Evaluation of an Automated Testing *Framework*: A Case Study

Evaluación del *Framework* de Pruebas Automáticas: Un Caso de Estudio

Abel Méndez-Porras[1], Jorge Alfaro-Velasco[2], Alexandra Martínez[3]

Méndez-Porras, A; Alfaro-Velasco, J; Martínez, A. Evaluation of an Automated Testing *Framework*: A Case Study. *Tecnología en Marcha*. Vol. 33-3. Julio-Setiembre 2020. Pág 3-12.

https://doi.org/10.18845/tm.v33i3.4372

Fecha de recepción: 17 de agosto de 2019
Fecha de aprobación: 6 de noviembre de 2019

1 Costa Rica Institute of Technology. Computer Deparment. Costa Rica. E-mail: amendez@tec.ac.cr
2 Costa Rica Institute of Technology. Computer Deparment. Costa Rica. E-mail: joalfaro@tec.ac.cr
3 University of Costa Rica. Computer and Information Science Department. Costa Rica. E-mail: alexandra.martinez@ecci.ucr.ac.cr

Keywords

Software quality assurance; automated software testing; mobile applications; SURF; user-interaction features; software engineering.

Abstract

Developing mobile applications without defects or in a minimum number is an important challenge for programmers and quality assurance teams. Automated software test can be the key to improve the traditional manual testing, often time consuming and repetitive. Mobile applications support user-interaction characteristics, which are independent from the application logic. They include content presentation or navigation features, such as scroll or zoom into screens, and a rotating device.

In this paper, an automated testing framework is proposed and evaluated. This framework integrates user- interaction features, historical bug information, and an interest points detector and descriptor to identify new bugs. It has shown that it works well detecting bugs associated with user-interactions.

Palabras clave

Aseguramiento de la calidad del software; pruebas automatizadas de software; aplicaciones móviles; SURF; interacciones del usuario; ingeniería de software.

Resumen

El desarrollo de aplicaciones móviles sin defectos o con un número mínimo es un desafío importante para los programadores y los equipos de control de calidad. Las pruebas automatizadas de software pueden ser la clave para mejorar las pruebas de software manuales tradicionales, que a menudo requieren mucho tiempo y son repetitivas. Las aplicaciones móviles soportan funciones de interacción con el usuario, que son independientes de la lógica de la aplicación; entre ellas se incluyen funciones de presentación de contenido o navegación, rotación del dispositivo como desplazamiento, aumento o disminución de las pantallas.

En este documento, se propone y evalúa una herramienta de pruebas de software automatizada. La herramienta integra funciones de interacción con el usuario, información histórica de defectos y un detector y descriptor de puntos de interés para identificar nuevos defectos.

La herramienta demostró que funciona bien detectando defectos relacionados con las interacciones con el usuario.

Introduction

Capabilities of mobile devices, such as CPU (Central Processing Unit) computation, memory, RAM (Random Access Memory) size, and many more have been improved. In few years these devices have increasingly replaced traditional computers [20]. Mobile applications are very popular and play a strategic role in the activities of society. The devices that run Android applications are the largest majority, currently corresponding to 85% of the devices sold worldwide [19].

The quality of mobile applications is lower than expected due to rapid development processes, where the activity of software testing is neglected or carried out in a superficial way [2]. Zaeem

[24] conducted a bug study on 106 bugs drawn from 13 open-source Android applications. Hu [11] conducted a bug mining study from 10 Android applications and he reported 168 bugs. For the trial work to which this article refers, a bug study from 38 open-source Android applications was conducted, and 79 bugs were reported.

There is a wide variety of mobile devices and versions of operating systems, known as fragmentation [10], [12], [13], [14]. Fragmentation represents a testing difficulty since mobile applications may behave differently regarding usability and performance, depending on the device they are run on. According to Muccini *et al.* [17], the main challenges of testing mobile applications are related to their contextual and mobile nature.

Manual testing is the main technique used for testing graphical user interfaces of mobile apps. Manual testing is often tedious and error-prone [8]. Mobile applications are small (code size), are event-centric, and have a simple and intuitive GUI (Graphical User Interface). According to Yang *et al.* [23], these characteristics make them suitable for specific automated testing techniques.

