SILESIAN UNIVERSITY OF TECHNOLOGY PUBLISHING HOUSE

SCIENTIFIC PAPERS OF SILESIAN UNIVERSITY OF TECHNOLOGY 2020
ORGANIZATION AND MANAGEMENT SERIES NO. 144

RESEARCH ON RELIABILITY OF MOBILE APPLICATIONS IN A DISTRIBUTED ENVIRONMENT

Mirosław ŁAWRYNOWICZ[1], Jakub LEGUT[2], Ireneusz J. JÓŹWIAK[3*]

[1] Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wroclaw; miroslaw.lawrynowicz@pwr.edu.pl, ORCID: 0000-0003-0317-9811
[2] Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wroclaw; jakub-legut@wp.pl
[3] Wroclaw University of Science and Technology, Faculty of Computer Science and Management, Wroclaw; ireneusz.jozwiak@pwr.edu.pl, ORCID: 0000-0002-2160-7077
* Correspondence author

Purpose: The purpose of the paper is to present the research on reliability of mobile applications in distributed systems.
Design/methodology/approach: Ensuring the reliable operation of the software created for Android OS is related to ensuring compatibility with different versions of the systems. The use of various security policies, formal and structural requirements in various Android releases results in a high failure rate of dedicated programs.
Findings: There is, therefore, a need to implement new testing methods.
Originality/value: The article proposes a method of software reliability analysis dedicated to distributed systems. The developed solution is based on concurrency using the actor model. As part of the research, the effectiveness of the developed solutions was evaluated.

Key words: mobile application, Android, actor model, gray box testing.

Category of the paper: Research paper, technical paper.

1. Introduction

Popularization of mobile technologies, understood as universal access to wireless Internet via portable devices (smartphones, tablets etc.), and simultaneous development of mobile operating systems (Android, iOS, Sailfish), determine the growth of developers' interest in creating dedicated applications. The popularity of Android system and the possibility of its implementation on various hardware configurations results in its strong fragmentation[1]

[1] The popularity distribution of different versions of the operating system.

(Wei et al., 2018). The result of uneven fragmentation is the need to adapt the developed applications to operate in systems that use various functionalities and define new restrictions (e.g. limiting the number of tasks running in the background within one program (Guide, 2019)), which leads to a high failure rate of dedicated applications (Han et al., 2012). Software failures and low code quality are among the main reasons for their arbitrary removal from the Google Play store (Wang et al., 2018). The basic techniques of testing the reliability of applications can be divided into the following three groups: black box (lack of knowledge about the internal structure of the application), white box (full knowledge of the source code) and gray box (partial knowledge of the source code) (Tretmans, 1999). The literature presents different approaches to black box testing. In the work by C. Hu the results of reliability tests obtained using the dedicated *Monkey* tool were presented (Hu et al., 2011). Randomly generated test events (hereinafter referred to as scenarios) are entered into the application via a graphical user interface, emulated in a virtual environment, and the program's responses are aggregated in the form of reports. Criteria for reliability evaluation included the application not reacting to triggered events, the inability to display activity and the detection of errors related to data types (e.g. projection).

An approach to reliability analysis, formulated as an optimization problem, is discussed in (Amalfitano et al., 2015). The effectiveness of the source code coverage test (Yang et al., 2009) was adopted as a criterion, and a solution to the problem using the genetic algorithm was developed. It was proposed to use the robotic arm to simulate the use of the application by the user (Mao et al., 2017). The system is divided into two separate parts: test data generator and test implementer. The key aspect of testing is the analysis and interpretation of the application's output image to compare with the desired output.

The testing of mobile software using the white box method comes down to the structural analysis of the code. Full knowledge of the code allows to carry out tests of the correctness of functionality, e.g. unit tests, mutations and dynamic tests. A technique for conducting security tests of mobile applications in the cloud was also developed (Mahmood et al., 2012). The authors use reverse engineering tools to obtain full source code and assess it in the form of a unit test set.

