MobiGUITAR – A Tool for Automated Model-Based Testing of Mobile Apps

Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana
Department of Electrical Engineering and Information Technologies
University Federico II of Naples, Via Claudio 21, Naples, Italy
E-mails: {domenico.amalfitano, anna.fasolino, porfirio.tramontana}@unina.it

Bryan Dzung Ta, Atif M. Memon
Department of Computer Science
Univ. of Maryland, College Park, MD, USA
E-mails: {bryanta, atif}@cs.umd.edu

***Abstract*—As mobile devices become increasingly smarter and more powerful, so too must the engineering of their software. User-interface driven system testing of these devices is gaining popularity, with each vendor releasing some automation tool(s). We feel that these tools are inappropriate for amateur programmers, an increasing fraction of the app developer population. We present *MobiGUITAR* for automated GUI-driven testing of Android apps. *MobiGUITAR* is based on observation, extraction, and abstraction of the run-time state of GUI widgets. The abstraction is a scalable state-machine model that, together with test coverage criteria, provides a way to automatically generate test cases. We apply *MobiGUITAR* to 4 open-source Android apps, automatically generate and execute 7,711 test cases, and report 10 new bugs. A number of bugs are "Android-specific," stemming from the event- and activity-driven nature of Android.**

Keywords: Software testing, GUI testing, Android testing.

I. Introduction & Contributions

Given all the recent attention to mobile platforms, there is little doubt that mobile will be the dominant personal computing and internetworking platform for the foreseeable future. It is widely believed that mobile platforms are largely adopted because of the apps they offer [1], [2]. The issue of app quality has already started to become important [3].

We focus on the popular Android system and on software testing, one of the most frequently used QA techniques, even in the mobile app context. A relevant family of techniques and tools for automated testing of mobile apps focus on their GUI to find bugs. They were recently classified into Random testing, Model-based testing, and Model-learning testing techniques [4]. Random testing generates random sequences of UI events to the app under test. It has the advantage of being a black-box technique that does not require any knowledge of the app under test and can be easily implemented. As an example, Monkey [5] and Dynodroid [6] are two tools available for random testing of mobile apps. Model-based techniques require a model of the app under test to generate inputs. TEMA is a suite of tools proposed for model-based testing of Android apps [7]. Model-learning testing techniques test the application while they infer a model of the app under test [4], [8]. To cope with the well-known model explosion problem, they implement strategies to bound the input set, which in turn may limit the coverage and fault detection capability of the testing technique.

In past work, we already showed the feasibility of using a model-learning technique for testing Android mobile apps [8]. In our previous work on conventional desktop applications [9], we developed an automated model-based testing technique based on reverse engineering that generates test cases to execute directly on the software's GUI.

We now develop a test generation technique based on a reverse-engineered mobile app model. Our original model—event-flow graph (EFG) [10]—was "stateless". This was a deliberate design decision to avoid the state-space explosion problem. Having a stateless model is no longer an option because mobile apps are extremely state sensitive, i.e., consider the state-based life-cycle of the Android "Activity" classes, which form the basis for Android app's GUI. Moreover, we can no longer use the original test adequacy criteria based on the EFG. We need new ones that are state sensitive. Third, our test-case generators, also based on EFGs, are unusable; we need new ones that operate on state machines. Finally, in prior work, we never needed to handle security when testing desktop applications; we simply executed the test harness and application as the same user. Because most mobile platforms have enhanced security (e.g., each Android app by default executes in its own sandbox), we need novel techniques for our reverse engineering harness.

To overcome these challenges, our current work makes several new intellectual contributions. We now model the state of the app's GUI; this helps us to more accurately model the state-sensitive behavior of mobile apps. This approach is coherent with other similar ones already proposed in the context of desktop Java apps [11] and Web applications [12], [13]. We define new test adequacy criteria that are based on state machines. We have a new test generation technique that uses the models and criteria to generate test cases automatically. Finally, we provide fully automatic testing by working with the security policies of mobile platforms.

We realize all our contributions as a conceptual framework called *MobiGUITAR* for automated mobile app testing. We have implemented *MobiGUITAR* in a tool chain that executes on Android. Our first tool in the chain is an enhanced version of *AndroidRipper* [8] that automatically reverse engineers the state-machine model from the executing app. Our other tools create abstractions of the machine, generate test cases, and replay the test cases.

