A Static-dynamic Approach for UI Model Generation for Mobile Applications

Ibrahim Anka Salihu[1], Rosziati Ibrahim[2], Asmau Usman[3]

[1,2,3]Faculty of Computer Science and Information Technology
Universiti Tun Hussein Onn Malaysia, Parit Raja, Malaysia
[1]isankah3@gmail.com,[2]rosziati@uthm.edu.my; [3]asmee08@yahoo.com

Abstract: **Nowadays, smartphone users are increasingly relying on mobile applications to complete most of their daily tasks. To ensure acceptable quality and to meet its specifications, mobile apps need to be tested thoroughly. As testing mobile apps becomes challenging and tedious, test automation can alleviate this process. Model-based testing is an approach for test automation that is popularly used to test mobile applications. In order to benefit from model-based testing, there is a need for technique and tool for automated model generation. Therefore, this paper presents a hybrid approach for automated User Interface (UI) model generation for mobile applications. It performs static analysis of application's bytecode to extract UI information, followed by a dynamic crawling to systematically explore and reverse engineer a model of the application under test. We then evaluate our approach on several open-source mobile applications. The results showed that our approach can generate a high-quality model from mobile applications.**

Keywords: **Mobile apps, Model-based testing, Test automation, Reverse engineering**

I. INTRODUCTION

In recent years, people largely rely on smartphones due to the applications (apps) they offer. Mobile applications (mobile apps) are used for various computational tasks, such as email access, mobile banking, online business, social networks. Hence, there is increasing concern on applications' quality such as app correctness, performance and reliability [1, 2]. In the context of mobile apps, to improve reliability is not just about making sure that the application is bug-free, but also to survive the competition in the marketplace. This has forced applications' developers to verify apps before releasing to the apps' market. The popular quality assurance technique employed by developers is software testing. Hence, approaches/tools for automated test input generation and execution that can expose buggy behaviour in applications are in high demand [3]. This has no doubts prove the importance of testing tools to improve software reliability.

Unlike the traditional software testing, mobile app testing poses different challenges because they are typically used in more uncontrolled conditions and uses a variety of input data through user interactions [4, 5]. There are various approaches/tools for mobile apps testing utilising the black box (dynamic) or white box (static) approach. The static analysis of app binaries such as in [6], although scalable, it can fail to uncover app's faults. This is due to the run-time issues associated with event-driven apps [7]. For the Android platform certain peculiarities (e.g., an event-based model of app development, depending highly on system events and user input) made the programs developed for this platform complicated for pure black-box (dynamic analysis) testing approaches [8]. On the other hand, the dynamic analysis which examines the runtime behaviour of an app by executing it such as [9], have its own weaknesses. One of such weakness is the inability to explore certain UIs due to the presence of infeasible paths such as modal UI (dialog box) that can sometimes be visible or invisible [7].

Several techniques/tools for automated testing of mobile apps focuses on app UIs to detect bugs. These techniques are broadly classified as script-based, capture/replay, random, systematic exploration and model-based testing [10, 11]. Model-based testing (MBT) is a technique where the test suite is automatically derived from model of the application under test (AUT) [10]. It provides notable improving in test case generation, improving test coverage and achieving better fault detection [9, 12]. However, mobile apps model is often not available due to the use of agile development methodologies by developers or insufficient quality due to lack extensive coverage of an app behaviour.

Despite the growing number of automated model-based testing techniques/tools for mobile apps such as SwiftHand [13], MobiGUITAR [9] and ORBIT [14], this area is still in its infancy and needs further contribution. Addressing this issue, we propose an approach for automatic model generation for Android mobile apps named AMOGA (Automated Model Generator for Android). Our approach utilises static analysis of app's bytecode to extract useful information that can be used to dynamically crawl the model of the app at run-time.

The rest of the paper is organised as follows. Section II presents the proposed approach. Section III shows the evaluation of AMOGA. Section IV discusses the related works. Finally, section V concludes the paper.

II. THE PROPOSED APPROACH

The approach comprises of two components: a static analyzer and dynamic analysis component. The static analyzer performs

978-1-5386-4692-2/18/$31.00 ©2018 IEEE

static analysis of application's bytecode to extract set of events supported by an application and used the events set as input to the dynamic analysis component, whose main goal is to systematically fire events on the running application to explore and reverse engineer a model of the mobile app.