Gray box reliability analysis combines testing of functionality and correctness of the code. The authors of the publication (Arifiani et al., 2016) have developed a test method based on the result of the Ant Colony Optimization heuristic. Generating possible scenarios is done randomly based on the UML diagram. In (Yang et al., 2013) function calls are presented in the form of a deterministic finite automaton. Two construction algorithms have been developed to find unreliable sequences in the machine.

The subject of the work is to propose a distributed environment structure for testing the reliability of software created for Android (version 4.4 with the code name "KitKat" to version 8.1 named "Oreo"). An application was developed, in which the proposed assumptions were implemented and the evaluation was carried out. The tests were performed on the basis of the

gray box methodology, having information on the full list of available functions and their sequence of calls in the tested programs (the sequences of calls within each function are not taken into account). The main goal of the research is to analyze errors related to multithreading: errors in the implementation and design. The focus was also on the analysis of technical parameters of the tested applications, i.e. on the analysis of the demand for hardware and software resources, that may contribute to the occurrence of unhandled exceptions or the lack of response from the program.

The second chapter presents the research problem and discusses the cases of errors that are to be identified. The third chapter describes the structure of the developed environment and the method of testing implementation. The results of the research and conclusions are discussed in the fourth and fifth chapter respectively.

2. Problem formulation

Applications dedicated to the Android system are built on activities[2] associated with views[3] or components that represent the interface fragment. Switching between activities is carried out using selected elements of graphical interface (buttons, sliders etc.) or using predefined gestures. Fail-safe transition between successive application screens (activities) or functions within the same activity is considered the most important measurable feature determining the reliable functioning of the graphic interface (Takala et al., 2011). The subject of this work is to extend such approach to include research on application parameters (number of running services in the background, integrity of downloaded data) and devices (memory requirements, processor load).

The first methodology of reliability tests includes checking the integrity of data entered into the SQL database and stored in the device's memory in a situation, when the process of downloading the data from the network is carried out by many services within the same application. Such a case is closely related to downloading data (e.g. from the cloud), which is necessary for the correct operation of the program. This may be the case (Figure 1), in which the data collected when switching from Activity 1 to Activity 2 has not yet been fully downloaded, and when returning from Activity 2 to Activity 1, another portion of data starts to download. In this case, the deposition of services in the background (multiple switching of activity) may lead to them being turned off by the system, which results in the lack of integrity of the desired data. The problem becomes real, if the correct operation of Activity 2 depends on the data downloaded in Activity 1. Limiting the services running in the background is a special feature of Android in the "Oreo" version (Amalfitiano et al., 2015). It is implemented

[2] The current functionality of the application (Content displayed on the screen).
[3] Graphical user interface designed for the selected functionality or presenting multimedia content.

automatically and is not preceded by an informational message. The amount of downloaded data is taken as the criterion for this methodology. The application is considered to meet the requirements if the amount of data downloaded k is equal to nk, where n is the number of activity switches.

Another aspect of the work is the analysis of data access reliability, which is modified and saved from the level of many threads, when the download service is running simultaneously (Figure 2). If the selected dataset is modified from the A-thread level, and in the B-thread the program attempts to operate on this set, then the *ConcurrentModificationException* error may occur (Concurrent, 2019).

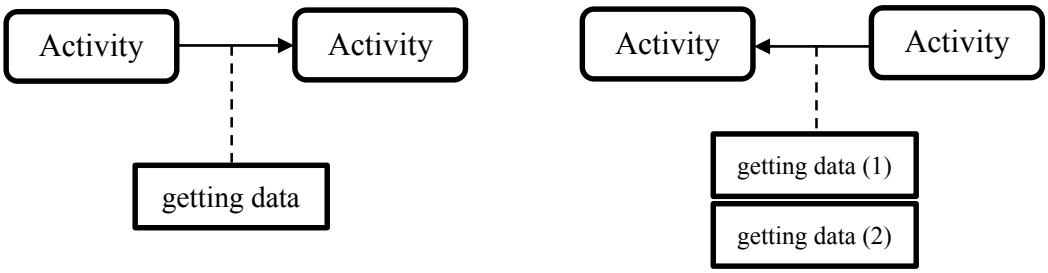


Figure 1. Switching activity scenarios. Source: own work.

