International Journal of Scientific Research in Information Systems and Engineering (IJSRISE)
Volume 1, Issue 2, December-2015. ISSN 2380-8128

Automated Black-Box GUI Testing for Revealing System Bugs in Mobile Applications

Özlem Muslu, Department of Information and Communications Technology, Netaş
Istanbul, TURKEY
E-mail: omuslu@netas.com.tr
Yunus Mete, Department of Information and Communications Technology, Netaş
Istanbul, TURKEY
E-mail: yunusm@netas.com.tr

Abstract - As smartphones came to dominate mobile communications, mobile application world have advanced rapidly to address the mobile user needs, resulting with many applications that target these devices and filling app stores with an exponential rate. If these applications don't meet customer needs, i.e. if they are buggy, slow etc., they are bound to perished due to the intense competition among similar applications. This brings the need for thorough testing, particularly for graphical user interfaces. This paper aims to inspect how guided GUI testing is realized; specifically using model-based and model-learning methods. The main focus is to elucidate the shortcomings of current methods, such as automatically entering text inputs, deciding which action to trigger or creating test sequences. Such shortcomings are open to interpretation by artificial intelligence and machine learning techniques. By revealing aforementioned shortcomings this paper plans to raise new research questions in the area of software testing.

Keywords *– black-box testing, GUI testing, mobile testing, model based, model learning*

1. INTRODUCTION

The needs of the mobile community continuously change which brings the evolution of mobile communication technologies. Mobile phones are widely used; hence mobile application market grows constantly. Similar applications populate the app stores and if the developers wish to stand out of the crowd, they need to produce flawless applications.

Mobile applications are characteristically different from desktop applications. They are written for devices with relatively low processing power, little memory and smaller screens. This limits their abilities. Also, there are fewer widgets in one page when compared to traditional applications (see Figure 1). Many applications rely on 'sensed data'; cameras, internet connection, GPS sensor and so on. Furthermore, there is a huge amount of diversity between devices. They have different screen sizes, they are produced by various manufacturers, and even their operating systems are distinct. Mobile developers need to consider these facts when writing their applications, but the amount of possible combinations is colossal. This leads to a large portion of test conditions to be neglected. When these apps are released without being comprehensively tested, undesired behavior such as crashes may occur, and they receive low ratings in app stores. To avoid commercial loss, white box testing can be applied, but tests generated using source codes tend not to reveal bugs relating to GUI. If there was a tool that could automatically traverse an application by interacting with GUI widgets and save the interaction sequence, the testing process would be able to cover some of the cases discussed above. This is where GUI testing sets in. GUI testing is the automatic guided traversal of an application using GUI elements. It can be performed using an emulator, a device or even multiple devices. There are three main categories of GUI testing: random, model based and model learning testing.

Random tests aim to find crashes by firing widgets on the screen randomly. They work fully automatic and are easy to implement, yet the possibility of running same tests over and over is high [1]. Moreover, track of the action sequence that caused a bug can be lost. One example is Monkey Runner, a test tool integrated into Android SDK. There are also more directed versions of random testing such as Dynodroid [1].

Corresponding Author
Özlem Muslu, Department of Information and Communications Technology, Netaş
Istanbul, TURKEY
E-mail: omuslu@netas.com.tr

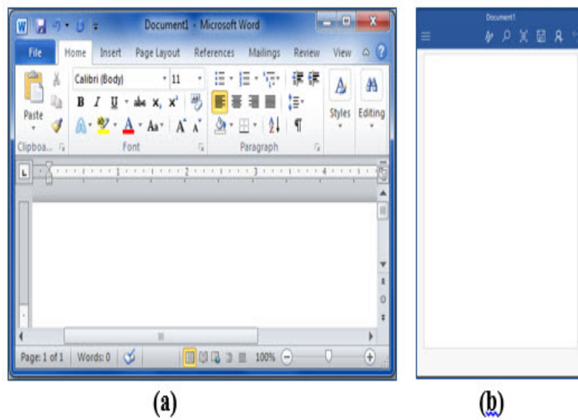

Figure 1:*Screenshots from Microsoft Word (https://products.office.com/en-us/word). Desktop application has many GUI elements whereas mobile application has only a few of them. (a): Microsoft Word in Windows 7 platform. (b): Microsoft Word on Android 5.1.1.*

Model based tests are a popular test system where a correct GUI model of the application should be given as an input. Using this model possible test sequences are generated. Since the user needs to create a GUI model of the application, these tools are generally not considered fully automatic. For the GUI model finite state machines or event flow graphs are utilized. [2].