In order to obtain higher quality mobile applications, greater attention should be devoted to testing throughout the development process, and effective models, methods, techniques, and tools for testing should be available for testers [2]. Automation can play a strategic role in ensuring the quality of mobile applications.

Developing mobile applications without defects or with the minimum number of defects is an important challenge for programmers and quality assurance teams. As the testing process is often tedious, test automation can be the key to alleviating such heavy activities [20].

In the literature several approaches to address the automated testing in mobile applications can be found. AMOGA [20] is a strategy that uses a hybrid, static-dynamic approach for generating a user interface model from mobile apps for model-based testing. GATS [7] is a tool that uses finite-state machine to learn a model of the app during testing; then it uses the learned model to generate user inputs or system events to visit the rest states of the app, and at last, it uses the result of the input to refine the model. Virtual Test Engineer [3] has been developed to convert the UML (Unified Modeling Language) sequence diagram into Android APK (Android Application Package) to test Android mobile applications. A model-based test generation methodology has been proposed to evaluate the impact of the interaction of the environment, the wireless network, and the app configurations on the performance of a mobile streaming app and thereby, on the experience of the end user [1].

The research described here aimed to propose an automated testing framework for mobile applications based on user-interaction features and historical bug information. A user-interaction feature is an action supported by the mobile platform and associated to content presentation or navigation. This action is independent of the application´s logic [24].

The framework proposed is organized in four components: an exploration environment, an inference engine, a bug analyzer, and a test storage database. A case study was conducted to evaluate the effectiveness of the automated testing framework.

The following are the research contributions: First, digital image processing and GUI information were combined in a new way to find bugs in mobile applications generated by user-interaction features; second, based on historical bug information, the selection of sequences of events to be used as test cases was improved, and third, the proposed framework was evaluated through a case study.

Automated testing framework

Designing an automated testing framework that can identify bugs related to user-interaction features requires resolving four main challenges: the automated exploration of mobile applications, the automated introduction of user-interaction features during the automated exploration, the automated analysis of bug identification, and the storage and use of historical bug information to improve bug detection. An overview of the automated testing framework for mobile applications based on user-interaction features and historical bug information [15], [16] is shown in figure 1. Each of the components of the proposed framework are represented here.

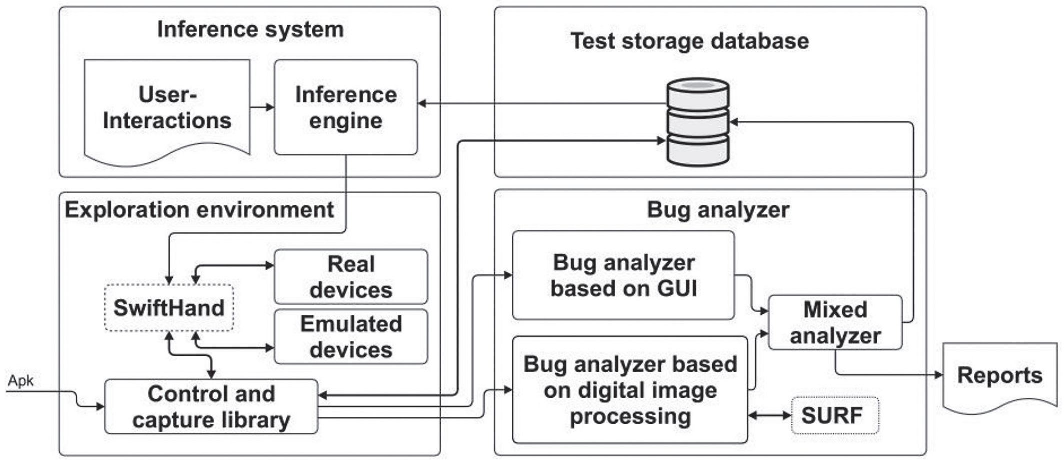

Figure 1. Automated testing framework components

A. Exploration Environment

Automated exploration is one of the more complex tasks in the automated testing of mobile applications. The decision of which method to use for automated exploratory events within applications will have a direct effect on the results obtained. The exploration environment launches an application using the APK file. After starting the application, the exploration environment navigates through the application's different screens. The aim is to achieve the widest possible branch coverage of the application (quantity of screens visited, and events executed).