This is because an attempt may be made to iterate after the size of the data set changed during the iteration. Otherwise, an attempt to refer to a collection from thread A when it was removed by a function in thread B leads to a *NullPointerException* error (Null Pointer, 2019).

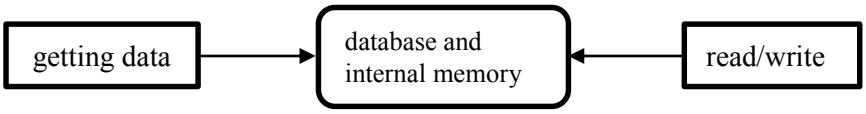


Figure 2. Performing read/write operations while getting data from the cloud. Source: own work.

In addition, communication between threads can lead to an *ExecutionException* error (Execution, 2019) if thread A (in this case the main thread) tries to get the result of thread B that threw the exception. This applies to cases when topping up the browsed multimedia content (e.g. photos, video) is interrupted due to lack of Internet access or lack of internal memory of the device. The reliability test consists in counting the occurrence of errors: *ConcurrentModificationException, NullPointerException* and *ExecutionException*, and aggregation the sequence of actions (application states and called functions) that lead to them. By the occurrence of an undefined error, we mean the inability to change the state (regardless of the code signal/instruction) and, consequently, the lack of reaction from the program. The subject of the analysis is additionally subjected to the following device parameters: RAM memory requirements and processor load.

3. Developed solution

The reliability of the application was tested using a proprietary environment using the Actor model supplied with Akka[4] libraries. Programming with the use of actors, i.e. independent synchronized concurrent threads that can communicate with each other (Agha et al., 1997), is one of the ways to implement applications based on concurrency of processes and operating on large data sets. Software written for the needs of the work was done in the SCALA[5] language, since it provides mechanisms for scalability of calculations and effective management of the available memory (Haller, 2012). An additional criterion for choosing the implementation technology is the flexibility and efficiency of error handling. In the Android environment, the occurrence of an error is equivalent to displaying an information message and restarting the application. The deployment of tests in many virtualized environments (containers) allows for simulation of many scenarios, in parallel with simultaneous data exchange between containers, while error reporting comes down to designation of one actor, referred to as an aggregator, which acts as an independent process. The aspect of correct error aggregation is related to the implementation of statistics regarding code locations (running specific functions, in which errors occurred), memory requirements and identification of exceptions.