Model learning systems are developed to automate the model building process so that users of the automation system wouldn't have to create a GUI model manually and update it every time a change in GUI occurs [2]. Model learning stage is the most computationally expensive part of these methods. Moreover, it is crucial to avoid loops that would cause revisiting a page.

A recent survey on GUI testing [3] conducted an empirical study with different Android GUI testing tools by letting each tool run an hour per application. They compared six tools including Android's Monkey, Dynodroid [1], A^3E [4], MobiGUITAR [5] and Puma [6]. The results on others showed that random tests produced higher statement coverage and triggered more failures. However, random tests do not support reproducible bugs, thus they may not be useful.

This paper will present challenges of systematic GUI testing. The work it takes into consideration includes model-based and model learning mobile testing mostly. Specifically, Swifthand [2], MobiGUITAR [5], AutoBlackTest [7], ICrawler [8] Puma [6] and A^3E [4] will be analyzed. Moreover, random methods for mobile applications [1], evolutionary [9] and data-based [10] methods for desktop applications are also included. This section presented motivations for GUI testing and introduced different approaches used for GUI testing. In Section 2, model generation will be inspected. Two main aspects, model exploration strategies and state equivalence will be discussed. In Section 3, methods for automatically entering text inputs will be analyzed. In Section 4, test suite generation methods will be explained and in Section 5, conclusions will be drawn.

2. MODEL GENERATION

A model is a connected structure of the application's GUI. It constitutes of states and transitions between those states. State and transition may carry slightly altered meanings, but in the most general sense, a state is the abstraction of actions and a transition denotes a shift from one state to another by means of a fired event. Here, a fired/triggered event or an action may represent taps, scrolls, or any other gesture and these terms are used interchangeably throughout this paper.

Model generation is traversing an application such that the traversal creates a correct definition of the GUI in form of a model. There are two main difficulties when generating the model: How to traverse the application so that the generated model can converge to the actual model as fast as possible and how to decide if two states are equivalent so that the generated model is a correct representation of the actual one. This section will seek answers to above inquiries.

2.1. Model Exploration Strategies

An exploration strategy determines which state and which action at that state should the system fire next. Exploration strategies are important for a crawler either to achieve maximum state/code coverage or to act as a genuine user so that it can reveal the crashes app users are most likely to encounter. Several approaches are proposed including depth first search, user-statistic based search, reward based search and so on. This section will elaborate on particular strategies different papers use.

MobiGUITAR [5] employs a list for keeping the unfired widgets and adds new fireable elements to that 'task list'. This is es-

sentially an implementation of breadth first traversal.

A^{3}E [4] has two different strategies: Targeted and Depth-First. In targeted strategy static byte code analysis is performed in order to obtain a primitive model of the application. Following, the application is run and targeted exploration is applied on the primitive model for systematic exploration. In Depth-First approach, depth first exploration is utilized and all GUI events are fired.

Swifthand [2] has a list of unexplored states from which it picks a random state. The action is also picked randomly. The reasoning behind picking random states and events are based on their experiments. They utilized diverse heuristics for the task but no heuristic surpassed others in effectiveness.

Puma [6] picks an event according to four different strategies which users can specify using PUMAScript, a Java-based scripting language created for Puma. These strategies are sequential exploration, maximizing the types of elements clicked, imitations of actual users and using static analyses to reveal common bugs. The authors use depth first search to pick the state. In sequential analysis, every element in every state is fired, starting from the top of the page, stopping at the bottom. When maximizing the types of fired elements, information on the already fired elements' type is utilized, e.g. if a button from ButtonClass was never pressed before, this strategy aims to press that button. Imitations of actual users are acquired either by using authors' insights or actual traces from users. Detecting an HTTP request and pressing to the button that would fire it is given as an example of a utilization of static analyses.