A. Static Analyzer

In this step, we performed control flow analysis (CFA) on the AUT using its APK as input with the help of GATOR [15], a static program analysis toolkit for Android. First, the APK is decompiled to extract the bytecode and use it for the analysis. The analysis targets specific Android components (Activities, menus and dialogs) that determines app's UI behaviour. The callback methods (both event handler and lifecycle callback) are analysed which gives the ability to explore both the UI events and system events. The CFA is abstracted in the form of windows transition graph (WTG). The WTG is a directed graph with nodes depicting app windows and edges representing events. Figure 1 shows the workflow of the static analyser.

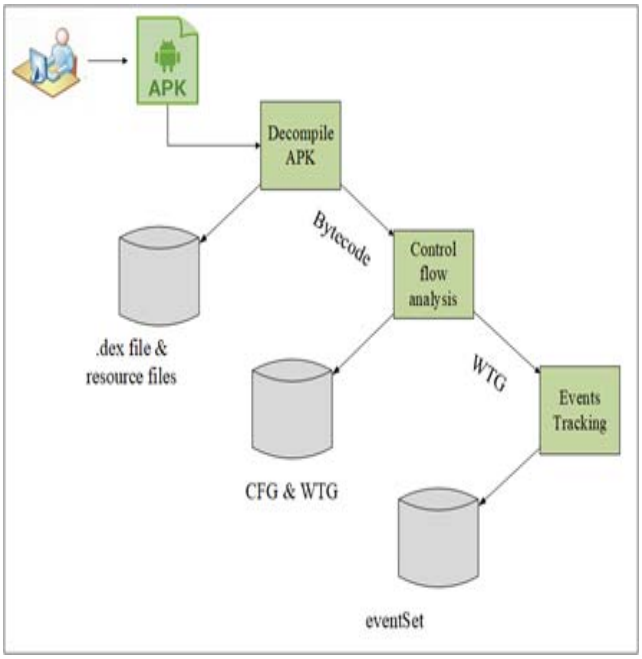

Fig. 1. Workflow of the static analysis.

Based from Figure 1, WTG is for the events tracking. We traversed the graph to generate events set where each edge of the graph is used to represent sequence events that can trigger transition from a particular UI to another UI.

B. Dynamic Analysis Component

The events set generated from the static analysis is used asinput for the dynamic analysis. We implemented a crawler that receives the events set and fire action on the events to explore apps states at run-time. The crawler leverages an open-source tool and Robotium testing framework [16] that have the capability of extracting UIs (e.g., views, check-boxes, buttons) and trigger action on the event handlers. Our crawler exploits these capabilities in Robotium to extract the UI widgets of a running Activity and the event Handlers that implemented the Activity and fire action on the UI. Figure 2 shows the workflow of the dynamic analysis.

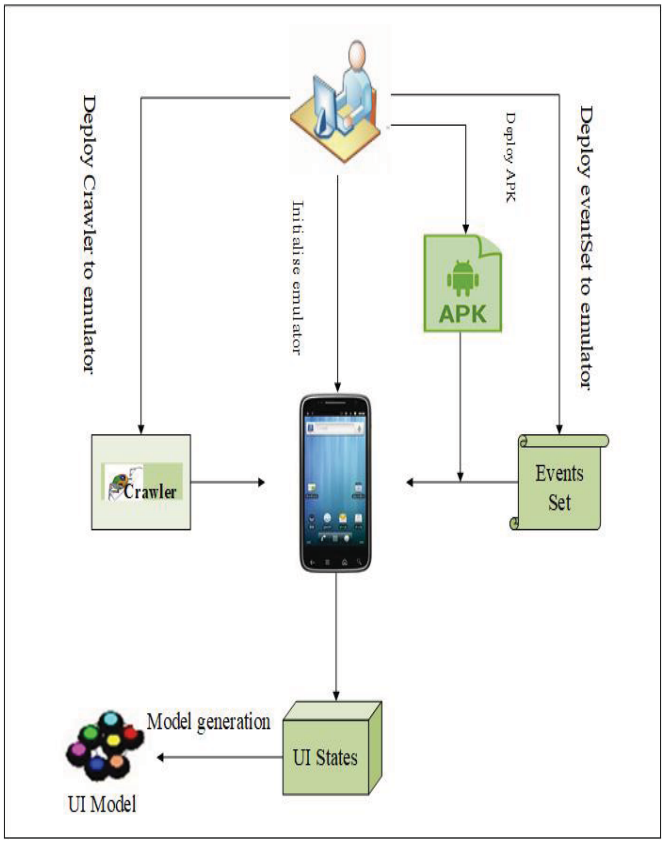

Fig. 2. Workflow of the dynamic analysis.

