A Hybrid Approach for Reverse Engineering GUI Model from Android Apps for Automated Testing

Ibrahim Anka Salihu, Rosziati Ibrahim and Aida Mustapha
Universiti Tun Hussein Onn Malaysia, 86400 Parit Raja, Batu Pahat, Johor, Malaysia.
hi130015@siswa.uthm.edu.my

***Abstract*—Nowadays, smartphone users are increasingly relying on mobile applications to complete most of their daily tasks. As such, mobile applications are becoming more and more complex. Therefore, software testers can no longer rely on manual testing methods to test mobile applications. Automated model-based testing techniques are recently used to test mobile applications. However, the models generated by existing techniques are of insufficient quality. This paper proposed a hybrid technique for reverse engineering graphical user interface (GUI) model from mobile applications. It performs static analysis of application's bytecode to extract GUI information followed by a dynamic crawling to systematically explore and reverse engineer a model of the application under test. A case study was performed on real-world mobile apps to evaluate the effectiveness of the technique. The results showed that the proposed technique can generate a model with high coverage of mobile apps behaviour.**

***Index Terms*—Graphical User Interface Testing; Mobile Application; Reverse Engineering; Test Automation.**

I. Introduction

Over the years, mobile devices are rapidly replacing traditional computers for an increasing number of users for various computational tasks, such as access to emails, mobile banking, e-services, social networks, etc. The popularity of these devices has impacted the area of software development with a huge increase in the development of mobile applications in recent years [1] to meet the respective needs of their users.

Mobile applications (Mobile Apps) are software systems designed for mobile devices (smart phones, tablets and other handheld devices). They belong to the class of event-driven graphical user interface (GUI) applications where the GUIs serves as the main user interface for interacting with the application. Though mobile apps are initially simpler and smaller with less complex design architecture and having a small set of functionalities, they are recently increasing in capacity, functionality, structure and behaviour [2]. They are now becoming more and more complex [3]. The reliance on mobile apps in everyday life has brought concerns about the quality of mobile apps such as correctness, performance and security [4-7]. However, the increased complexity of mobile apps has brought several challenges for the software engineering community in understanding mobile apps' behaviour and testing them [8-10].

GUI testing is typically an important activity that is aimed at detecting faults that lead to failures in the GUI or an application in general [11], and it plays a significant role in improving the quality of software systems [12]. Specifically, GUI test automation is essential in testing today's mobile apps because the GUIs are increasingly becoming more complex. Several techniques are used to automate GUI testing for mobile apps such as script-based, capture/replay, random-walk, systematic exploration and automated model-based [13-15]. Model-based testing (MBT) is one of the popular ways of automating the testing process for mobile apps [16].

MBT is becoming increasingly popular among the software engineering community [17-19] as an approach for testing mobile apps. In MBT, the test cases are automatically derived from the model of an application under test (AUT) [13, 15, 20]. It can enhance the creation of test scripts and test coverage of an application, and reduce the time and cost for testing [13, 19, 21, 22]. However, such model is not always available or of insufficient quality. The model can be constructed manually or using automated techniques. Constructing the model manually is tedious and error-prone. On the other hand, building the model fully automatically for mobile apps is challenging due to the dynamic behaviour of their GUIs [11, 23, 24]. One way to relieve the effort in constructing GUI model is to automatically reverse engineer the model from a given mobile app. The latter is the focus of this study.

This paper described a technique that reverse engineers mobile apps to automatically generate a high quality model representing the GUI behaviour. The proposed technique is based on a hybrid approach that performs static analysis of mobile app source code followed by dynamic analysis at runtime. The contributions of this paper are summarised as follows.

i. A hybrid technique for reverse engineering of mobile apps is proposed and implemented in a prototype tool.
ii. We proposed a crawler for the dynamic model exploration of GUI from mobile apps.
iii. An evaluation was performed on real world mobile app based on code coverage analysis.

The rest of the paper is organised as follows. Section II discusses the state-of-the-art in GUI reverse engineering. Section III presents the related work. Section IV presents the proposed technique. A case study on real-world mobile apps is shown in section V. The conclusion is presented in section VI.

