 47, no. 2, pp. 46–55, 2014, doi: 10.1109/MC.2013.445.

[5] I. A. Salihu, R. Ibrahim, B. S. Ahmed, K. Z. Zamli, and A. Usman, “AMOGA: A Static-Dynamic Model Generation Strategy for Mobile Apps Testing,” *IEEE Access*, vol. 7, pp. 17158–17173, 2019, doi: 10.1109/ACCESS.2019.2895504.

[6] Statista, “Annual number of global mobile app downloads 2016 - 2021," URL: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/ [accessed 2023-09-05] [WebCite Cache ID 6ynHSFx8g] XSL•FO RenderX https://mhealth.jmir.org/2023.

[7] I. A. Salihu, R. Ibrahim and A. Usman, "A Static-dynamic Approach for UI Model Generation for Mobile Applications," 2018 7th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO), Noida, India, 2018, pp. 96-100, doi: 10.1109/ICRITO.2018.8748410.

[8] X. Ma, N. Wang, P. Xie, J. Zhou, X. Zhang, and C. Fang, “An Automated Testing Platform for Mobile Applications,” *Proc. - 2016 IEEE Int. Conf. Softw. Qual. Reliab. Secur. QRS-C 2016*, pp. 159–162, 2016, doi: 10.1109/QRS-C.2016.25.

[9] A. Usman, N. Ibrahim, and I. A. Salihu, “TEGDroid: Test Case Generation Approach for Apps considering Context and GUI events,” *International Journal on Advanced Science, Engineering and Information Technology, vol. 10, no. 1, p. 16, Feb. 2020, doi:10.18517/ijaseit.10.1.10194.*

[10] Saucelabs.com, “Mobile App Testing : main challenges, different approaches, one solution,” 2017.

[11] N. O. Eke and I. A. Salihu, “Design and Implementation of a Mobile Library Management System for Improving Service Delivery,” *Path Sci.*, vol. 7, no. 4, p. 3001, 2021, doi: 10.22178/pos.69-7.

[12] C. Tao and J. Gao, “Building a Model-Based GUI Test Automation System for Mobile Applications,” *Int. J. Softw. Eng. Knowl. Eng.*, vol. 26, no. 9–10, pp. 1605–1615, 2016, doi: 10.1142/S0218194016710042.

[13] I. C. Morgado, A. C. R. Paiva, and J. P. Faria, “Automated pattern-based testing of mobile applications,” *Proc. - 2014 9th Int. Conf. Qual. Inf. Commun. Technol. QUATIC 2014*, pp. 294–299, 2014, doi: 10.1109/QUATIC.2014.47.

[14] A. Usman, N. Ibrahim, and I. A. Salihu, “Comparative Study of Mobile Applications Testing Techniques for Context Events,” *Advanced Science Letters*, vol. 24, no. 10, pp. 7305–7310, 2018, doi: 10.1166/asl.2018.12933.

[15] I. A. Salihu, R. Ibrahim, and A. Mustapha, “A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing,” *Journal of Telecommunication, Electronic and Computer Engineering (JTEC),* vol. 9, no. 3, pp. 45–49, 2017.

[16] S. Anand *et al.*, “An orchestrated survey of methodologies for automated software test case generation generation,” *J. Syst. Softw.*, vol. 86, pp. 1978–2001, 2013.

[17] M. Pan, T. Xu, Y. Pei, Z. Li, T. Zhang, and X. Li, “GUI-Guided Test Script Repair for Mobile Apps,” *IEEE Trans. Softw. Eng.*, vol. 48, no. 3, pp. 910–929, 2022, doi: 10.1109/TSE.2020.3007664.

[18] G. Imparato, “A Combined Technique of GUI Ripping and Input Perturbation Testing for Android Apps,” *Software Engineering (ICSE), 2015 IEEE/ACM 37th International Conference*, vol. 2, pp. 760–762, 2015, doi: 10.1109/ICSE.2015.241.

[19] B. Jiang, X. Long, and X. Gao, “MobileTest: A tool supporting automatic black box test for software on smart mobile devices BT - 29th International Conference on Software Engineering, ICSE 07 - 2nd International Workshop on Automation of Software Test, AST’07, May 20, 2007 - May 26, 20 IEEE Computer Society Technical Council on Softwar.