AutoBlackTest [7] uses a reinforcement learning technique, Q-Learning to decide on which action to trigger. Firstly, it selects a random state from the state space. It assigns a reward to the action with respect to changes in GUI. If the action induced many changes in the GUI, the reward is high; if the contrary takes place, the reward is low. The authors also add a heuristic to this immediate reward function, stating that a low-reward action such as filling in a text box may bring higher rewards in the future. Thus, Q-values depend both on "immediate utility and the utility of functions that can be executed in the future."

2.2. Checking for State Equivalence

A critical part of GUI crawling is determining state equivalence; whether the newly encountered state was seen before. Establishing state equivalence avoids re-traversing a previously explored state and forms the factual GUI model rather than a possibly infinite model.

Comparing the new page with the states already existing in the model is an arduous task. One can use the information in the page, such as types of widgets, their location etc. An abstraction of the state is usually established before comparison because the content of the screen may change while the state remains the same. An example is a news application where news changes every time page is reopened (see Figure 2). However, such abstraction may still not be enough to determine if two states are actually same. A list where one can add new items can be used as an illustration (see Figure 3). This approach is also troublesome because two different states with distinct behaviors can be visually similar. This leads us to methods that use the behavior of the state as the distinctive factor, i.e. if two states are considered to be the same but same widgets steer the application to different states, those states are different. In this section similarity measures used by different papers will be explained.

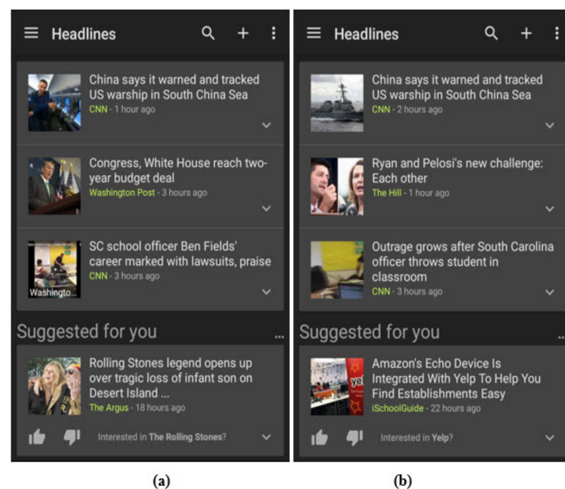

Figure 2: *Two screenshots from Google's News & Weather (*https://play.google.com/store/apps/details?id=com.google.android.apps.genie.geniewidget&hl=en*) application. (a) and (b) are from the same page, taken one hour apart. If an abstraction is not performed and content of the news is included when comparing states, every update to the page wouldg bring a new state leading to infinite state space and previously seen states could not be reopened.*

In MobiGUITAR [5] ID and type properties determine the state equivalence. If all widgets in two states are not equivalent, states

are not considered equivalent as well.

AutoBlackTest [7] is similar to MobiGUITAR in the sense that it also checks the occurrence of same widgets with some level of abstraction. In AutoBlackTest, this abstraction corresponds to widget type and large number of properties such as class id, text, if it is enabled/editable etc. Furthermore, some modifications in those widgets don't necessarily imply the states are not equivalent. They also debate the problem of treating different states as equivalent but express that possible different behavior of these states are significant only if these cases are frequent in the model.

Puma [6] bases state equivalence on cosine similarity. They encode both states' widgets as a vector; compute cosine similarity using this vector and use hard thresholding to determine if states are equivalent

In Swifthand [2] bounding boxes and screen coordinates of enabled user inputs are considered as the initial similarity metric. If a state is discovered to be same as one of the states observed before, they are merged in the same state. However, if this merger results in a nondeterministic state machine (i.e. as same events lead to different states), the model is relearned using a passive learning algorithm derived from [11]. The algorithm repeatedly merges states to generalize the model using a prefix-tree acceptor. If merging leads to a nondeterministic model child states of merged states' transitions are tried to be merged in a recursive manner. If that fails due to different states are tried to be merged, original states are restored.