II. State of the Art

Reverse engineering (RE) is gaining more popularity from the research community as an act of analysing a software system, either in whole or in part, to extract design and implementation information [25] that can be useful for several tasks such as software comprehension, documentation, maintenance, and re-engineering [26]. Nowadays, RE is used for various purposes other than software comprehension such as software testing, software reusability, updating user interfaces, migration and porting user interfaces to new platforms [27, 28]. There are two approaches to reverse engineering GUI applications; the static approach and dynamics approach. The static approach performs an analysis on the application's source code or binary code without executing the application [11, 29]. On the other hand, dynamic approach extracts information from an application by executing and analysing its external behaviour [21, 30].

Both static and dynamic approaches for reverse engineering GUI applications have their strengths and weaknesses. The static approach is capable of retrieving more accurate and complete information from an application but the dynamic object-oriented nature of GUI applications can sometimes complicate the analysis, which makes it very difficult or even impossible to retrieve comprehensive information about the behaviour of GUIs by just analysing their source code [2, 12]. This is because access to some components depends on other components and some components are only reachable from a particular state. In addition, information about overlapping windows is not accessible through static analysis. On the other hand, the dynamic approach to reverse engineering GUI applications is easier to implement. One of the most challenging issue in dynamic reverse engineering is how events are found and fired in controlling the model exploration [9, 21, 31] and the inability to explore certain GUI due to the presence of infeasible paths that requires user inputs such as user credentials [21]. Hence, the extracted information about the behaviour of the application could be inaccurate and incomplete, which affects the quality of model generated [21, 31, 32].

Recently, the hybrid approach has been the focus of researchers in the area of GUI reverse engineering particularly for the Android apps [31, 33]. The hybrid approach can provide enhancement in terms of the scope, completeness and precision of the process as it exploits the capabilities of both static and dynamic approaches while trying to maximise the quality of the extracted information [34]. Several researchers believed that using static analysis to generate meaningful input for the dynamic exploration can ensure the generation of a high quality model [31, 35]. However, the static analysis in existing hybrid approaches such as Orbit [9] is not comprehensive, which affects the quality of the model generated.

III. Related Works

Several model reverse engineering techniques and tools were proposed for automated testing of Android apps over the last decade. Most of these techniques are pure black-box techniques that are based on dynamic analysis of mobile apps, with few that are based on the hybrid approach. One of the earliest techniques is GUI ripping [39] that was implemented as part of GUITAR tool [40] for testing desktop applications. The technique reverse engineers a model of an application by automatically executing and exercising the applications' GUI. An extension of the tool has been proposed for the Android platform known as Android GUITAR [36]. The technique is not able to capture the rich set of user inputs associated with a mobile app (such as swiping, pinching etc). It also produces many false event sequences which may need to be weeded out later.

A2T2 (Android Automatic Testing Tool) [7] is based on a crawler that simulates real user events on the user interface to generate test cases that can be automatically executed on an app for crash testing and regression testing. It constructs GUI tree model which can be used for driving the test cases. The technique manages only a small subset of widgets and does not have support for the rich set of user inputs associated with an Android app. AndroidRipper tool [37] is based on a ripper that systematically analyses and rips mobile app's GUI to obtain event sequences that can be fired on the GUI, with each sequence representing an executable test. It automatically generates GUI tree model. The exploration is not comprehensive and it can sometimes lead to unexpected faults. ICRAWLER tool [41] systematically reverse engineers a state model from iOS applications. The technique does not capture some UI elements such as the toolbar, slide bar, search bar, and advanced gestures such as swiping pages and pinching. Swifthand [42] dynamically crawl a given mobile app and systematically reverse engineer state machine model of a mobile app. The technique only generates default touching and scrolling events of the GUI but does not consider system events.

ORBIT tool [9] is based on hybrid static/dynamic analysis of an application. It performs static analysis on the source code of an app to generate set of user actions supported by an app. A crawler is used to dynamically fire actions on the GUI objects of a running app to extract a state model of the application. However, the static analysis in the ORBIT is not comprehensive as the one proposed in this paper. Our technique performs static analysis on the bytecode of an app (considering fact that source code of mobile app is rarely or not available) to extract all supported events and used them for the dynamic exploration. Tao and Gao [36] proposed a test automation system that models GUI dependencies between various scenarios in mobile apps. Their aim is to avoid test case failures coming from test cases having scenario dependencies between GUI components.

