MobiCoMonkey - Context Testing of Android Apps

Amit Seal Ami
Institute for Information Technology
University of Dhaka
Dhaka
amit.seal@iit.du.ac.bd

Md. Mehedi Hasan
Institute for Information Technology
University of Dhaka
Dhaka
mehedi.iitdu@gmail.com

Md. Rayhanur Rahman
Institute for Information Technology
University of Dhaka
Dhaka
rayhan@du.ac.bd

Kazi Sakib
Institute for Information Technology
University of Dhaka
Dhaka
sakib@iit.du.ac.bd

ABSTRACT

The functionality of many mobile applications is dependent on various contextual, external factors. Depending on unforeseen scenarios, mobile apps can even malfunction or crash. In this paper, we have introduced *MobiCoMonkey* - automated tool that allows a developer to test app against custom or auto generated contextual scenarios and help detect possible bugs through the emulator. Moreover, it reports the connection between the bugs and contextual factors so that the bugs can later be reproduced. It utilizes the tools offered by Android SDK and `logcat` to inject events and capture traces of the app execution.

CCS CONCEPTS

• **Computing methodologies** → *Simulation tools*; • **Software and its engineering** → *Integrated and visual development environments*; *Software testing and debugging*; *Empirical software validation*;

KEYWORDS

Android, Mobile, Simulation, Contextual Testing, Stress Testing, Emulator, Empirical Software Engineering

ACM Reference Format:
Amit Seal Ami, Md. Mehedi Hasan, Md. Rayhanur Rahman, and Kazi Sakib. 2018. MobiCoMonkey - Context Testing of Android Apps. In *MOBILESoft '18: MOBILESoft '18: 5th IEEE/ACM International Conference on Mobile Software Engineering and Systems , May 27–28, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3197231.3197234

1 INTRODUCTION

Performance of Android applications is dependent on various external contextual factors, such as device and OS heterogeneity, network condition, user interaction with external settings and data connection reliability. Due to the differences in contextual scenarios in different geographic regions, apps may not perform well in all conditions. For example, in a network where packet drop is frequent or network disruption is common; an app may lag, show incorrect result or can even crash. This is a severe issue as only 16% of users return to a bug prone app more than twice [20]. There has been many different testing approaches designed by researchers and industry organizations to find bugs and test apps, such as [15], [2], [1], [14], [7], [18], [22], [19], [6], [21], [17] and [10]. However, as found by Poshyvanyk et al. [11] the approaches designed by researchers and industry organizations alike are seldom used for various reasons; such as lack of exposure, applicability under certain conditions as well as incompatibility with existing Standard Development Kit and other third party tools. Another problem of these approaches is lack of reproducibility of event sequences.

To aid the testing process specially considering the above problems and the contextual factors of mobile applications, we present a lightweight, non obstructive tool called *MobiCoMonkey* (**Mobi**le **Co**ntextual **Monkey**)[3] through this work. It allows developers and testers to test an app in different contextual scenarios. Provided a *config* file, it allows user to test an app while executing a set of contextual events either automatically generated or predefined by user. It extracts necessary information from the android app installer (*apk*) and produces an insightful report that allows one to trace an error or warning back to the relevant contextual conditions. It is capable of working without modifying standard Android SDK and any other third party tools.

2 BACKGROUND STUDY AND RELATED WORKS

As shown by Muccini et al. [16], mobile apps are different from traditional software and require specialized, new testing techniques. As a result, many different works were done including augmenting existing approaches and introducing novel approaches of mobile app testing. Several state of the art works focus on providing contextual testing as a cloud service [10], testing as a service [9], generating different input for Android apps [13], recording input events to later replay the same events to reproduce bugs [8], generating automatic GUI testing [1], on device bug reporting for android applications [14], generating contextual events for stress testing Android Apps [17], intent fuzzing [19] and, automatic discovering, reporting and reproducing mobile app crashes through static and dynamic analysis [15]. However, as surveyed in [12] by Vasquez et al., most of

---

MOBILESoft '18, May 27–28, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5712-8/18/05…$15.00
https://doi.org/10.1145/3197231.3197234

Table 1: Event Types and Possible Values of MobiCoMonkey based on Android Guidelines [4]

GSM Profile	Network Delay	Network Status		Key Events	User Rotation
0	GSM	GSM	HSDPA	Alphabets	Portrait
1	EDGE	HSCSD	LTE	Numerals	landscape
2	UMTS	GPRS	EVDO	Symbols	Reverse Portrait
3	None	UMTS	FULL	Back space	Reverse Landscape
4		EDGE		Delete	

these approaches are not used in industry for various reasons; such as lack of applicability, heavy overhead for integrating in existing approach, and incompatibility with existing tools. Further, the author made several recommendations for future testing approaches such as automatic test cases that evolve with app, has low overhead to integrate with existing practices, and better trace-ability between test cases and features. *MobiCoMonkey* focuses on being a low overhead testing tool that utilizes non modified Android SDK and interacts with the app in a non obstructive manner. Therefore, it is possible to utilize it with other existing approaches. Further, it offers trace-ability between the injected contextual events and the resulting logs generated within the activities of the app. This allows the developer to zero-in on the possibly bug prone features.

