Multi-device, Robust, and Integrated Android GUI Testing: A Conceptual Framework

Multi-device, robust, and integrated Android GUI testing: A conceptual framework

Riccardo Coppola[1 [0000−0003−4601−7425]], Luca Ardito[1 [0000−0002−0501−7886]], and Marco Torchiano[1 [0000−0001−5328−368X]]

Department of Control and Computer Engineering, Polytechnic University of Turin
Corso Castelfidardo, 34/d, 10138 Torino, Italy name.surname@polito.it

Abstract. Android GUI (Graphical User Interface) testing is often overlooked by developers, even if it holds the potential to guarantee sufficient quality for the apps. It is typically regarded as a burdensome activity. High maintenance costs, fragmentation, fragility, and flakiness of the test artifacts are the main hurdles for wider adoption in practice. This article identifies the main modules that could enable efficient and robust mobile testing in continuous development environments. On top of them, we sketch the infrastructure of a conceptual framework for the generation, execution, and maintenance of mobile test suites. We also present a call to action for software testers, developers, and researchers towards the framework realization in practice.

Keywords: GUI testing · Mobile Testing · Android development · Testing Framework

1 Introduction

According to recent analyses, Android has achieved the highest overall market share among all - not just mobile - operating systems [1]. Nowadays, Android apps have reached a very high complexity in terms of both graphical appearance and provided features. The above characteristics, and the fact that most interactions with Android apps take place through their GUI (Graphical User Interface) widgets, justify the need for thorough employment of GUI testing techniques.

Many tools for automating Android GUI testing are available in the market [15]. However, research has shown that manual testing through the GUI is still preferred to test automation because many challenges about the latter are still lingering [18]. In recent years, academic literature and tools from the industry have tackled many mobile-specific GUI testing challenges. However, to the best of our knowledge, no comprehensive approach or framework is available, which considers all the principal issues of the three phases of the test cases lifecycle: generation, execution, and maintenance.

Here we put forward a comprehensive conceptual framework for mobile GUI testing: to do so, we summarize the most critical challenges of the practice, survey the existing techniques and tools tackling them, and propose an agenda for

[1] https://gs.statcounter.com/os-market-sharemonthly-202111-202303

practitioners aiming at incorporating mobile GUI testing in continuous development pipelines.

2 Mobile GUI testing: state of the art and practice

Both industry and academia have proposed many different tools and techniques to perform GUI testing on mobile applications and identified several challenges for the practice. We adopt two orthogonal classification schemes to categorize mobile GUI testing techniques. The process of GUI testing revolves around the identification of elements on the GUI. The properties and means to identify the elements are called *locators*. Test cases also require *oracles*, i.e., properties that have to be verified with assertions to verify that the test executed correctly. GUI testing techniques can be classified according to how the elements on the GUI are identified (i.e., based on the type of locator that is used) [1]:

- *Coordinate-based* GUI testing techniques identify widgets through exact on-screen candidates. Due to major volatility of such properties, these techniques have been largely abandoned in practice.
- *Layout-based* GUI testing techniques identify widgets through properties that are declared in the Android layout descriptor of the current screen (e.g., ids, text content, content description, widget type).
- *Visual*, or *Image recognition-based* GUI testing is based on computer vision techniques, utilizing screen captures as locators. State-of-the-art image-based approaches leverage techniques ranging from pixel-per-pixel comparison to more elaborate matching algorithms (e.g., SIFT, SURF or Akaze feature vectors [3]).

GUI testing techniques can also be classified according to how the test sequences are generated. Adopting a taxonomy proposed by Linares-Vazquez et al. [19]:

- *Automation APIs* or *Scripted* testing tools allow manual writing of test scripts, using platform-specific scripting languages; test scripts can then be executed using dedicated or universal test runners.
- *Capture & Replay* (C&R) testing tools automatically generate test scripts from sequences of user interactions with the AUT (Application Under Test), thus 'capturing' (recording) real usage scenarios [10].
- *Automated Test Input Generation* techniques automate the definition of the test sequences; the generation can be based on heuristics (e.g., a random selection of locators and verification of the occurrence of exceptions or bugs) or on the coverage of a GUI model, which can be itself automatically generated [5].