IV. Proposed Approach

This section discussed the hybrid technique for the reverse engineering of mobile apps. The technique consists of a static analyser that extracts set of events supported by a mobile app and a dynamic crawler that is responsible for dynamically exercising the events to record their states. It performs 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 crawler, whose main goal is to systematically fire events on the running application to explore and reverse engineer a model of the mobile app. The proposed technique was implemented in a prototype tool called AMOGA (Automated Model Generator for Android Apps). Figure 1 shows the framework of the proposed approach.

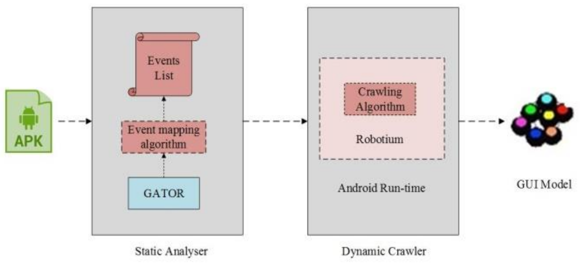

Figure 1: Framework of the proposed approach

A. *Static Analyser*

The static analyser is implemented on top of GATOR [37], a static analysis tool for Android. To analyse a mobile app, the technique receives as input, the apk file of an application and decompile it to bytecode. The analyser starts the analysis on the application’s bytecode to construct windows transition graph (WTG) of an application which can be used to create the events list that can be used as input to the dynamic crawler. Figure 2 shows workflow of the static analyser.

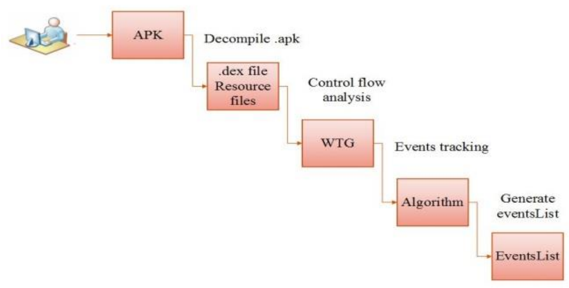

Figure 2: Workflow of the static analyser

B. *Dynamic Crawler*

The dynamic crawler is responsible for executing the mobile app to trigger the events and explore app’s states. The crawler is built on top of Robotium testing framework [38] which has the capability to extract GUIs (such as test views, check boxes, buttons, spinners etc.) and fire action on the event handlers. It also has the functionality for editing and clearing text boxes, clicking on home, menu and back button. The proposed crawler exploits the capability in Robotium to extracts the GUI widgets of the running Activity and the event Handlers that implemented the Activity.

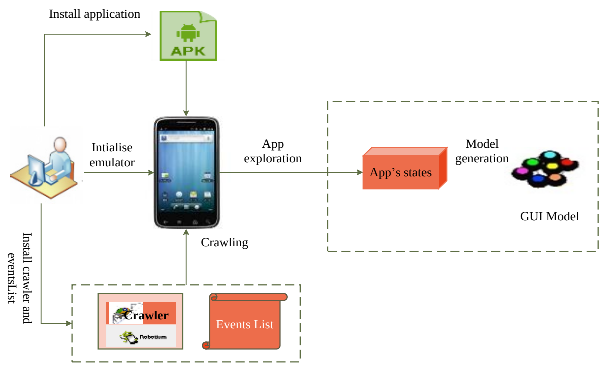

Figure 3: Workflow of the dynamic crawling

The crawling algorithm systematically extracts the GUI widgets associated with each event and fire action on the GUI to trigger state transition. It produces a data structure describing the obtained GUI state model, based on the description of the identified Interfaces and of the triggered events.