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE

II. Overview of *MobiGUITAR*

We now present an overview of *MobiGUITAR*, focusing on the process it employs to perform fully automatic testing of AardDict, an app from our evaluation study subjects (Section III). *MobiGUITAR* is based on three primary steps: (1) *Ripping* that dynamically traverses an app's GUI and creates its state-machine model, (2) *Generation* that uses the model and test adequacy criteria to obtain tests that are sequences of events, and (3) *Execution* that replays the tests.

Ripping: This step obtains a state-machine model of the app's GUI to be used for test generation. Because one of our original goals is full automation, we create the model via an automated reverse engineering technique called *GUI Ripping* that has roots in our previous reported work on GUI testing [9]. The difference between our previous and current work is that we now obtain a state-machine model of the GUI instead of an event-flow graph, and that we use algorithms better suited for mobile platforms; our previous work was for desktop applications.

Our ripper starts by launching the app in a given *start state*, obtaining a list of events that can be performed on the GUI in this state, and adding this list, each as a separate *task*, to a "task list," which it uses to fire events. An element in the task list is removed and fired. New states are encountered and the GUI's focus changes as the events are fired. Whenever the current state changes, the list of new fireable events is obtained and appended to the task list in such a way that the path from the start state is prepended to each event. Hence, formally, a task is a *sequence of events* that always begins with an event fireable in the *start state*.

This process of using a task list essentially realizes a "breadth-first" traversal of the app's GUI. During this process, a tree of GUI states may be maintained. In practice, the number of encountered states may be extremely large, making the GUI tree and traversal inefficient. For the purpose of test generation, several GUI states may be viewed as "equivalent" using a given criterion and merged. In practice, this turns the GUI tree into a directed graph model as the same state may be encountered in multiple ways by the Ripper. The Ripper determines the equivalence between encountered GUI states on the basis of the properties of their constitutive objects. Equivalent states comprise of equivalent objects having the same values of their ids and type properties.

Application of the ripper algorithm to AardDict yields the state machine of Figure 1. The state and event IDs have been compacted because of space. The diagram of the state machine demonstrates several points. First, we see that the user interaction space is quite flexible in that there are many paths and loops in this machine. For instance, the user can perform many event sequences switching between states a5, a45, a58, a46, and a61; the user can then go back to the start state and navigate to a3, a17, a53, and a2; and still go back to the start state; and so on. This flexibility creates a ripe situation for failures that result from specific event sequences. Second, we show a special shape, namely exit; this is not a GUI state;

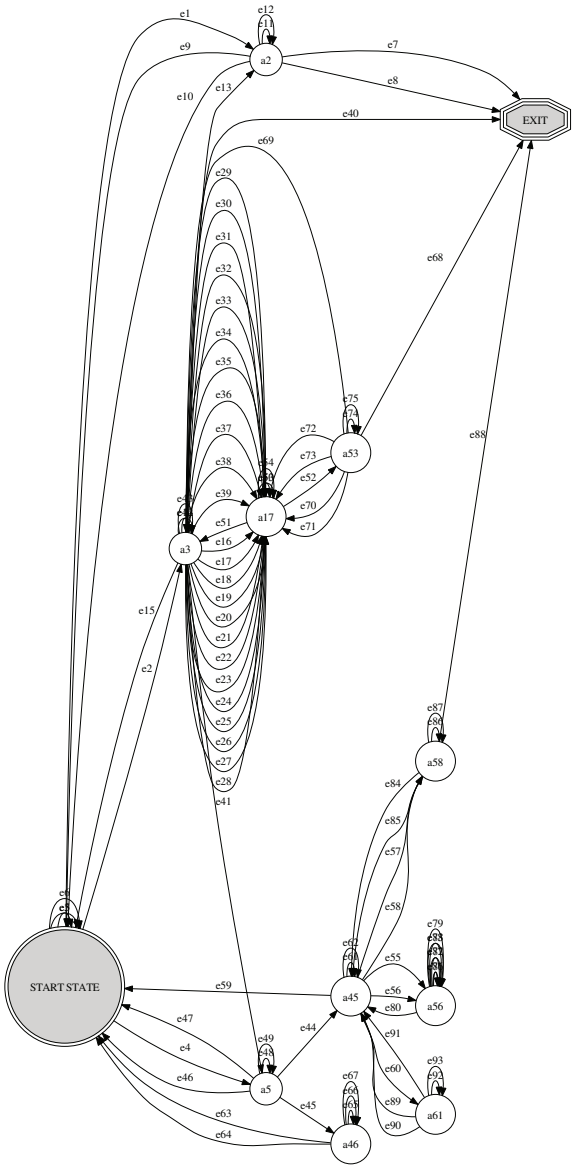

Fig. 1. The abstract state machine for AardDict.

rather, it is meant to show that the app was terminated, via events e7, e8, e40, e68, or e88.