Based from Figure 2, the crawler builds a model of all discovered states during the exploration. We model the UI states as finite state machine (FSM) where a vertex represents a state of the app and edges represent transition from one state to another.

III. EVALUATION OF AMOGA

We tested our tool named AMOGA by selecting five open-source apps. The apps were selected to demonstrate the application of our approach. We executed AMOGA on the selected apps. Models were construct for all the apps which were further used to generate test cases. The test cases were executed on the apps to measure the code coverage and several bugs were revealed. Figure 3 shows an example of a model derived from Tomdroid app.

EMMA [17] tool was integrated in AMOGA to measure the code coverage. The statement coverage metric is used in our experiments for the evaluation of the code coverage results. The statement coverage is used to calculate and measure the

number of statements in the source code which have been executed.

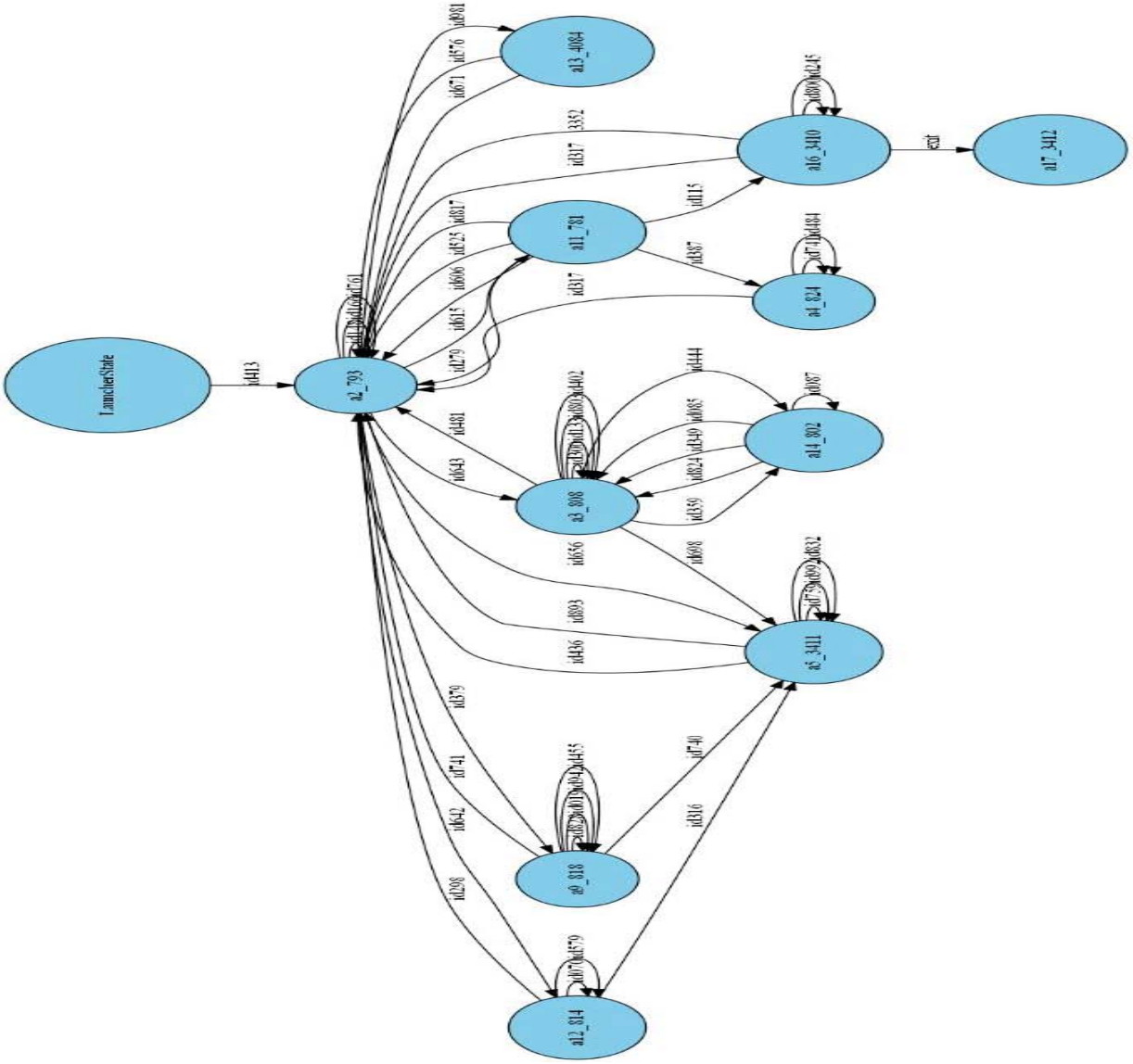

Fig. 3. FSM Model for Tomdroid app.

Table I reports the code coverage obtained for the apps. Column 1 gives apps name. Column 2, 3, and 4 show results of coverage for AMOGA, MCrawlT and MobiGUITAR respectively. The results indicate that AMOGA obtained the highest coverage on all the applications with the highest coverage of 92% on Tomdroid.

Table II reports the bugs detected by AMOGA that led to crash of the app. Column 1 shows names of the apps. Column 2, 3 and 4 present the number of crash detected by AMOGA, MCrawlT and MobiGUITAR respectively.

TABLE I: Code Coverage in (%)

App	AMOGA	MCrawlT	MobiGUITAR
Tomdroid	92	76	77

App	AMOGA	MCrawlT	MobiGUITAR
Aarddict	79	67	65
Notepad	91	88	72
Aagtl	68	25	42
OpenManager	72	65	60

Table II: Crash Detection

App	AMOGA	MCrawlT	MobiGUITAR
Tomdroid	2	1	1
Aarddict	3	2	1
Notepad	2	5	2
Aagtl	2	1	1
Wordpress	24	16	2

Based from Table II, AMOGA and MCrawlT tools detected high number of crash on Wordpress app. Most of the crashes are as results of improper handling of the system events (such as from dialogs). AMOGA detected the highest because it covers system events. Based on the results in Table II, it is concluded that AMOGA outperforms the others in automatic crash detection.