Figure 4 provides a detailed description of the application crawling. First, it starts the app from the launcher node, and then the algorithm starts with the events list generated from the static analysis. It takes an event from the events set and fire the event to trigger a transition to the next state (Line 2-5). When a new state is discovered it will be added to the model with its trigger conditions (Lines 7-9). The algorithm continues to the next open state iteratively (Lines 11). The set of states and transitions is updated accordingly and added to the model (Line 12) to reflect the changes.

```
Algorithm 1. App crawling
Input: A: app under test, eS: event set,
Output: M: generated model
 1 Initialisation M←∅;e←getEventsSet(A)
 2 Start the app
 3 foreach event e ∈ eS do
 4    s←getCurrentWindow(A)//get state of event
 5      while s ≠ null do
 6         newS ← is new state(eS,A)
 7         if s is newS
 8         APPLICATIONCRAWLING(A,eS)
 9          addToModel(newS,M)//add newState to
                model
10          end if
11          get next e to explore
12    updateModel(newS, e, m)
13      end
14    end
15 end
```

Figure 4: The application crawling algorithm

C. *Model Definition*

Finite state machines (FSM) are widely used to model the behaviour of event-driven, interactive system, in particular, GUI applications [9, 39]. The proposed technique model the GUI behaviour using a finite state machine. A FSM is composed of states, actions and transitions, and can be represented using a diagram [40]. The FSM maps events and related conditions to a list of GUI actions references. The FSM is defined as follows.

Definition: Finite state machine for a mobile app MA is defined as a graph <V, A, T>
where:

- V is a finite set of nodes representing all possible states. Each v ∈ V represents a state in MA.
- A represents user events on a MA
- a ∈ A ⊆ V × V is a set of directed arcs between vertices. Event e2 follows e1 if e2 can be executed immediately after e1.
- w ∈ V is a set of vertices representing those windows of MA that are ready for user interaction when the mobile app is launched.
- T defines all transitions from a state to another through.

V. Experimental Evaluation

Several experiments were conducted on selected mobile apps to evaluate the performance of the model reverse engineering technique. A total of eight (8) mobile apps were selected that were used by previous techniques on automated model generation [9, 24, 41]. In other to avoid being bias, the selection covered a range of popular real-world open source mobile apps benchmarks that were used in the evaluation of the tools selected for the comparison and fall across different categories of apps such as productivity, business, etc. Test cases were derived from the generated model from each of the selected mobile apps and were used to test the application.

Table 1 presented the characteristics of selected apps. The lines of code in an app is shown in column 2, the Activities in column 3 and the last column shows the number of downloads based on Google Play analytics as of January 2017. The experiments intend to answer the following question. Does the crawler offer good coverage in a reasonable time?

Table 1
Characteristics of mobile apps used in evaluation

Apps	#LOCs	Activities	Category	Download
TippyTipper	2248	6	Tool	100K–500K
ToDoManager	5623	2	Productivity	100K–500K
ContactManager	802	2	Productivity	1K–5K
Tomdroid	5038	5	Business	10K–50K
AardDict	5097	4	Books & Ref.	10K–50K
OpenManager	3647	6	Business	5M–10M
Notepad	8172	8	Productivity	500K–1M
Aagtl	43105	3	Tool	500K–1M

A. Results

To answer the research question, experiments were conducted on all the selected mobile apps and the code coverage and time taken to crawl each application were recorded. The proposed technique and AndroidRipper [24, 42], were also used to run an experiment on applications used by the existing techniques in their published articles. Seven (7) apps were selected from the apps used to evaluate Orbit [9], and 6 from MCrawlT [43]. However, due to the unavailability of Orbit [9] tool, and difficulty in the setup of other tools, MCrawlT [43], and Android GUITAR [41] the results published in their articles was used. The effectiveness of our technique was measured and compared with the selected tools.

Table 2 presents the percentage code coverage obtained with each tool on the eight (8) applications. The coverage result showed that our technique achieved 45%-93% coverage across the 8 applications.

Table 3 presented the execution time recorded for each of the selected tools on all the applications. The results showed that AMOGA explored all the applications within 102 seconds – 370 seconds maximum. This indicated that AMOGA took an average time of 102 seconds to explore all the applications. The time along with the coverage obtained showed that AMOGA tool offers good coverage against all the tools on the selected mobile apps.