Generation: This step obtains test cases, each modeled as a sequence of GUI events. As can be imagined, the number of all possible event sequences that may be executed on any non-trivial app's GUI is extremely large (in principle infinite because of loops). The test generation strategy needs to sample from this space. Our state machine allows just that. We develop a pair-wise edge coverage criterion. Conceptually, this means that *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. For example, a2 has 4 incoming edges (e1, e11, e12, e13) and 6 outgoing edges

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE

(e7, e8, e9, e10, e11, e12); this creates 4×6=24 pairs that need to be covered. The test case ⟨e1, e11, e8⟩ covers 2 of these pairs, namely (e1, e11) and (e11, e8). In the case of the Aardict app, we generated 678 test cases, with a total of 2,747 fireable events.

Execution: Our current implementation of the test generator outputs test cases in JUnit format. Such test cases are able to detect crashes in the app during its execution. A tester may enhance a test case by adding JUnit-like *assert* statements to check for functional errors.

For our Aardict app, 674 of the 678 test cases executed without any problems and covered 70% of the code. Of the 678 test cases, 4 detected a bug that led to an unhandled *IllegalArgumentException*.

III. Demonstration of *MobiGUITAR*

We now select 4 study subjects from Google Play (https://play.google.com/store/apps): Aard Dictionary 1.4.1 (https://github.com/aarddict/android) - a dictionary and offline Wikipedia reader, Tomdroid Notes 0.5.0 (https://launchpad.net/tomdroid) - a note-taking app, BookCatalogue 3.8.1 (https://github.com/eleybourn/Book-Catalogue/wiki) - a book cataloging app, and Wordpress Revision #394 of the alpha version for Android (http://android.svn.wordpress.org) - an interactive client for creating, updating, and managing blogs saved on a Wordpress server. All the subjects are open source projects developed and maintained by active communities of programmers. For brevity, we call these applications AardDict, Tomdroid, BookCat, and Wordpress.

We want to show the bug detection capability of our tool-chain. To this aim we automatically executed *MobiGUITAR* on all subject applications; in all, we generated and executed 7,711 test cases. Several of these test cases revealed bugs; in all, we detected 10 bugs. The test cases obtained from the model were able to find all the seven bugs already detected during the Ripping step, besides three new ones.

Table I provides more details about these bugs. For "Bug Class," we use a part of an existing Android bug classification [14] that distinguishes between Concurrency (C) (interaction of multiple processes or threads), Activity (A) (incorrect management of the Activity lifecycle), and Other (O) (incorrect application logic implementation) bugs. We show the exception that was thrown and the ticket number that we opened to report the bug.

Some bugs showed us that Android applications may have incorrect behaviors due to a wrong management of the life-cycle of their Activities. Activity components are responsible for presenting a visual user interface for each focused task the user can undertake. In its lifecycle, an Activity instance passes through three main states, namely running, paused and stopped. When it transits into and out of the different states described above, it is notified through various callback methods (like onPause, onResume, onStop,. . . , onDestroy) that are hooks the programmer can override to do appropriate work when the state of the Activity changes. If the programmer fails to override or incorrectly overrides any of these methods, the app may show a wrong behavior. As an example, testing AarDict by *MobiGUITAR* we found Bug 1 resulting in the incorrect redefinition of the *onPause()* method of an Activity. This bug produced five crashes during the execution of the *MobiGUITAR* test cases, due to an unhandled *IllegalArgumentException*.