The exploration environment is responsible for creating a model of the application. This model is then used to explore the application automatically For this task, the tool SwiftHand [8] was used. This tool uses learning-based testing to explore the application and creates amodel. To fulfill this task SwiftHand was modified to allow the inclusion of user-interaction features while it explored the application model. The function of capturing images before and after the execution of the user-interaction features was added; these images were sent to the bug analyzer to find potential bugs in the application. Additionally, the function of capturing GUI information was enabled before and after the execution of the user-interaction features, to be also sent to the bug analyzer to find potential bugs.

B. Inference Engine

The simplest way to introduce user-interaction features while an application is being automated explored is using a random technique. However, this technique can conduct analysis in application states under testing that have little relevance in relation to the appearance of bugs. More information is needed in regard to when is the best time to automate the introduction of the user-interaction features to best take advantage of the automated exploration of the applications.

In the inference engine, two strategies were implemented:

- Random.— The modified Swifthand device was run, and while automatically exploring the application, user-interaction features were randomly introduced.
- Frequency of bugs by widget.— Widgets were identified in the current activity (window) of the application, and the historical information of bugs was analyzed to find out if these widgets had presented major bugs and which user-interaction features had been associated with these bugs. If these types of widgets had presented bugs in other applications, the inference engine introduced the same user interaction features that reported the bugs determined to find bugs associated with the same widgets in the new applications.

C. Bug Analyzer

All the user interaction features treated have an oracle; it allows to know which the desired result is as an user-interaction is introduced in the application under testing. In this part of the research, a bug analyzer based on digital image processing was implemented, as well as another one based on GUI information. Finally, a technique tocombine the results of these two analyses was applied in order to obtain greater precision in the detection of bugs associated with user-interaction features.

Bugs analysis can be described as follows:

The bug analyzer receives a set of image pairs and GUI information from the inference engine. These images and GUI information are captured before and after the introduction of each user-interaction feature. The interest points detector and descriptor (SURF) [4] finds interest points in each image. A comparison among interest points found in the images results in a percentage of similarity. If the percentage of similarity is lower than a set threshold, it is a possible indicator of a bug. GUI information is used then to identify widgets that generated the bug. If widget's properties have changed after the introduction of the user interaction feature, that is a possible indication of a bug as well. Interest points and GUI information are used to determine whether bugs were produced by the user interaction feature. If bugs are found, information about is stored in the repository of historical bug information.

The different interest points obtained from the analysis done with SURF contain coordinates. Given that the images examined for analyzing the effectiveness of the user-interaction feature have exactly the same resolution as the size of the screen of the mobile device or emulator used while exploring the app, these coordinates can serve to locate the widget that is apparently showing a bug exactly in the user interface layout.

To do this the absolute coordinates of each widget and the properties of width and height of each one are determined using the GUI structure of the application. Knowing this data, widgets can be exactly located in the screen; in the same way, the exact amount of interest points in the area of each widget can be determined. The technician decides then if the amount of these is high enough to consider that the widget might be showing a bug.

On the other hand, in most cases, it is perfectly recognizable when a widget is showing a bug by obtaining and comparing its properties before and after the user-interactions are processed. Since the absolute coordinates of each widget have been previously determined, they can be used to confirm that the widget considered bugged is indeed a widget that is showing some kind of bug.

D. Test storage database

This database stores all the test cases created by user interactions and all historical bug information. Information about the event sequences and user interactions is stored in the database. Also, GUI information obtained before and after each user-interaction introduced is stored in the database.

E. Flowchart of the automated testing framework

Figure 2 shows the flowchart of the Automated Testing Framework. The Application Explorer launches the application to be tested in the actual devices or emulated devices. It launches the application using the (.apk) file. After starting the application, it navigates through the application's different screens. The Inference Engine inserts user interaction features while the application is explored automatically. The Analyzer receives a set of image pairs and GUI information from the Inference Engine. Images and GUI information are captured before and after the introduction of each user interaction feature. If bugs are found, information about the bugs is stored in the repository of historical bug information.