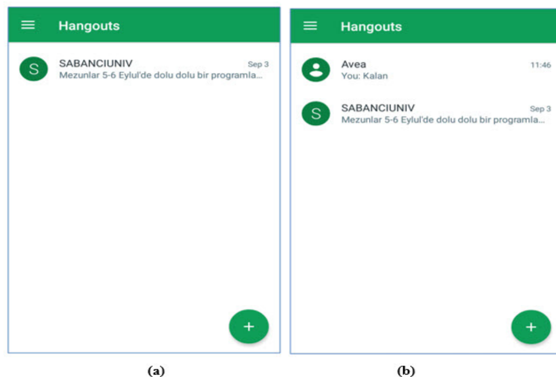

Figure 3: *Two screenshots from Google's Hangouts (* https://hangouts.google.com/ *) application. In (a) only one message is seen whereas in (b) two messages are present. If they are assumed to be same states, one-by-one comparison of the elements is not viable. If they are acknowledged as different states, every new message would bear one more state, leading to infinite state space.*

3. TEXT INPUTS

Text inputs cannot be overlooked since for some events to be triggered, they need to be correctly specified (e.g. login screens). The current approach for such cases is proiding an interface for the automation tool's user to specify such inputs. Puma [6] employs this approach and if the required inputs are not found, they stop exploring the app. In addition to asking for the user to enter text inputs, random text generation can also be applied. In Dynodroid [1], the part of the program that selects which event to fire is discouraged to select text boxes, but the part that executes events selector sent is required to populate all text boxes in the current UI before it executes the selected non-text event. Swifthand [2] also allows pre-defined user strings, and if not present, generates a random string. ICrawler [8] uses a dummy string based on the keyboard type (numeric/email address etc). AutoBlackTest [7] defines some predefined literals to handle text inputs. They associate a label to available input widgets and fill in the necessary parameters using predefined literals. For example, if the label is birthday, AutoBlackTest fills in the necessary date information.

4. TEST SEQUENCE GENERATION

The purpose of model generation is setting up a model that can later be used to create a test sequence. If all test sequences were generated using this model there would be astronomic number of test sequences to execute. Hence, the following step in model-based test systems is to restrict the number of test sequences by eliminating similar test suites. There are also alternative approaches other than using a model to perform GUI testing. These approaches may simply traverse GUI as if it was creating a model [12], or use actual usage profiles to traverse the application [10], use system events as a way of traversal [1] or use genetic algorithms to generate the test suites [9]. This section will explain how test suites are generated using the previously generated models and what kind of approaches are followed when models are nonexistent.

MobiGUITAR [5] samples the huge number of possible event sequences using pair-wise edge coverage criterion: "*all pairs of adjacent edges (events)* need to be exercised together. To this end, we create pairs of all edges in our state machine that are adjacent

to a node. And for each pair, we generate a test case that is a path in the state machine from the start state to the pair being covered."

Puma [6] clusters the GUI models and draws the conclusion that there are relatively small number of clusters (about 40) when it comes to different GUI models. It is an open research question whether these clusters would share similar sequences for crashes.

In [1], system events are used to guide the automation program through the GUI of application under test. This kind of usage allows not only to find bugs related to the GUI, but also bugs caused by system events such as crashes due to lost connection to network. Their algorithm runs on an observe-select-execute cycle. Observer is responsible for computing the set of relevant UI, broadcast receiver and system service events. Selector selects events according to three different strategies: Frequency, UniformRandom and BiasedRandom. Frequency selects the least frequently selected event. UniformRandom selects events uniformly at random. BiasedRandom keeps the frequency of events selected before but it also respects the relevancy of an event; it assigns different scores for UI, system and text entering events. After Selector is done with one of the above strategies, Executor executes commands it receives from both humans and the selector.

In [9] genetic algorithms are utilized to imitate novice users. Novice users are defined as those who don't have much practice using apps similar to application under test, so they follow a slightly different path than experts. Therefore, usage profiles of novice users are more likely to reveal bugs that developers did not discover. Their method is as follows: First an expert user produces an initial test sequence. Then, with one or more *DEVIATE* commands genetic algorithms are engaged. Genetic algorithms start with random alleles and genes survive according to their reward which is based on window names, i.e. if the window names of successive events are equivalent, the reward is high.