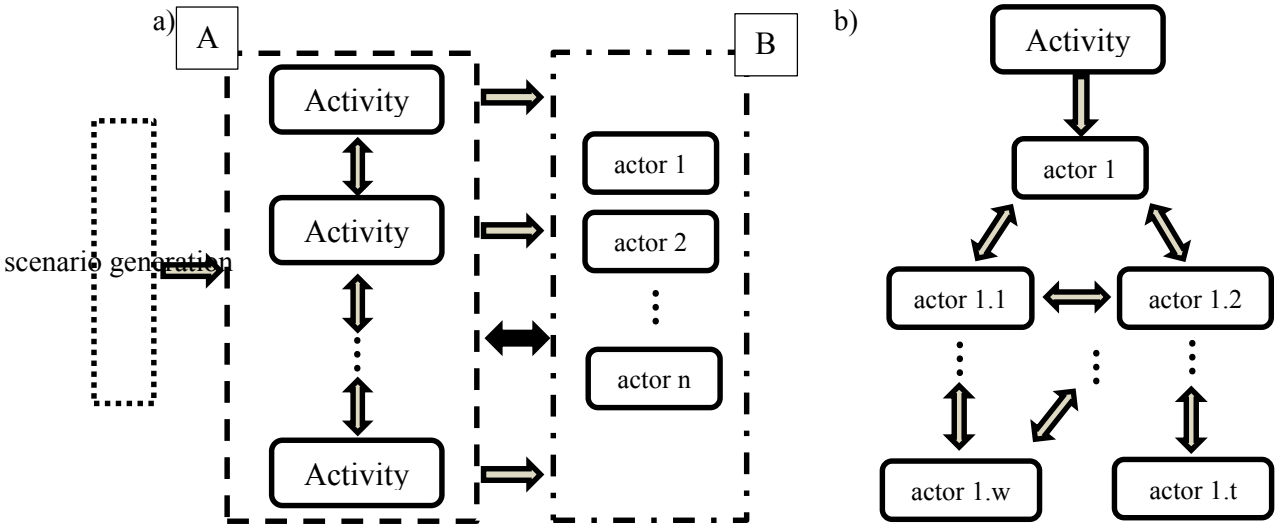


Figure 3. The structure of the testing environment, Source: own work.

The reliability test starts with the generation of *m* set of scenarios: $\mathbf{S} = \{S_i\}$, i = 1, 2, ..., *m* (Figure 3a), which may occur during the use of the program. The S_i element is a sequence of steps implemented through the interface. The **S** set is generated using the software proposed by Y. Yang (Yang et al., 2017) and, additionally, it has been extended with the original case

[4] A set of libraries providing solutions in the field of parallel and concurrent calculations directed at large data sets (Big Data), https://akka.io/.

[5] Programming language compatible with the Java virtual machine, characteristic element of which is combining features of function and object languages.

collection. The other components of the system are two applications: the subject of study "A" and the diagnostic tool "B". The "A" application has a set of *n* activities, each of which represents the functionality of the "A" application. Each activity is represented by a deterministic finite state machine, in which states define the functions associated with this activity. The "B" application represents the dependencies between function calls in the "A" code in the form of a set of actors. The set of actors in "B" represents the mapping of the automaton from the activity in "A". The consequence of this is the use of a hierarchical scheme of actors (Figure 3b). Although the functions are called in the code sequentially, asynchronous communication between the actors was allowed to exchange information about the generated exceptions. Three separate groups of actors were introduced: the superior actor, assigned to the activity, the listener (subordinate) actors and the ROOT actor (managing all actors). The number of generated actors in "B" is equal to the number of all tested functions, add the number of parent actors. The number of actors is limited by hardware parameters.

The communication between the actors and the "A" application is one-sided and always takes place using the main actor for the selected activity. The introduced superior actor can communicate with application "A" by cyclically querying its availability (in parallel with the listening actors, a two-way black arrow from A to B in Figure 3) if it does not respond for a long time *T*. The actor's instance can only access the information included in the subordinate actors, but can send error messages to actors with higher priority.

The communication between the actors within the "B" application is carried out according to the following steps (always in the same order):

1) if an error occurs in application "A", information about this fact is passed to the superior actor, responsible for listening to the appropriate activity,
2) the sequence of functions calling the error in "A" is transferred to the relevant listening actors in "B"; every function within a given activity has one lower priority actor assigned to the parent,
3) if (after the occurrence) application "A" is not restarted for a specified period of time (parameter *T* selected experimentally), the superior actor ends listening for the given activity and records this fact in the report,
4) in the event of an unspecified error in "B" (no power, network access etc.), the ROOT actor is obliged to carry out attempts to restart the application.

The communication between programs is carried out according to the following rules:

- the activity launched in application "A" sends information about its state, which was forced by the selected scenario, assigned to its superior actor in the "B" application,
- the occurrence of an error is signaled by sending the application "B" the sequence of calls that caused it.

An important element of the work is the technical way of implementing the proposed solutions. Proposing a distributed test environment is related to handling communication between virtual machines and correct handling of errors that may occur due to the fault of the

virtual machine. For a distributed environment, at least one server instance should always be set up, which will be able to perform calculations when an error has occurred on one of the previously selected workstations.

Single Test Evaluation

Require: Generate the **S** set of m scenarios, choose the set of instances and set T.
Ensure: Error dataset.