Despite the availability of tools for Android GUI testing, research has highlighted that the practice is often conducted manually by developers [7]. The reasons behind the limited adoption of automated tools are manifold and specific to the mobile domain:

Fragmentation: One of the main issues for Android GUI testing is the intrinsic *Fragmentation* of the domain: developers must verify and validate the compatibility of the AUT with multiple target configurations where it may be deployed [16]. Such configurations include screen sizes, screen densities and aspect ratios (*hardware fragmentation*) or different versions of the operating system where the AUT has to be installed (*software fragmentation*). The fragmentation issue particularly plagues the Android domain because of the multitude of devices running the operating system and the coexistence of many releases of the o.s. (Operating System) that receive parallel support. Therefore, the tester must run identical test sequences on multiple devices, either real or emulated. This repetition represents an obstacle to the adoption of continuous integration/development practices.

Fragility and Flakiness: Test scripts (either manually written or automatically generated) for mobile applications need critical maintenance to cope with even small changes in the application GUI during its life cycle. This issue is typically referred to as *Fragility* and is considered among the main hindrances to a wide adoption of GUI testing tools [9]. Fragility can be caused by changes in the GUI layout properties that invalidate layout-based locators and oracles or by aesthetic changes of the widgets that invalidate visual locators and oracles. The Fragility in GUI tests requires the tester to analyze the outcome of each test execution carefully. This action is needed because functionally valid test sequences may lead to false positives due to the inability to find the locators, and proper refactoring is therefore needed for fragile locators. Test cases are hampered also by a high level of *flakiness*, i.e., they may have a non-deterministic outcome over repeated runs. Flaky executions can be related to unpredictable execution times, network availability, concurrency in the test devices, and interaction with the execution environment [12].

Limited generalizability: The high complexity of the GUI and the many different ways of composing the individual screens (with orchestrations of *Activities*, *Fragments* and other components) pose relevant generalizability issues to GUI testing techniques. It is not trivial to identify universal models able to represent the GUI states of any application at the desired granularity. Several efforts have been provided for mutation testing [11]. As well, to the best of our knowledge, no coverage models have been defined yet in the academic literature [6]. Therefore, evaluating the quality of generated test sequences or comparing multiple testing tools' results over multiple AUTs is still a complex task.

Hybrid application testing: Finally, even though *Native* mobile applications still represent the vast majority in the Android marketplace, many different frameworks are available to construct *hybrid* or *progressive* web applications that are rendered on a mobile browser to guarantee a similar – if not equivalent – user experience to that of apps specifically designed for the mobile system. Such frameworks typically define components using properties that differ from those specified in the layout files of native apps, therefore making scripted *layout-based* testing tools harder to generalize to all Android apps.

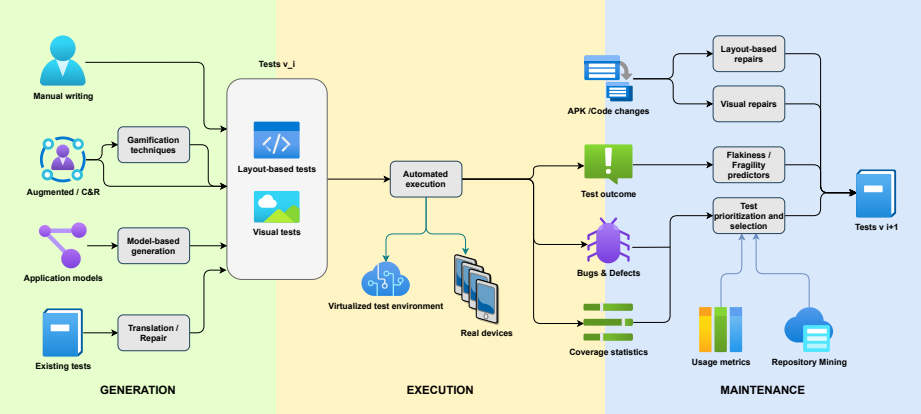

Fig. 1. Modules of the GEM framework: high-level view

3 Conceptualization of the GEM framework

The proposed framework, summarized in Figure 1, is structured according to the three main life cycle phases: Generation, Execution, and Maintenance (GEM).

3.1 Test generation and translation

The test generation module of the framework provides new test sequences to add to the test suite for the AUT. The tools should expose multiple and coexisting ways of defining test cases to generate test sequences sharing a common syntax. As the first basic way of developing test cases, the tools shall allow traditional *manual writing* of test sequences in the specific syntax adopted by the tool.