B. Comparison of AMOGA Features with other Tools

We compared the features of AMOGA with MobiGUITAR and MCrawlT. We choose these tools because they are tools for automated model generation with the availability of its implementation. Table III presents the features and the comparison results. Column 1 shows the name of the tools. Column 2 shows the analysis approach used by tool. MobiGUITAR and MCRawlT utilised dynamic analysis while AMOGA is based on static-dynamic analysis. Column 3 gives the types of events supported by tool. MobiGUITAR and MCRawlT both generate UI events only, while AMOGA generates both UI and system events. Column 4 gives the type of artefact (model) produced by tool. AMOGA and MobiGUITAR produce FSM while MCRawlT produces Parameterised Labeled Transition System (PLST).

IV. RELATED WORKS

The popular technique/tool category in the literature are based on static or dynamic analysis apps. Most techniques for automated model generation for mobile apps are based on dynamic approach. Android Automatic Testing Tool (A2T2) [18]dynamically reverse engineers an app model representing the structure and behaviour of its UI. It utilises a crawler that simulates real user events on the UI to generate test cases which are automatically executed on an app for crash testing and regression testing. The tool can only extract a small set of widgets of an Android app.

Table III: Comparison of Features of Tool

App	Approach		Events		Produced artefact
	Dynamic	*Hybrid*	*UI*	*System*	
MobiGUITAR	✓	x	✓	x	FSM
MCrawlT	✓	x	✓	x	PLTS
AMOGA	X	✓	✓	✓	FSM

AndroidRipper [19] utilises a ripper that systematically rips the app's UI to generate test cases for stress-testing that can reveal faults in the code. It does not develop a reusable model of an app. MobiGUITAR [9] is an extension of AndroidRipper which dynamically traverses an app to create a task list that is used to fire events on the UI to generate a FSM model of the app.