1. **while** i $< m$
2. For sequence S_i generate the proper number of the activities.
3. Assign the activities to Actors and assign the functions to Subactors.
4. Evaluate the scenario on a virtualized Android OS.
5. Store the data about errors.
6. **if** Scenario S_{i+1} needs more Actors **then**
7. Create the additional set of Actors **end if,** $m := m + 1$
8. **end while**

4. Computational experiments

The research was carried out as part of one computing unit (Intel Core Xeon W CPU 14-core processor of 2.5 GHz, equipped with 128 GB of RAM) and in a distributed environment (each unit is equipped with an i7-8550U processor and 512 MB RAM). The completed tests were conducted for test applications: model (no errors) and reference (with errors in implementation). Comparison of programs with different code quality will allow to check the correctness of results indicated by the developed diagnostic tool. The analyzes were carried out using the Google Pixel 3 mobile device virtualization operating under the control of Android version 8.1.

The first test (Figure 4) concerns multi-criteria evaluation of the developed solution. The following criteria were adopted as the evaluation criteria: the time of research for a given set of scenarios and the RAM used. From the point of view of test automation, the study allows to estimate the cost and implementation time. As for the application that is being tested, the Adobe Photoshop Express was chosen, the number of generated scenarios is equal to 25,000, while the sequence length in each scenario is in the range from 5 to 50.

Analysis of the test environment's demand for RAM in the area of one computing unit shows that increasing the amount of memory speeds up calculations. This is due to the possibility of keeping more actors performing calculations per unit of time. The processing unit's processor load ranged from 86% to 95% during testing. Dissociating calculations into many instances can significantly reduce computation time; by 59.6% in the worst case. In the context of the practical implementation of the solution, increasing the number of instances significantly increases the cost of implementing such a system. The advantage of this approach is lower failure rate, because stopping the work of one unit does not interrupt the tests.

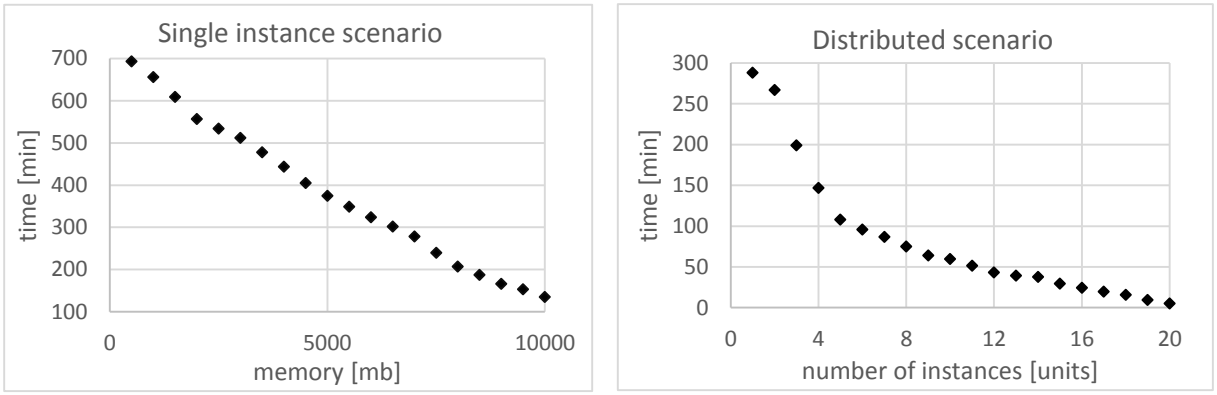


Figure 4. Dependence of the execution time on RAM memory (left) and the number of instances (right). Source: own work.

Subsequent tests include the use of the developed environment to check the integrity of the downloaded data. A model application was developed for the research, the task of which is to collect additional data in the form of services when changing activity. A total of 1000 scenarios were generated, in which attempts to navigate the user interface, stop and resume application operation, as well as to invoke functions in separate threads were performed, while downloading 1 GB of data.

Table 1.
Percent of the data downloaded from external source as a service while performing additional activities on different OS versions (scenario based on Figure 1)

KitKat	Lillipop	Marshmallow	Nougat	Oreo
94%	100%	92%	100%	100%

Source: own work.

The results of the tests in Table 1 show the disadvantage of Android fragmentation. Despite correct implementation, the tested application does not work correctly on all system versions. In the KitKat and Marshallow versions, the system repeatedly retrieves data when external events occur in the calling program.