One sequence of events that caused the exception was (e4: open the Dictionary Menu pressing on the menu button of the device, e44: click on a MenuItem, e57: select a dictionary item from the list by a Long Click, e88: rotate the device). In this sequence, after opening the menu (e4), clicking on the MenuItem "Dictionaries" (e44), and selecting a dictionary item from the list by a Long Click (e57), the handler *onItemLongClick* of the event e57 launched a thread to download a dictionary file from the SD card of the device and instantiated a ProgressDialog onto the running activity to show the download progress. When the device rotation event e88 was fired, the running activity was first destroyed and then restarted with the updated configuration layout. Unfortunately, the programmer did not correctly override the onPause() callback of the DictionaryActivity to save all resources shown by the running activity, including the references to the working thread and its ProgressDialog. Hence, when the dictionary download ended and the *verified()* method tried to dismiss the ProgressDialog, we had the crash of the app by an unhanlded *IllegalArgumentException*, because the progressDialog was no more attached to the window manager.

This bug showed us that testing the application by system events like the orientation change can be useful to reveal this type of problem that is very frequent in Android applications.

Bug 2 was found thanks to a sequence of events ending with the input of a wrong URI into an editText view. This event resulted in a java.lang.IllegalArgumentException that was essentially due to the lack of input validation in a method of the app. At the moment, *MobiGUITAR* uses no automatic strategy to define the input values at runtime, but the tester can preliminarily configure the ripper to use a whitelist of specific input values. This bug showed us the necessity of investigating effective techniques for input values definition, such as the ones based on symbolic execution [15].

We found some bugs due to incorrect code reuse such as Bug 3 and Bug 9. When programmers reuse existing code in different contexts, it is possible that they omit to test the same code in the new scenarios, making the wrong assumption that the reused code will behave as well as in the original context. *MobiGUITAR* allowed these bugs to be discovered because it is able to test the apps in different execution scenarios, launching them from different preconditions.

Our tool chain was able to detect three concurrency bugs that are typical of multi-thread software systems like Android applications. We found them by configuring the tool to send events to the apps rapidly i.e., with a short delay between consecutive events.

Moreover, *MobiGUITAR* was also able to discover another bug that depends on programming mechanisms that are not

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE

TABLE I
BUGS DETECTED BY *MobiGUITAR* AND THE RESULTING FAILURES

Bug ID	Subject app	Bug Class	Java Exception	Ticket
1	AardDict	A	IllegalArgumentException: View not attached to window manager	https://github.com/aarddict/android/issues/44
2	Tomdroid	O	IllegalArgumentException: Illegal character in schemeSpecificPart	https://bugs.launchpad.net/tomdroid/+bug/902855
3	BookCat	O	CursorIndexOutOfBoundException	https://github.com/eleybourn/Book-Catalogue/issues/326
4	BookCat	O	NullPointerException	https://github.com/eleybourn/Book-Catalogue/issues/305
5	Wordpress	O	StringIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/206
6	Wordpress	C	BadTokenException	https://android.trac.wordpress.org/ticket/208
7	Wordpress	C	NullPointerException	https://android.trac.wordpress.org/ticket/212
8	Wordpress	C	ActivityNotFoundException	https://android.trac.wordpress.org/ticket/209
9	Wordpress	O	CursorIndexOutOfBoundException	https://android.trac.wordpress.org/ticket/207
10	Wordpress	O	NullPointerException	https://android.trac.wordpress.org/ticket/218

traditional, but typical of Android applications, like the mechanism of the explicit Intents to launch new activities at runtime.

IV. MOBIGUITAR VS. EXISTING TOOLS

We now compare *MobiGUITAR* to two tools available for Android testing: Monkey[5] and Dynodroid[6]. We selected Monkey, a random testing tool that comes with the Android Development Toolkit, because of its popularity in the Android developers community. Dynodroid was chosen because of the variety of event-based testing techniques it implements. Like *MobiGUITAR* both tools test the app by sending it sequences of events. We abstracted a number of tool features that are relevant in event-based testing, such as: the types of events that are fired, the technique used to select the events (e.g., type of implemented testing technique), the possibility of sending events with specific input values, of setting the app pre-conditions or the time interval between events, and the types of testing artifacts produced by each tool.

Table II reports the considered features and how they are implemented by each tool. We see that the tools considerably differ as to the type of testing artifacts they produce and the tool configurability features. *MobiGUITAR* indeed generates several testing output (such as Crash Report, FSM Model, GUI Sequences, executable JUnit test cases, etc.) that provide useful information for the debugging step. Monkey and Dynodroid, instead, do not return any testing artifact else besides the Android LogCat file, so their support to debugging is limited. As to the configurability of the tools, *MobiGUITAR* is the only tool that allows a tester to choose a whitelist of input values to be assigned with input widgets, and likewise Monkey offers the possibility of choosing the event delay and of putting the app in some specific initial states. Dynodroid instead does not provide these features and tests the app always from the initial state of app just installed on the device.