Swifthand [13] tool analyses an app at run-time to generate sequences of test inputs for the apps. It implemented a model learning algorithm that is based on machine learning to construct a model of the app during testing. The focus of authors is not the quality of the generated model but rather to improve the test execution. MCrawlT [20] dynamically analyzes an app to create a tasks list corresponding to the UI of the app which can be executed on the app infers a model that captures the supportedevents. The tool only generates UI events but does not consider system events.

ORBIT [14] tool combines static and dynamic analysis to generate a state model from android app. It statically analysethe app source code to generate a set of app's supported actions. A crawler is used to dynamically fire actions on the UI objects. ORBIT does not take into account for the lifecycle events triggered by the life-cycle callback methods. For example, the onCreate method that manages the Activity life-cycle.

A^3T [1] is a static-dynamic analysis approach that automatically explores an apps. A^3T performs a data flow analysis on app's bytecode to construct static activity transition graph (SATG). SATG has nodes depictingactivities with edges showing the possible transitions between the activities (UI screens). It uses the SATG as input for systematic exploration of an app. It is unable to capture the menus/dialog and do not account for the general UI effects of event handlers such as window-close and the triggered callbacks. Hence, it is less

comprehensive as it does not cover some highly-required information.

V. CONCLUSION

This paper presents AMOGA, a tool for automated UI model generation for mobile apps. The results of AMOGA showed that AMOGA generated test cases that can be useful for detecting relevant bugs in an app. Further more, this study showed that the static-dynamic approach and model-based approach for test generation is a promising approach for achieving better crash detection on Android app.

ACKNOWLEDGMENT

The authors would like to thanks Universiti Tun Hussein Onn Malaysia (UTHM) and Research Management Centre (RMC) for funding this research under the Graduate Incentive Research Scheme (GIPS), Vote# U308.

REFERENCES

[1] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, 2013, pp. 641-660.

[2] I. A. Salihu, R. Ibrahim, and A. Mustapha, "A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing," *Journal of Telecommunication, Electronic and Computer Engineering (JTEC),* vol. 9, pp. 45-49, 2017.

[3] J. Voas, S. Quirolgico, C. Michael, and K. Scarfone, "Technical Considerations for Vetting 3rd Party Mobile Applications (Draft)," *NIST, NIST Special Publication,* pp. 800-163, 2016.

[4] A. I. Wasserman, "Software engineering issues for mobile application development," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, 2010, pp. 397-400.

[5] A. Usman, N. Ibrahim, and I. A. Salihu, "Test Case Generation from Android Mobile Applications Focusing on Context Events," in *Proceedings of the 2018 7th International Conference on Software and Computer Applications*, 2018, pp. 25-30.

[6] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in Android applications," in *International Conference on Software Engineering (ICSE)*, 2015.

[7] I. A. Salihu and R. Ibrahim, "Comparative Analysis of GUI Reverse Engineering Techniques," in *Advanced Computer and Communication Engineering Technology*, ed: Springer, 2016, pp. 295-305.

[8] Y. Zhauniarovich, A. Philippov, O. Gadyatskaya, B. Crispo, and F. Massacci, "Towards black box testing of android apps," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, 2015, pp. 501-510.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Software,* vol. 32, pp. 53-59, 2015.

[10] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Information and Software Technology,* vol. 55, pp. 1679-1694, 2013.

[11] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?(E)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429-440.

[12] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*: Morgan Kaufmann, 2010.

[13] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *ACM SIGPLAN Notices*, 2013, pp. 623-640.

[14] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Fundamental Approaches to Software Engineering*, ed: Springer, 2013, pp. 250-265.

[15] "GATOR: Program Analysis Toolkit For Android," *http://web.cse.ohio-state.edu/presto/software/gator/*.

[16] U. Apache, "Robotium," *http://code.google.com/p/robotium*.

[17] Emma, "An open source Java code coverage tool," *http://emma.sourceforge.net/*.

[18] D. Amalfitano, A. R. Fasolino, and P. Tramontana, "A gui crawling-based technique for android mobile application testing," in *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, 2011, pp. 252-261.

[19] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," presented at the Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 2012.

[20] S. Salva, P. Laurençot, and S. R. Zafimiharisoa, "Model Inference of Mobile Applications with Dynamic State Abstraction," in *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing 2015*, ed: Springer, 2016, pp. 177-193.