[10] collects usage profiles of the apps to detect the bugs in new versions of those apps, then generates tests using these profiles. They generate test sequences based on "concatenating pairs of events that have the highest probability of occurring… to determine the effectiveness of highly probable pairs of events"

5. CONCLUSIONS

This paper presented the difficulties of automatic testing of graphical user interfaces from an artificial intelligence perspective. The Main focus was on model-based and model learning techniques. Hence, after presenting the motivations for GUI testing, the paper started by explaining the challenges of generating a model. It continued with an important aspect for crawling the GUI; text inputs. Lastly, various methodologies for test suite generation are discussed. The research shows that although valuable work has been accomplished, improvements can be applied for each title presented. In model exploration, redundant actions should be avoided while actions that bring about unexplored states should be encouraged. Since the resulting state cannot be known in most cases, this task requires some kind of future prediction. When checking for state equivalence, the possibility of overfitting is high due to disparate natures of applications. Text inputs can be generated using a dictionary, but the response of the application is significant, especially in login and register pages. Generation of test sequences can be specialized for specific goals and data mining can be utilized in all of the above cases.

Acknowledgements: This project is supported by Netaş and TÜBİTAK. We would also like to thank Devrim Ergel, Ceyda Ülker, Alper Şen, Yunus Dönmez, Çağdaş Sözer, Tolga Tanrıverdi, Yavuz Köroğlu, Ercan Semerci for valuable discussions and Murat Can Özdemir for his feedback on our paper.

6. REFERENCES

[1] Aravind Machiry, Rohan Tahiliani, and Mayur Naik, "Dyrodroid: an input generation system for Android apps," in Proceedings of the 2013 9th joint Meeting on Foundations of Software Engineering, New York, 2013.

[2] Wontae Choi, George Necula, and Koushik Sen, "Guided GUI Testing of Android Apps with Minimal Restart and Approximate Learning," in ACM SIGPLAN Notices, vol. 48, New York, 2013, pp. 623-640.

[3] Alessandra Gorla, Alessandro Orso Shauvik Roy Choudhary, "Automated Test Input Generation for Android: Are We There Yet?," in 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Lincoln, Nebraska, USA, 2015.

[4] Iulian Neamtiu Tanzirul Azim, "Targeted and depth-first exploration for systematic testing of android apps," in OOPSLA '13 Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications, 2013.

[5] AR Fasolino, P Tramontana D Amalfitano, "MobiGUITAR--A Tool for Automated Model-Based Testing of Mobile Apps," Software, IEEE, vol. 32, no. 5, pp. 53 - 59, April 2014.

[6] Bin Liu,Suman Nath,William G.J. Halfond,Ramesh Govindan Shuai Hao, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in Proceedings of the 12th annual international conference on Mobile systems, applications, and services, 2014.

[7] Mauro Pezzè, Oliviero Riganelli, Mauro Santoro Leonardo Mariani, "Automatic testing of GUI□based applications," Software Testing, Verification and Reliability, vol. 24, no. 5, pp. 341-366, August 2014.

[8] Ali Mesbah Mona Erfani Joorabchi, "Reverse Engineering iOS Mobile Applications," in Reverse Engineering (WCRE), 2012 19th Working Conference on, Kingston, 2012, pp. 177 - 186.

[9] DJ Kasik and HG George, "Toward automatic generation of novice user test scripts," in Proceedings of the SIGCHI conference on Human factors in computing systems, 1996.

[10] Penelope A. Brooks and Atif M. Memon, "Automated GUI testing guided by usage profiles," in Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering, New York, 2007.

[11] C. Damas, and P. Dupont B. Lambeau, "State-merging DFA induction algorithms with mandatory merge constraints," ICGI, pp. 139–153, 2008.

[12] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon, "Using GUI ripping for automated testing of Android applications," in Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering, 2012.

[13] Xun Yuan and Atif M. Memon, "Generating event-sequence-based test cases using GUI runtime state feedback," in Software Engineering IEEE Transactions on, vol. 36, 2010.