We configured Monkey and Dynodroid to test the same four apps we tested by *MobiGUITAR* by the same number of events that were fired by our tool. We analyzed the exceptions arisen by the tools and the apps' lines of code causing them. We matched them against the ones found by *MobiGUITAR* and that were associated with bugs. Unfortunately, we were not able to debug all the other exceptions because of poor debugging support provided by Monkey and Dynodroid. The evaluation results are reported in Table III. Both tools were able to find only three of the ten exceptions; they did not find any additional bugs. The *IllegalArgumentException* caused by BUG 1 was not raised by DynoDroid because it is not able to rotate the device. Monkey did not detect this exception due to the randomness of its sent events. Both Monkey and DynoDroid did not find the crash related to BUG 2 since it is caused by specific values entered in a given EditText of the GUI. The unhandled exceptions concerning BUG 3 and BUG 9 were not identified by DynoDroid since they are related to particular preconditions of the application under test. Though Monkey allows to test an app by starting from a predefined initial state, it detected by chance just one of these. The crashes related to BUG 6, BUG 7 and BUG 8 were not discovered by DynoDroid since they depend on specific time delay between events. Although Monkey permits to configure such delays, it came across only one of the three exceptions. The *NullPointerExceptions* corresponding to BUG 4 and BUG 10 were not detected by Monkey.

TABLE III
UNHANDLED EXCEPTIONS FOUND BY MONKEY AND DYNODROID

Bug ID	Monkey	DynoDroid
1	No	No
2	No	No
3	Yes	No
4	No	Yes
5	Yes	Yes
6	No	No
7	Yes	No
8	No	No
9	No	No
10	No	Yes

V. CONCLUSIONS

This paper presented a new fully automatic technique to test GUI-based Android apps. The technique is based on the observation, extraction, and abstraction of the run-time state of GUI widgets. The abstraction is used to create a scalable state-machine model that, together with event-based test coverage criteria, provide a way to automatically generate test cases. The technique was demonstrated via an empirical study on 4 open-source software applications. The results showed that the test cases generated were useful at detecting serious and relevant bugs in the apps. Moreover, this study showed that the combination of model-learning with model-based testing techniques is a promising approach for achieving a better fault detection capability in Android app testing.

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE

TABLE II
TOOLS FEATURES COMPARISON

Features	Tools MobiGUITAR	DynoDroid	Monkey
Types of Fired Events	User Events	User and System Events	User and System Events
Implemented Testing Technique	Model Learning & Model Based	Model Learning & Random	Random
Input Value Definition	Yes	No	No
Time Delay Between Events Setting	Yes	No	Yes
Preconditions Settings	Yes	No	Yes
Produced Artifacts	Crash Report Code Coverage Emma Report Executable JUnit Test Cases GUI Sequences Finite State Machine Model Events Sequences Causing Crashes	LogCat Report Code Coverage Emma Report	LogCat Report

Because we are committed to the widest possible dissemination of our tool, we have made *MobiGUITAR* available at http://wpage.unina.it/ptramont/GUIRipperWiki.htm.

REFERENCES

[1] I. Popa, "What problems the next blackberry playbook has to solve in order to be successful," http://www.bbgeeks.com/blackberry-tablet/what-problems-the-next-blackberry-playbook-has-to-solve-in-order-to-\be-successful-888069, 2012.

[2] M. Andrici, "Windows phone: 70k available apps, but are they enough to take on android, ios?" http://www.androidauthority.com/windows-phone-7-apps-67900, 2012.

[3] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-aware adaptive applications: Fault patterns and their automated identification," *IEEE TSE*, vol. 36, pp. 644–661, 2010.

[4] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: ACM, 2013, pp. 623–640. [Online]. Available: http://doi.acm.org/10.1145/2509136.2509552

[5] "Monkey," http://developer.android.com/tools/help/monkey.html.

[6] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: ACM, 2013, pp. 224–234. [Online]. Available: http://doi.acm.org/10.1145/2491411.2491450

[7] T. Takala, M. Katara, and J. Harty, "Experiences of system-level model-based gui testing of an android application," in *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, 2011, pp. 377–386.