The framework should also offer ways of directly capture the interaction of testers with the GUI of the AUT through the implementation of *Capture & Replay* test sequence generation mechanisms. The literature about C&R testing has recently witnessed some evolution for the concept, in the form of *augmented* layers providing live information and suggestions to the tester to generate input sequences and assertions. C&R test case generation can also be enriched by incorporating concepts that can increase testers' user experience and engagement during the test capture sessions. Recent Software Engineering literature has explored the benefits offered by the implementation of gamification mechanics in software testing. Gamification consists in applying concepts that are proper of game design to other practices, e.g. scoring mechanisms and leaderboards, or graphical live feedback [14]. Gamification has been applied to the phase of test case generation in Capture & Replay tools, where the testers can be incentivized in performing more thorough exploration by the usage of scoring schemes, leaderboards, progression indicators and live feedback [13].

Test generation should be guided by knowledge of the structure of the GUI and the nature of the AUT. Traditional model-based approaches, which mainly

apply coverage-based heuristics on automatically built graph-based models, can be paired with AI-based test generation techniques. The latter apply learning mechanisms to automatically adapt general test sequences written in natural language to the specific AUT after classifying it and its activities (i.e., individual screens shown to the user). AI-based test case generation can significantly lower the effort in writing test case sequences and allow stakeholders without software development skills, to easily write test cases without having access to the application code or knowing the internal GUI structure of the AUT [4].

In our conceptualization, testing tools should allow the coexistence of variants of the same test sequences, based on *layout-based* and *visual* locators and oracles. The benefits of having both variants of test cases are manifold. Firstly, it enhances the expressive power of the test suite since the type of bugs found by the two approaches are different: layout-based oracles can only verify the GUI structure and properties of the AUT, while the application of image-recognition based assertions can only spot defects in the visual rendering of the widgets. Similarly, the two types of locators and oracles are fragile to a complementary type of changes in the GUI of the AUT: pictorial changes invalidate visual test cases, whilst changes in the layout properties invalidate the Layout-based test cases. In our conceptualized framework, we envision applying a translation-based mechanism, allowing the automated creation (or regeneration) of layout-based locators and oracles from visual ones and vice versa. Such an approach has already been proven effective for both the web and mobile domains [17] [8] in enhancing the test suites' effectiveness and mitigating graphic fragilities.

In the generation phase of the framework, we identify a primary research gap in the generation of visual test cases and the application of gamified mechanics. Both methodologies have been explored for general-purpose or web-based tools, but not specifically for the Android domain.

3.2 Test execution

Once the test cases are generated, the Execution phase must take place for each test sequence. The most important issue to tackle in this phase is the software and hardware fragmentation of the Android OS and devices. Therefore, it is fundamental to pair real devices with virtualized environments emulating most of the available hardware, software and graphical configurations the AUT must provide compatibility.

Several infrastructures have been conceptualized to provide virtual testbeds to deploy mobile applications. Infrastructure-as-a-Service solutions have been provided to perform layout-based GUI testing or performance testing [2]. Existing commercial tools (e.g., test.io) allow crowd-sourced execution of test cases on real hardware devices; however, they mainly allow pure manual and exploratory test sequences. In general, the current state of the art and practice lacks ways to automatically and extensively execute image-recognition based test suites on multiple configurations. At the end of the test execution, we envision the generation of complete test reporting, including the test cases outcome, found defects,

statistics and analyses of the execution issues that can lead to refinement and enhancements of the test suites.

We envision that each testing environment should track the quantity of fragile or *flaky* test cases. A possible solution to identify flaky test cases is to execute each test case multiple times in order to flag the test cases as passing (all executions are passing), flaky (some executions are passing) or failing (all executions fail). Failing tests can furtherly be divided into true positives (i.e., test cases failing due to real defects in the AUT) and fragile tests (i.e., test cases failing due to unrelated changes in the AUT). Sets of change-based metrics have been defined in the empirical software engineering literature to measure the number of fragile tests and their impact on the maintenance effort for the test suite [9]. The objective of fragility tracking is to mark the test cases as fragile if they require too much intervention during the evolution of the AUT. Other modules of the framework can use this flagging activity to aid test prioritization.

Regarding the *Execution* phase of our framework, we identify a primary research gap about the coverage measurement when mobile test cases are executed since no coverage model for mobile applications has been widely accepted by research in the field. Albeit several precise coverage models exist (e.g., multi-device coverage proposed by Vilkomir et al. to measure the reduction of the fragmentation issue [22]), a universally generalizable coverage model is still missing.