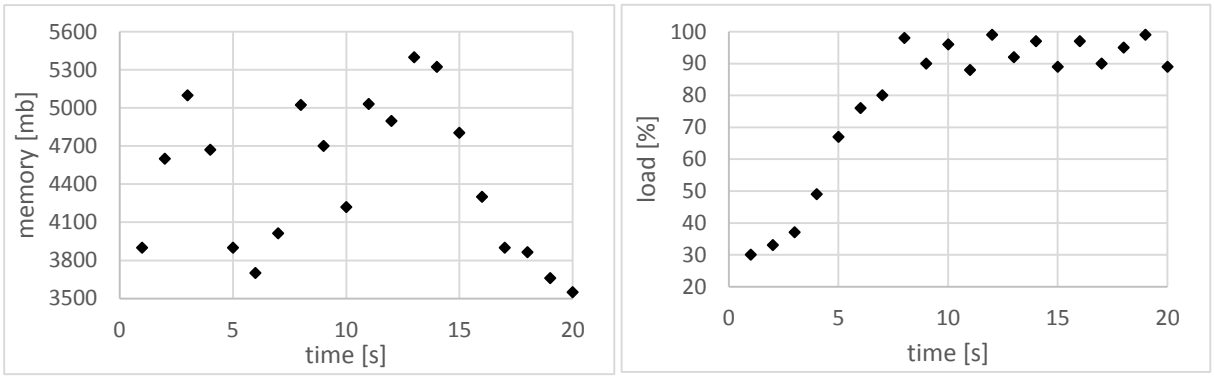
Table 2.
Percent of the data successfully saved in the SQL database (scenario based on Figure 2)

KitKat	Lollipop	Marshmallow	Nougat	Oreo
74%	79%	80%	100%	100%

Source: own work.

The next study was carried out using a reference application, the task of which was to save 1 GB of data to the database, if it was simultaneously downloaded from the Internet. In addition, 5 parallel services were started (each time after every 100 downloaded megabytes) in order to load the system. The study showed the expected *ConcurrentModificationException* and *NullPointerException* errors in the reference application. Moreover, it turned out that the mechanism of protection implemented by the manufacturer against the occurrence of a lack of

operational memory (as a result of running a large number of services) works efficiently in the Nougat and Oreo systems (Table 2).



a) b)

Figure 5. Dependence of the RAM memory (left) and the processor's load (right) on time in virtualized Google Pixel 3. Source: own work.

The last study concerned the use of the test environment to identify errors related to many threads trying to communicate with each other. Analysis of the vulnerable application for a set of 3000 scenarios generated by the *monkeyrunner* program (Monkeyrunner, 2019) was performed on the Oreo system. An application was used to transfer data between folders using multiple threads. In the reference application, 4 sequences were identified that led to an *ExecutionException* error for 10 GB of data. An additional aspect of the study was the analysis of device parameters. According to graph 5a in Figure 5, the occurrence of exceptions in different threads during copying resulted in a decrease in the number of data being copied. However, the high CPU load (graph 5b in Figure 5) remained high for another 10 minutes after the end of the test (the application threads were also terminated), which indicates the presence of unidentified system errors (the test for confirmation of behavior was performed tenfold.

5. Summary

For the purposes of the research, it was possible to develop a tool that effectively detects selected errors, caused by improper implementation of parallel operations in applications designed for Android. The proposed hierarchical structure of the solution based on the actor model guarantees a higher percentage of code coverage and review of more scenarios in a time shorter than in the case of open solutions, implementing calculations sequentially. The reliability tests of properly prepared applications were carried out to check the quality of the developed environment. An interesting conclusion from the research is the fact, that it is possible to identify errors in applications or system based on the characteristics of the memory demand and the CPU load. This research direction should be continued, because the analysis of

the characteristics of selected device parameters can significantly reduce the space of scenarios, which should be taken into account in order to identify specific and rare errors in the code. Another direction of research that should be considered is to develop a way to get practical test scenarios that simulate natural user behavior or, on the other hand, constitute sequences of potentially dangerous activities (e.g. by identifying threats based on UML diagram analysis) for stable program operation. The development of a tool that performs reliability diagnostics, combined with penetration tests, is an interesting issue in the context of analysis of sensitive data leaks.