[8] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2012. New York, NY, USA: ACM, 2012, pp. 258–261. [Online]. Available: http://doi.acm.org/10.1145/2351676.2351717

[9] A. Memon, I. Banerjee, and A. Nagarajan, "Gui ripping: reverse engineering of graphical user interfaces for testing," in *Reverse Engineering, 2003. WCRE 2003. Proceedings. 10th Working Conference on*, Nov 2003, pp. 260–269.

[10] A. M. Memon, M. L. Soffa, and M. E. Pollack, "Coverage criteria for gui testing," *SIGSOFT Softw. Eng. Notes*, vol. 26, no. 5, pp. 256–267, Sep. 2001. [Online]. Available: http://doi.acm.org/10.1145/503271.503244

[11] F. Gross, G. Fraser, and A. Zeller, "Search-based system testing: High coverage, no false alarms," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA 2012. New York, NY, USA: ACM, 2012, pp. 67–77. [Online]. Available: http://doi.acm.org/10.1145/2338965.2336762

[12] A. Mesbah, A. van Deursen, and S. Lenselink, "Crawling ajax-based web applications through dynamic analysis of user interface state changes," *ACM Trans. Web*, vol. 6, no. 1, pp. 3:1–3:30, Mar. 2012. [Online]. Available: http://doi.acm.org/10.1145/2109205.2109208

[13] D. Amalfitano, A. Fasolino, and P. Tramontana, "Reverse engineering finite state machines from rich internet applications," in *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, Oct 2008, pp. 69–73.

[14] C. Hu and I. Neamtiu, "Automating gui testing for android applications," in *Proceedings of the 6th International Workshop on Automation of Software Test*, ser. AST '11. New York, NY, USA: ACM, 2011, pp. 77–83. [Online]. Available: http://doi.acm.org/10.1145/1982595.1982612

[15] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 59:1–59:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393666

Domenico Amalfitano is a postdoctoral researcher at the University of Naples Federico II. His research interests concern the topics of Software Engineering, mainly including the reverse engineering, comprehension, migration, testing and testing automation of Event Driven Software Systems, mostly in the fields of Web Applications, Mobile Applications and GUIs. He received the Laurea degree in Computer Engineering and the Ph.D. degree in Computer Engineering and Automation in 2011 by the University of Naples Federico II. Contact him at domenico.amalfitano@unina.it

Anna Rita Fasolino is an Associate Professor of Computer Science at the University of Naples Federico II (Italy). Her research interests are in software engineering, maintenance, reverse engineering, Web engineering, and software testing. She received the Laurea degree in Electronic Engineering in 1992 and a Ph.D. in Electronic and Computer Engineering in 1996 by the University of Naples Federico II. Contact her at anna.fasolino@unina.it

Porfirio Tramontana is currently an assistant professor at the University of Naples Federico II. He graduated in Computer engineering in 2001 and had a Ph.D. degree in 2005

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE

at the same university. His research is focused on Software Engineering applied to Mobile and Web applications. His research fields include reverse engineering, testing, maintenance, comprehension, migration of legacy systems, software quality. He is in the program committee of several conferences and workshops related to its research interests. Contact him at ptramont@unina.it

Bryan Ta is a fourth year Ph.D. student at the Department of Computer Science, University of Maryland. He received his M.S. degree in Computer Science from University of Maryland in 2013 and his B.Eng. degree in Computer Science from Hanoi University of Technology, Vietnam in 2009. His research interest includes empirical software engineering, software testing, program analysis and security with a focus on GUI-base and mobile applications. Outside of work, he likes traveling and soccer. Contact him at bryanta@cs.umd.edu

Atif M Memon is an Associate Professor at the Department of Computer Science, University of Maryland. His research interests include program testing, software engineering, artificial intelligence, plan generation, reverse engineering, and program structures. He has served on numerous National Science Foundation panels and program committees, including the International Conference on Software Engineering (ICSE), International Symposium on the Foundations of Software Engineering (FSE), International Conference on Software Testing Verification and Validation (ICST), Web Engineering Track of The International World Wide Web Conference (WWW), the Working Conference on Reverse Engineering (WCRE), International Conference on Automated Software Engineering (ASE), and the International Conference on Software Maintenance (ICSM). Contact him at atif@cs.umd.edu

Digital Object Indentifier 10.1109/MS.2014.55 0740-7459/$26.00 2014 IEEE