3.3 Test maintenance and repair

The literature highlights that test scripts maintained manually (either layout-based or visual) can be tedious and costly when they become obsolete. Since mobile applications typically have a quick evolution, the cost of test case maintenance can be required frequently and become unsustainable for developers.

In our framework, we envision a module in charge of automatically adapting the test cases to the changes in the APK, in the GUI pictorial appearance and the application code. Since two equivalent sets of locators and oracles are maintained for the test suite, both layout properties and visual locators have to be updated automatically. Several approaches have been proposed for the automated repair of test cases. Some tools are based on event-sequence models describing the behaviour of the application and abstracting the changes made to the GUI between different releases of the same application. These tools, however, are mainly aimed at preserving the connection between different screens of the AUT traversed during the execution of test sequences and still require the manual intervention of the testers for the preparation of the original models that guide the testing process. Alternative model-less approaches rely on the definition of similarity indexes for the widgets to be interacted during test cases to identify locators that should be treated as the same one even in the presence of changes in their properties or visual appearance [23].

Even incorporating a sophisticated mechanism for test case repair, some test sequences may still need manual maintenance during the evolution of the application. The execution of test cases on multiple devices and configurations, especially when the GUI has to be rendered and verified through computer

vision algorithms, can also become unsustainable if the test suite grows significantly. Therefore, a module for the maintenance of test cases should include mechanisms to prioritize and select them to reduce the execution and maintenance time for the subset of test sequences to execute in continuous integration and development settings. Test prioritization should be guided from metrics resulting from the test case execution module, e.g., generated bugs and coverage reports. At the same time, the test prioritization and selection module should incorporate diverse information gathered from repository mining. Usage metrics gathered through mobile APIs on pilot users can help identify the activities and user interaction sequences on which test cases should focus. Suppose the AUT is available as an open-source repository (e.g., on GitHub). In that case, mechanisms can be developed to mine and interpret the issues left by contributors to the project to identify the most critical sections of the code. As well, if the AUT is released on a marketplace, techniques of marketplace analysis can be deployed to mine the user reviews and identify typical usage patterns leading to crashes.

We identify important research gaps regarding the Maintenance phase of our framework. To the best of our knowledge, no automated mobile test suite repair tools have been validated with real-world test suites. Similarly, no prioritization model has been specifically described for mobile GUI test cases used for regression testing, and existing ones are mostly applied to non-functional properties (e.g., for security testing [21]).

4 Discussion

The proposed framework constitutes a vision for the modules that may enable a mobile testing pipeline incorporating multiple means to generate test cases, execute tests on both real and virtual devices, and ease test suite maintenance along the evolution of a mobile app. In our vision, a platform implementing all the modules described in the framework would guarantee:

- Seamless Continuous Integration execution with tracking of test metrics on multiple device and screen configurations. This could enable writing the tests once and run them on all devices (solving the *hardware fragmentation* issue);
- Incremented robustness of tests to changes in structural layout and visual appearance, reduced maintenance effort, and reduced time to find bugs when tests are used with regression purposes. This would allow writing the test once and run it for the whole mobile app lifecycle (solving the *test fragility* issue).

Currently, no testing tool, either available on the market or described in the literature, implements all the features described in our framework. The generation, execution and maintenance pipeline could be partly obtained by combining multiple available instruments. Without claims of exhaustiveness, Table 1 reports, for each module of the framework, existing examples from academia and industry. As reported in the table, there is a high availability of tools for manual test case generation and acquisition of test sequences through the Capture &

Replay technique. Several of these tools come embedded in the Android development environment (namely, UIAutomator, Espresso and the related Recorder). Many tools, mostly academic, have been developed for model-based generation of mobile test sequences. Several commercial platforms for crowdsourced testing on multiple devices are also available. Definition and execution of visual test cases can still be performed by utilizing multi-domain visual testing tools (e.g., Sikuli or EyeAutomate) that can be applied to emulated devices on a desktop environment. Conversely, most of the modules in the frameworks related to the maintenance of test cases are not fully implemented by available tools. Some of them are still in early-stage academic investigation (e.g., fragility prediction mechanisms and translation-based tools). However, academic literature in the field proposed heuristics and metrics that can be adapted as add-ons to existing open-source testing frameworks. Finally, research on gamified mechanisms is still in an early stage, and mainly tied to the practice of software engineering in general, with very few verticalizations on the practice of desktop and web application testing.