Table 2
Comparison of code coverage

Apps	Android GUITAR	Android Ripper	ORBIT	MCrawlT	AMOGA
TippyTipper	47	-	78	79	83
ToDoManager	71	-	75	81	84
ContactManager	61	-	91	68	93
Tomdroid	-	40	70	76	83
AardDict	-	27	65	67	71
OpenManager	-	-	63	65	72
Notepad	-	-	82	88	91
Aagtl	-	-	-	25	45

Table 3
Comparison of exploration time

Apps	Android GUITAR	Android Ripper	ORBIT	MCrawlT	AMOGA
TippyTipper	322	-	198	110	102
ToDoManager	194	-	121	210	116
ContactManager	247	-	125	135	114
Tomdroid	-	529	340	196	180
AardDict	-	694	124	580	120
OpenManager	-	-	480	489	370
Notepad	-	-	102	175	110
Aagtl	-	-	-	920	220

VI. Conclusion

This paper has presented a hybrid approach for reverse engineering a model of mobile apps. We described our prototype tool called AMOGA that implements the hybrid approach which consists of static analyser that generates a list of application's supported events and dynamic crawling to explore the state of events in an application. AMOGA can generate a model that represents the behaviour of a mobile app. The model can be used to create test cases for testing an application.

We applied AMOGA to 8 mobile applications and reverse engineer a FSM of the GUI. We used the model to generate test cases that we used to test the applications. The experimental results showed that AMOGA can generate a model with high coverage for testing mobile apps.

Acknowledgment

This project is sponsored by Universiti Tun Hussein Onn Malaysia via the Contract Grant Vot U436.

References

[1] F. Nayebi, J.-M. Desharnais, and A. Abran, "The state of the art of mobile application usability evaluation," in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2012, pp. 1-4.

[2] R. Islam, R. Islam, and T. Mazumder, "Mobile application and its global impact," *International Journal of Engineering & Technology IJET-IJENS*, vol. 10, no. 6, pp. 72-78, 2010.

[3] R. Minelli and M. Lanza, "Software Analytics for Mobile Applications-Insights & Lessons Learned," in *2013 17th European Conference on Software Maintenance and Reengineering*, 2013, pp. 144-153.

[4] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, 2011, pp. 77-83.

[5] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *17th European Conference on Software Maintenance and Reengineering (CSMR)*, 2013, pp. 133-143.

[6] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones," in *OSDI'10 Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, 2010, pp. 393-407.

[7] A. Rountev and D. Yan, "Static reference analysis for GUI objects in android software," in *CGO'14 Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Orlando, FL, USA, 2014, pp. 143.

[8] 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.

[9] 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, V. Cortellessa, and D. Varró, Eds. Berlin Heidelberg: Springer, 2013, pp. 250-265.

[10] S. Yang, D. Yan, H. Wu, Y. Wang, and A. Rountev, "Static control-flow analysis of user-driven callbacks in android applications," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, 2015, pp. 89-99.

[11] A. P. Grilo, A. R. Paiva, and J. P. Faria, "Reverse engineering of GUI models for testing," in *2010 5th Iberian Conference on Information Systems and Technologies (CISTI*), 2010, pp. 1-6.

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

[13] B. Nguyen, B. Robbins, I. Banerjee, and A. Memon, "GUITAR: An innovative tool for automated testing of GUI-driven software," *Automated Software Engineering*, vol. 21, no. 1, pp. 65-105, 2014.

[14] A. Méndez-Porras, C. Quesada-López, and M. Jenkins, "Automated testing of mobile applications: A systematic map and review," in *XVIII Ibero-American Conference on Software Engineering*, Lima-Peru, 2015, pp. 195-208.

[15] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *ASE '15 Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 429-440.

[16] G. de Cleva Farto and A. T. Endo, "Evaluating the model-based testing approach in the context of mobile applications," *Electronic Notes in Theoretical Computer Science*, vol. 314, pp. 3-21, 2015.

[17] M. Young, *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, 2008.

[18] I. Banerjee, B. Nguyen, V. Garousi, and A. Memon, "Graphical user interface (GUI) testing: Systematic mapping and repository," *Information and Software Technology*, vol. 55, no. 10, pp. 1679-1694, 2013.

[19] L. Lu and Y. Huang, "Automated GUI test case generation," in *2012 International Conference on Computer Science & Service System (CSSS)* , 2012, pp. 582-585.