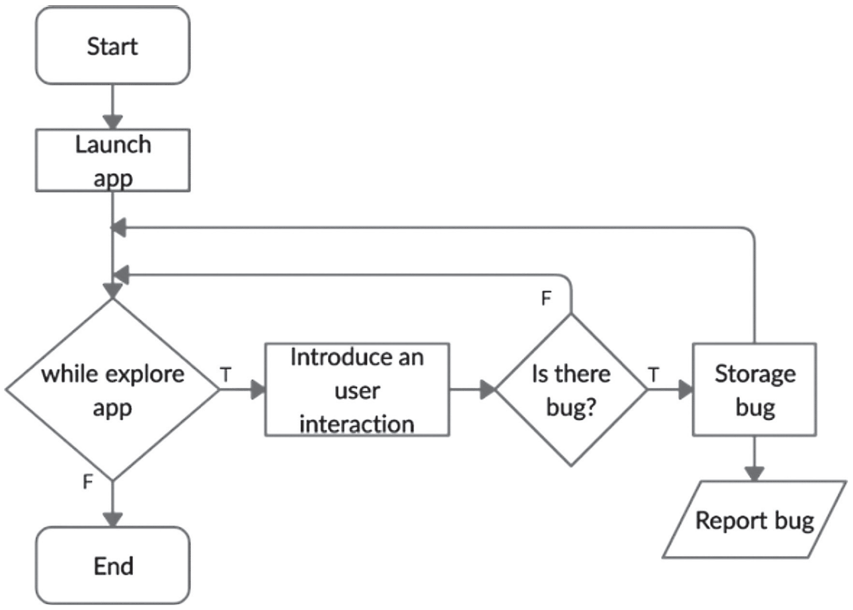

Figure 2. Flowchart of the application test and bug detection

Evaluation of the automated testing framework

In this research, the goal was to evaluate the complete platform of Automated Testing

Framework on its efficacy to find bugs. That meant probing its functions all the way through each component: the exploration environment, the inference engine, the bug analyzer, and test storage database.

A confusion matrix is a simple but effective way to visualize the successes and the errors that a classifier has made. Metrics based on the confusion matrix [6] such as accuracy [21], precision [18], recall [18], and F measure [5] were selected to evaluate the bug analyzer.

Thesamongsthesior-The evaluation of the framework proposed consisted of two phases. In the first phase, bugs derived from user-interaction features were detected using automated testing. In the second phase, a statistical comparison between manual and automatic testing was conducted.

A. Double rotation test for 20 minutes within 51 mobile applications

For the test, 100 mobile applications were randomly selected from the F-Droid catalog. They were tested in the SwiftHand tool to probe if it could launch and explore them automatically. Of the 100 applications selected, the SwiftHand tool managed to launch and automatically explore 51 mobile applications. These 51 applications were those used in this study.

A confusion matrix was created by comparing the bugs reported by automated testing based on SURF and the bugs checked by manual testing. They were compared using four different similarity thresholds. Table 1 shows the four thresholds in similarity percentages (column 1): the true positive (column 2), the false positive (column 3), the true negative (column 4), and the false negative (column 5).

Four metrics were applied: accuracy (column 6), recall (column 7), precision (column 8), and F measure (column 9).

As it can be seen from table 1, the bug analyzer proven (classifier) offers better overall results (in terms of the F measure) when the threshold is set at 90 similarity percent. This is also evident from figure 3, where the closest point to the (0,1) perfect classifier corresponds to the 90 percent thresholds. Figure 3 shows the ROC curve of our bug analyzer, when varying the similarity percentage threshold in this case study.