Table 1. Available tools implementing modules of the framework

Module	Tool	Origin	Open-Source	URL / Notes
◆ Manual test case writing	UIAutomator	Industry	Yes	https://developer.android.com/training/testing/ui-automator
	Espresso	Industry	Yes	https://developer.android.com/training/testing/espresso
	Appium	Industry	Yes	https://appium.io
	Ranorex	Industry	No	https://lp.ranorex.com
	Calabash	Industry	Yes	https://github.com/calabash/calabash-android
● Gamified test case generation	-	-	-	Described in academic research for web application testing
◆ Model-based generation	MobiGUITAR	Academia	Yes	https://github.com/AndrewZcc/mobiGUITAR/actions
	Tricentis Tosca	Industry	No	https://www.tricentis.com/products/automate-continuous-testing-tosca/
	Quantum	Industry	No	https://www.perfecto.io/integrations/quantum
	STOAT	Academia	Yes	https://github.com/tingsu/Stoat
	Droidbot	Academia	Yes	https://github.com/honeynet/droidbot
■ Translation-based tools	TOGGLE	Academia	No	[8]
◆ Capture & Replay	TestProject	Industry	No	https://testproject.io/mobile-test-recorder/
	Repeato	Industry	No	https://www.repeato.app
	RERAN	Academia	Yes	https://www.androidreran.com
	Espresso Test Recorder	Industry	Yes	https://developer.android.com/studio/test/espresso-test-recorder
	Barista	Academia	Yes	https://github.com/AdevintaSpain/Barista
◆ Automated Execution Environments	Test.io	Industry	No	http://test.io
	CrowdSprint	Industry	No	http://crowdsprint.com
	Firebase TestLab	Industry	No	https://firebase.google.com/products/test-lab
◆ Visual Test Case Execution	Sikuli	Academia	Yes	http://sikulix.com
	EyeAutomate	Industry	No	http://eyeautomate.com
■ Test Repair Mechanisms	GUIDER	Academia	No	[23]
■ Fragility Prediction	Coppola et al.	Academia	No	[9]
■ Test Prioritization and Selection	Michaels et al.	Academia	No	[20]

5 Call to action

Based on the proposed framework and the relative mapping to existing and missing tools, we can issue a call to action for three distinct stakeholders:

- *Software Testers* can leverage the framework to identify the tools needed to generate test cases efficiently and adopt instruments for each of the categories for which tools are already available (◆);
- *Tool developers* can leverage the framework to assess the existing tools against the most crucial needs of the practice of GUI testing in industrial settings. Practitioners can find opportunities for new implementations in the categories of tools that have been – to the best of our knowledge – explored exclusively by academic research, e.g. gamified testing tools, test repair and fragility prediction mechanisms (■);
- *Researchers* can leverage the framework to guide future research efforts, discriminating between categories of tools that are widely implemented by the industry and others that have not yet been explored for the mobile testing domain. Systematic research efforts can be conducted in Software Engineering literature to categorize and classify all available tools for mobile testing and provide a comprehensive state of the art mapping according to the framework's modules. Empirical research and industrial case studies are needed to assess the benefits produced by each module of the framework in reducing fragility, fragmentation, and maintenance effort required by test suites (●).

6 Conclusions

In this paper, we envisioned future trends in the automated GUI testing landscape and provided action points for different stakeholders in the field. Our framework can serve as an instrument for researchers and developers of testing tools to assess which among envisioned modules are available in the literature and the market and which need further research and development.

A tentative evaluation of the proposed framework would involve assessing its potential to address the challenges and goals it sets out to achieve. In particular, the framework aims to tackle issues related to hardware fragmentation, test fragility, and test maintenance costs. It is important to note that a full evaluation would require empirical research and industrial case studies, to fully validate the benefits produced by each module of the framework in addressing the challenges it aims to tackle.

It is still worth stressing that some of the modules in the framework, such as fragility prediction mechanisms and translation-based tools, are still in early-stage academic investigation or have not been fully implemented by available tools. The success of the proposed framework in reducing test maintenance costs would rely on the development and integration of these modules and their effectiveness in practical scenarios.

References

1. Alégroth, E., Gao, Z., Oliveira, R., Memon, A.: Conceptualization and evaluation of component-based testing unified with visual gui testing: an empirical study.