[20] P. Aho, M. Suarez, A. Memon, and T. Kanstrén, "Making GUI testing practical: Bridging the gaps," in *2015 12th International Conference on Information Technology - New Generations (ITNG)*, 2015, pp. 439-444.

[21] A. Kull, "Automatic GUI model generation: State of the art," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2012, pp. 207-212.

[22] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2010.

[23] A. M. Memon, "An event-flow model of GUI-based applications for testing," *Software Testing Verification and Reliability*, vol. 17, no. 3, pp. 137-158, 2007.

[24] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of android applications," in *012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, Essen, Germany, 2012, pp. 258-261.

[25] T. Cipresso and M. Stamp, "Software reverse engineering," in *Handbook of Information and Communication Security*, P. Stavroulakis, and M. Stamp, Eds. Springer, 2010, pp. 659-696.

[26] E. Stroulia, M. El-Ramly, P. Iglinski, and P. Sorenson, "User interface reverse engineering in support of interface migration to the web," *Automated Software Engineering*, vol. 10, no. 3, pp. 271-301, 2003.

[27] G. Canfora, M. D. Penta, and L. Cerulo, "Achievements and challenges in software reverse engineering," *Communications of the ACM*, vol. 54, no. 4, pp. 142-151, 2011.

[28] J. Krijnen, "Software reverse engineering," 2013. Available at https://pdfs.semanticscholar.org/41f7/8442ab7032ec90f2890c99a2ac8435bad924.pdf

[29] J. C. Campos, J. Saraiva, C. Silva, and J. C. Silva, "GUIsurfer: A reverse engineering framework for user interface software," Reverse Engineering-Recent Advances and Applications, pp. 31-54, 2012.

[30] M. M. Moore, "Rule-based detection for reverse engineering user interfaces," in *Proceedings of the Third Working Conference on Reverse Engineering 1996*, 1996, pp. 42-48.

[31] C. E. Silva and J. C. Campos, "Combining static and dynamic analysis for the reverse engineering of web applications," in *EICS '13 Proceedings of the 5th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, London, United Kingdom, 2013, pp. 107-112.

[32] P. Aho, T. Raty, and N. Menz, "Dynamic reverse engineering of GUI models for testing," in *2013 International Conference on Control, Decision and Information Technologies (CoDIT)*, 2013, pp. 441-447.

[33] P. Aho, M. Suarez, T. Kanstren, and A. M. Memon, "Murphy tools: Utilizing extracted gui models for industrial software testing," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops (ICSTW)* , 2014, pp. 343-348.

[34] I. C. Morgado, A. C. Paiva, and J. P. Faria, "Dynamic reverse engineering of graphical user interfaces," *International Journal On Advances in Software*, vol. 5, no. 3 and 4, pp. 224-236, 2012.

[35] S. Yang, D. Yan, and A. Rountev, "Testing for poor responsiveness in android applications," in *2013 1st International Workshop on the Engineering of Mobile-Enabled Systems (MOBS)* , 2013, pp. 1-6.

[36] C. Tao and J. Gao, "Building a model-based GUI test automation system for mobile applications," *International Journal of Software Engineering and Knowledge Engineering*, vol. 26, no. 09n10, pp. 1605-1615, 2016.

[37] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev, "GATOR: Program analysis toolkit for android," 2016, Available at http://web.cse.ohio-state.edu/presto/software/gator/.

[38] U. Apache, "Robotium," Available at http://code.google.com/p/robotium.

[39] M. E. Joorabchi and A. Mesbah, "Reverse engineering iOS mobile applications," in *2012 19th Working Conference on Reverse Engineering (WCRE)*, 2012, pp. 177-186.

[40] R. J. Jacob, "Using formal specifications in the design of a human-computer interface," *Communications of the ACM*, vol. 26, no. 4, pp. 259-264, 1983.

[41] "Android GUITAR," Available at https://sourceforge.net/projects/guitar/.

[42] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and G. Imparato, "A toolset for GUI testing of android applications," in *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, 2012, pp. 650-653.

[43] 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*, 2016, pp. 177-193.