[20] C. Hu, “Automating GUI Testing for Android Applications,” in *Proceedings of the 6th International Workshop on Automation of Software Test, AST’11, ACM, New York, NY, USA*, 2011, pp. 77–83. doi: 10.1145/1982595.1982612.

[21] P. Patel, G. Srinivasan, S. Rahaman, and I. Neamtiu, “On the effectiveness of random testing for Android: Or how I learned to stop worrying and love the monkey,” *Proc. - Int. Conf. Softw. Eng.*, pp. 34–37, 2018, doi: 10.1145/3194733.3194742.

[22] D. Amalfitano, N. Amatucci, A. M. Memon, P. Tramontana, and A. R. Fasolino, “A general framework for comparing automatic testing techniques of Android mobile apps,” *J. Syst. Softw.*, vol. 125, pp. 322–343, 2017, doi: 10.1016/j.jss.2016.12.017.

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.

[23] W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," *ACM SIGPLAN Not.*, vol. 48, no. 10, pp. 623–639, 2013, doi: 10.1145/2544173.2509552.

[24] P. Mcminn, R. Court, and P. Street, "Search-based Software Test Data Generation : A Survey," pp. 1–58, 2004.

[25] P. McMinn, "Search-Based Software Testing: Past, Present and Future," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, Berlin, Germany, 2011, pp. 153-163, doi: 10.1109/ICSTW.2011.100.

[26] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. J. Softw. Tools Technol. Transf.*, vol. 11, no. 4, pp. 339–353, 2009, doi: 10.1007/s10009-009-0118-1.

[27] K. Sen, "Concolic testing," *ASE'07 - 2007 ACM/IEEE Int. Conf. Autom. Softw. Eng.*, pp. 571–572, 2007, doi: 10.1145/1321631.1321746.

[28] T. Shiba, T. Tsuchiya and T. Kikuno, "Using artificial life techniques to generate test cases for combinatorial testing," *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.*, Hong Kong, China, 2004, pp. 72-77 vol.1, doi: 10.1109/CMPSAC.2004.1342808.

[29] C. Nie and H. Leung, "A Survey of Combinatorial Testing," vol. 43, no. 2, pp. 1–29, 2011, doi: 10.1145/1883612.1883618.

[30] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *2013 9th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE 2013 - Proceedings*, 2013, pp. 224–234. doi: 10.1145/2491411.2491450.

[31] I. C. Morgado and A. C. R. Paiva, "The iMPAcT tool: Testing UI patterns on mobile applications," *Proc. - 2015 30th IEEE/ACM Int. Conf. Autom. Softw. Eng. ASE 2015*, pp. 876–881, 2016, doi: 10.1109/ASE.2015.96.

[32] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of Android apps," *Proc. - 10th Int. Conf. Availability, Reliab. Secur. ARES 2015*, pp. 501–510, 2015, doi: 10.1109/ARES.2015.70.

[33] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, 2014, doi: 10.1109/MS.2014.55.

[34] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," *Proc. Conf. Object-Oriented Program. Syst. Lang. Appl. OOPSLA*, pp. 641–660, 2013, doi: 10.1145/2509136.2509549.

[35] P. Maiya, A. Kanade, and R. Majumdar, "Race detection for Android applications," *ACM SIGPLAN Not.*, vol. 49, no. 6, pp. 316–325, 2014, doi: 10.1145/2666356.2594311.

[36] W. Yang, M. R. Prasad, and T. Xie, "A Grey-box Approach for Automated GUI-Model Generation of Mobile Applications,". *In: Fundamental Approaches to Software Engineering.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 250-265. ISBN: 978-3-642-37057-1. DOI: 10.1007/978-8-642-37057-1_19.

[37] P. Wang, B. Liang, W. You, J. Li, and W. Shi, "Automatic android GUI traversal with high coverage," *Proc. - 2014 4th Int. Conf. Commun. Syst. Netw. Technol. CSNT 2014*, pp. 1161–1166, 2014, doi: 10.1109/CSNT.2014.236.

[38] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with appDoctor," *Proc. 9th Eur. Conf. Comput. Syst. EuroSys 2014*, 2014, doi: 10.1145/2592798.2592813.

Authorized licensed use limited to: Wikipedia. Downloaded on January 22,2025 at 10:15:25 UTC from IEEE Xplore. Restrictions apply.