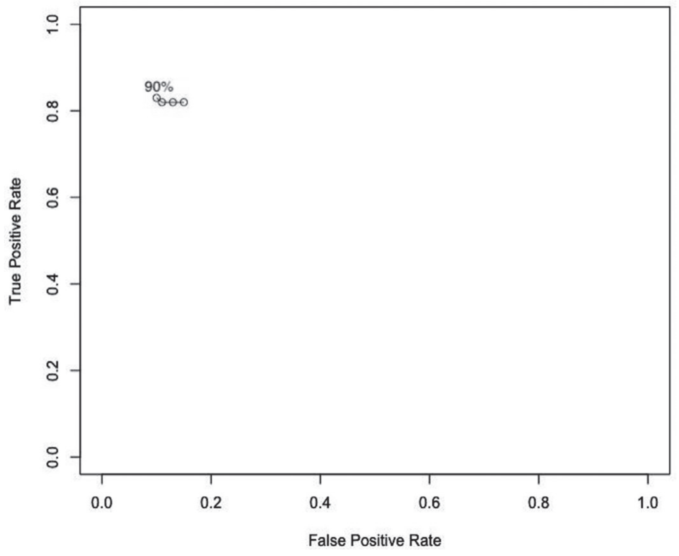

Figure 3. ROC curve described by the classifier in case test 3, when varying the similarity threshold (as in table 1)

It is noticeable in this figure that the bug analyzer performed right in terms of proximity to the (0,1) corner, and described a smooth curve, which means that the classifier did not show abrupt changes in behavior when varying the threshold. The false negative is less than 4% of the sample data and the false positive is less than 12% of the sample data; that explains why the points on the ROC curve are so close.

Table 1. Confusion matrix obtained from case study 3, using four different similarity thresholds

Threshold	TP	FP	TN	FN	Accuracy	Recall	Precision	F
90.0	351	155	1467	74	0.89	0.83	0.69	0.75
92.5	352	181	1438	76	0.87	0.82	0.66	0.73
95.0	354	213	1400	80	0.86	0.82	0.62	0.71
97.5	359	234	1273	81	0.80	0.82	0.61	0.70

Statistical comparison between manual and automated testing

The main purpose in this research was to understand whether there was a significant difference or not in the number of bugs reported under two different testing techniques, automated and manual (see table 2). The independent variable was the *testing technique* and the dependent variable was the *number of bugs*. The object of experimentation was *the mobile application running on an specific device*. The null hypothesis was that the number of bugs identified by means of the automated testing technique were the same as the bugs identified by means of the manual testing technique. The alternative hypothesis was that they were not.

The dependent variable consisted in related bugs groups. The dependent variable was measured at a continuous level (specifically ratio).. Related groups indicate that the same objects are present in both groups. Each object has been measured on two occasions on the same dependent variable. The objects, “the mobile applications running on specific device”, were tested using “automated testing” and “manual testing”.

The SURF descriptor and detector threshold was fixed at 92.5% of similarity, based on the findings obtained in the case studies.

A hypothesis test for matched or paired samples is used when two samples are drawn from the same set of objects [9]. It is recommended to use the paired t-test when two samples resulting from repeated measures are compared [22].

A Shapiro-Wilk normality test was applied to conclude whether the data were normally distributed or not. The *p*-value obtained was 0,01384; therefore, the assumption of normality was rejected.

In a homogeneity of variance test, the null hypothesis states that the difference in mean is equal to 0 and the alternative hypothesis states that the difference in mean is not equal to zero.

There was not normality in the data; for that reason, a parametric test couldn´t be applied. Hence, a non-parametric test was chosen: the Wilcoxon Signed-Rank test, which is used to compare two sets of scores that come from the same participants. This occurred within the data because mobile applications were subjected to more than one condition (automated and manual testing). The assumptions of the Wilcoxon SignedRank test are that data are paired and come from the same sample; each pair is chosen randomly and independently, and the data are measured on at least an interval scale.

Since *p*-value was 0,0092 for the alternative hypothesis (“greater”), and was less than the 0,05 significance level, the null hypothesis was rejected. From the sample data there was sufficient evidence to conclude that bug measurement using the automated framework is, on average, more effective than measurement using manual testing (within a 5% level of significance).