In: 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST). pp. 1–10. IEEE (2015)
2. Ali, A., Maghawry, H.A., Badr, N.: Automated parallel GUI testing as a service for mobile applications. Journal of Software: Evolution and Process **30**(10), e1963 (2018)
3. Ardito, L., Bottino, A., Coppola, R., Lamberti, F., Manigrasso, F., Morra, L., Torchiano, M.: Feature matching-based approaches to improve the robustness of Android visual gui testing. ACM Transactions on Software Engineering and Methodology (TOSEM) **31**(2), 1–32 (2021)
4. Ardito, L., Coppola, R., Leonardi, S., Morisio, M., Buy, U.: Automated test selection for Android apps based on apk and activity classification. IEEE Access **8**, 187648–187670 (2020)
5. Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for Android: Are we there yet?(e). In: 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 429–440. IEEE (2015)
6. Coppola, R., Alégroth, E.: A taxonomy of metrics for gui-based testing research: A systematic literature review. Information and Software Technology p. 107062 (2022)
7. Coppola, R., Ardito, L., Morisio, M., Torchiano, M.: Mobile testing: new challenges and perceived difficulties from developers of the Italian industry. IT Professional **22**(5), 32–39 (2020)
8. Coppola, R., Ardito, L., Torchiano, M., Alégroth, E.: Translation from layout-based to visual Android test scripts: An empirical evaluation. Journal of Systems and Software **171**, 110845 (2021)
9. Coppola, R., Morisio, M., Torchiano, M.: Mobile GUI testing fragility: A study on open-source Android applications. IEEE Transactions on Reliability **68**(1), 67–90 (2018)
10. Di Martino, S., Fasolino, A.R., Starace, L.L.L., Tramontana, P.: Comparing the effectiveness of capture and replay against automatic input generation for Android graphical user interface testing. Software Testing, Verification and Reliability **31**(3), e1754 (2021)
11. Escobar-Velásquez, C., Linares-Vásquez, M., Bavota, G., Tufano, M., Moran, K., Di Penta, M., Vendome, C., Bernal-Cárdenas, C., Poshyvanyk, D.: Enabling mutant generation for open-and closed-source Android apps. IEEE Transactions on Software Engineering **48**(1), 186–208 (2020)
12. Fazzini, M., Gorla, A., Orso, A.: A framework for automated test mocking of mobile apps. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 1204–1208. IEEE, Washington, DC, USA (2020)
13. Fulcini, T., Ardito, L.: Gamified exploratory gui testing of web applications: a preliminary evaluation. In: 2022 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 215–222. IEEE (2022)
14. Fulcini, T., Coppola, R., Ardito, L., Torchiano, M.: A review on tools, mechanics, benefits, and challenges of gamified software testing. ACM Computing Surveys (2023)
15. Kong, P., Li, L., Gao, J., Liu, K., Bissyandé, T.F., Klein, J.: Automated testing of Android apps: A systematic literature review. IEEE Transactions on Reliability **68**(1), 45–66 (2018)
16. Lanui, A., Chiew, T.K.: A cloud-based solution for testing applications' compatibility and portability on fragmented Android platform. In: 2019 26th Asia-Pacific Software Engineering Conference (APSEC). pp. 158–164. IEEE (2019)

17. Leotta, M., Stocco, A., Ricca, F., Tonella, P.: Pesto: Automated migration of DOM-based web tests towards the visual approach. Software Testing, Verification And Reliability **28**(4), e1665 (2018)
18. Linares-Vásquez, M., Bernal-Cárdenas, C., Moran, K., Poshyvanyk, D.: How do developers test Android applications? In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 613–622. IEEE, Washington, DC, USA (2017)
19. Linares-Vásquez, M., Moran, K., Poshyvanyk, D.: Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In: 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 399–410. IEEE, Washington, DC, USA (2017)
20. Michaels, R., Khan, M.K., Bryce, R.: Test suite prioritization with element and event sequences for Android applications. In: 2021 IEEE 11th Annual Computing and Communication Workshop and Conference (CCWC). pp. 1326–1332. IEEE, Washington, DC, USA (2021)
21. Sadeghi, A., Esfahani, N., Malek, S.: Mining mobile app markets for prioritization of security assessment effort. In: Proceedings of the 2nd ACM SIGSOFT International Workshop on App Market Analytics. pp. 1–7. Association for Computing Machinery, New York, NY, USA (2017)
22. Vilkomir, S.: Multi-device coverage testing of mobile applications. Software quality journal **26**(2), 197–215 (2018)
23. Xu, T., Pan, M., Pei, Y., Li, G., Zeng, X., Zhang, T., Deng, Y., Li, X.: Guider: Gui structure and vision co-guided test script repair for Android apps. In: Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 191–203. Association for Computing Machinery, New York, NY, USA (2021)