References

1. Han, D. et al. (2012). *Understanding android fragmentation with topic analysis of vendor-specific bugs.* 19th Working Conference on Reverse Engineering. IEEE, pp. 83-92.
2. Wang, H., Li, H., Li, L., Guo, Y., & Xu, G. (2018). *Why are Android apps removed from Google Play?: a large-scale empirical study.* Proceedings of the 15th International Conference on Mining Software Repositories, ACM, pp. 231-242.
3. Guide to Background Processing. Retrieved from https://developer.android.com/guide/background, 25.09.2019.
4. Agha, G.A., Mason, I.A., Smith, S.F., & Talcott, C.L. (1997). A foundation for actor computation. *Journal of Functional Programming, 7, 1,* pp. 1-72.
5. Amalfitano, D., Amatucci, N., Fasolino, A.R., & Tramontana, P. (2015, August). *AGRippin: a novel search based testing technique for Android applications*. Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile, ACM, pp. 5-12.
6. Arifiani, S., & Rochimah, S. (2016). *Generating test data using ant Colony Optimization (ACO) algorithm and UML state machine diagram in gray box testing approach*. 2016 International Seminar on Application for Technology of Information and Communication (ISemantic), IEEE, pp. 217-222.
7. Concurrent Class (2019). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/ConcurrentModificationException.html, 20.03.2019.
8. Execution Class (2019). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/ExecutionException.html, 20.03.2019.
9. Haller, P. (2012). *On the integration of the actor model in mainstream technologies: the scala perspective.* Proceedings of the 2nd edition on Programming systems, languages and applications based on actors, agents, and decentralized control abstractions. ACM, pp. 1-6.

10. Hu, C., & Neamtiu, I. (2011, May). *Automating GUI testing for Android applications*. Proceedings of the 6th International Workshop on Automation of Software Test, ACM, pp. 77-83.
11. Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., & Stavrou, A. (2012, June). *A whitebox approach for automated security testing of Android applications on the cloud.* Proceedings of the 7th International Workshop on Automation of Software Test, IEEE press, pp. 22-28.
12. Mao, K., Harman, M., & Jia, Y. (2017). Robotic testing of mobile apps for truly black-box automation. *IEEE Software, 34, 2,* pp. 11-16.
13. Monkeyrunner (2019). Retrieved from https://developer.android.com/studio/test/monkeyrunner, 20.03.2019.
14. Null Pointer (2019). Retrieved from https://docs.oracle.com/javase/7/docs/api/java/lang/NullPointerException.html, 20.03.2019.
15. Takala, T., Katara, M., Harty, J. (2011). *Experiences of system-level model-based GUI testing of an Android application.* Fourth IEEE International Conference on Software Testing, Verification and Validation. IEEE, pp. 377-386.
16. Tretmans, J. (1999). *Testing concurrent systems: A formal approach. In International Conference on Concurrency Theory.* Berlin-Heidelberg, Springer, pp. 46-65.
17. Wei, L., Liu, Y., Cheung, S.C., Huang, H., Lu, X., & Liu, X. (2018). *Understanding and detecting fragmentation-induced compatibility issues for android apps*. IEEE Transactions on Software Engineering.
18. Yang, Z., Guo, Y., & Chen, X. (2017, May). *DroidBot: a lightweight UI-guided test input generator for Android.* 2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C), IEEE, pp. 23-26.
19. Yang, Q., Li, J.J., & Weiss, D.M. (2009). A survey of coverage-based testing tools. *The Computer Journal, 52, 5,* pp. 589-597.
20. Yang, W., Prasad, M.R., & Xie, T. (2013). *A gray-box approach for automated GUI-model generation of mobile applications.* International Conference on Fundamental Approaches to Software Engineering. Berlin-Heidelberg: Springer, pp. 250-265.