Table 2. Manual testing versus automatic testing

	Manual		Automatic	
Name	User-Interactions	Bugs	User-Interactions	Bugs
An.stop.10	13	3	39	12
android.androidVNC.13	6	5	41	0
byrne.utilities.converter.1	7	4	46	23
ch.fixme.cowsay.5	6	1	42	11
com.anoshenko.android.mahjongg.14	7	4	37	0
com.ath0.rpn.17	5	0	43	0
com.chmod0.manpages.3qqqq	8	5	42	24
com.drismo.17	6	0	39	0
com.gcstar.scanner.1	5	5	40	29
com.ginkel.hashit.25	5	2	44	3
dk.andsen.asqlitemanager.17	7	4	44	17
nindroid	6	0	41	0
org.jessies.mathdroid.293	5	0	42	23
org.projectmaxs.main.2000100151	6	2	41	19
trolly	5	2	39	1
Total	97	37	620	162

Conclusions and future work

The behavior of mobile applications is affected by different types of user events: events produced through GUI, events generated by the device hardware platform, events from the Internet, and events of mobile phones. These types of events are likely to generate bugs in mobile

applications. The periodicity with which user events are triggered and the variety of events make the process of automatic software testing in mobile applications difficult.

Use of historical bug information to find bugs in mobile applications is complex because it requires storing information about all the bugs detected each time that other applications are tested. The first difficulty is knowing when there is enough information to infer when to enter a user interaction during the testing process. The second difficulty is that it is necessary to obtain applications that are being developed and have not yet been tested to increase the probability of storing bugs associated with user interaction features.

A top-down technique was used to design the automated testing framework. The advantage of using this technique was that, through a formal process, the architecture of framework was designed to foster further research progress. This architecture was organized into four components: an exploration environment, an inference system, a bug analyzer, and a test storage database.

Evaluation of the automated framework showed that it works well detecting bugs associated with user-interactions, on average, better than manual testing (within a 5% level of significance). In addition, it has the advantage of reporting the similarity percentage of the window before and after executing a user-interaction. It also shows the original images of the window before and after the user-interaction is executed. Finally, can see the images where the differences between the window before and after the execution of the user-interaction are highlighted.

There are defects associated with user-interactions in mobile applications that reduce their usability, although they do not cause the application to crash. For example, there is an open dialog in the window for the user to select an option, but when an interaction is executed, the dialog closes, and the user must re-select the action so that the dialog is shown again. In this case, the application has not stopped but has generated an additional task for the user. Obviously, reducing this type of bug in applications will allow them to offer better usability and be more user-friendly.

As conductors of this study, we plans to develop our own exploration environment because we need more control over the event-sequences automatically entered in the applications under test. Also, we would like to improve the exploration coverage in each application under test.

References

[1] M. Al-Tekreeti, K. Naik, A. Abdrabou, M. Zaman, and P. Srivastava, “A methodology for generating tests for evaluating user-centric performance of mobile streaming applications.” *Communications in Computer and Information Science*, 991: 406–429, 2019.

[2] D. Amalfitano, A.R. Fasolino, and P. Tramontana, “A gui crawling-based technique for android mobile application testing,” in *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW, 2011*, pages 252–261.

[3] R. Anbunathan and A. Basu, “Automation framework for test script generation for android mobile,” *Advances in Intelligent Systems and Computing*, 731: 571–584, 2019.

[4] H. Bay, T. Tuytelaars, and L Van Gool. “Surf: Speeded up robust features,” in *Computer Vision - ECCV* , vol. 3951, pages 404–417, 2006.

[5] C. Cagatay, “Performance evaluation metrics for software fault prediction studies,” 9 01 2012.

[6] N.V. Chawla, “Data mining for imbalanced datasets: An overview,” in *Data Mining and Knowledge Discovery Handbook*. Boston, MA: Springer, pages 853–867, 2017.

[7] T. Chen, T. Song, S. He, and A. Liang, “A gui-based automated test system for android applications,” *Advances in Intelligent Systems and Computing*, 760: 517–524, 2019.

[8] W. Choi, G. Necula, and K. Sen, “Guided gui testing of android apps with minimal restart and approximate learning,” *ACM SIGPLAN Notices*, 48(10): 623–639, 2013.

[9] S. Dean and B. Illowsky, *Collaborative Statistics*. Houston, Texas: Rice University, , 2012.

[10] H.K. Ham and Y.B. Park. “Mobile application compatibility test system design for android fragmentation,” *Communications in Computer and In-formation Science*, 257 CCIS: 314–320, 2011.

[11] C. Hu and I. Neamtiu, Automating gui testing for android applications,” in *Proceedings - International Conference on Software Engineering, 2011*, pages 77–83.

[12] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, “Testdroid: Automated remote gui testing on android,” in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia, MUM 2012.*