3 THE *MobiCoMonkey* TOOL

MobiCoMonkey consists of several independent, configurable components. An overview of the components are discussed in subsection 3.1. Any developer can download it from `GitHub` [3] and use *python3.6*, `JavaFX` and `pip` to utilize its various components. Underneath, it utilizes various high level and low level elements of *Android SDK*, such as `adb`, `emulator`, `shell`, `telnet` and `aapt`. MobiCoMonkey is capable of injecting several low level contextual factors, which are: GSM Profile, Network Delay and Network Status. Additionally, it is capable of changing Rotation View, inserting Key Press events, and toggling Airplane Mode. The categories and values of currently supported events are provided in Table 1. It should be clarified that contextual network condition variation is only possible in emulated devices. Enabling support for such in actual android devices require high overhead modification of the Android OS and rooting. Therefore, *MobiCoMonkey* supports only android emulators as of now.

3.1 Components

The internal components of *MobiCoMonkey* are created using Python 3.6. These components are accessed by the *MobiCoMonkey* GUI to perform various types of actions. Due to various user configurable settings, *MobiCoMonkey*'s nature is quite flexible. The internal components are discussed as follows.

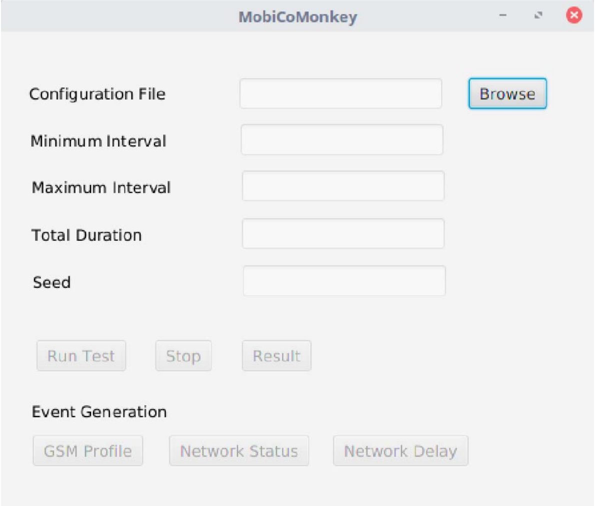
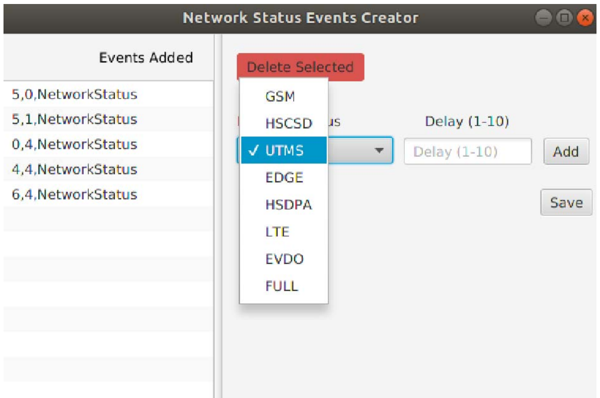


Figure 1: MobiCoMonkey GUI

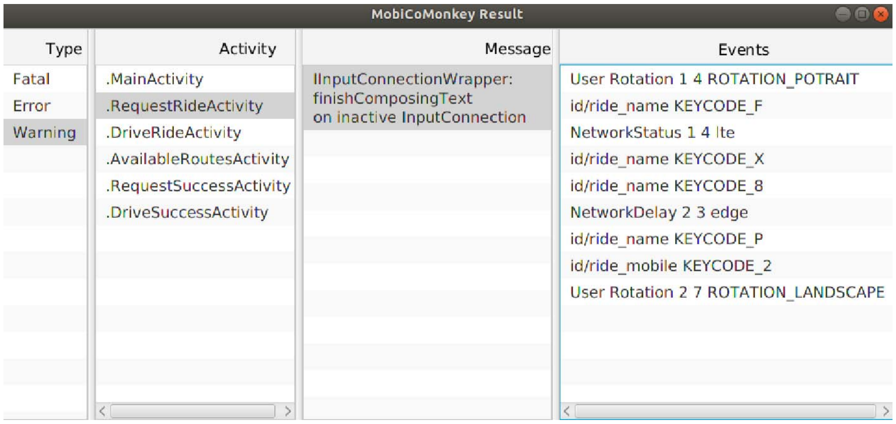


5,0,NetworkStatus
5,1,NetworkStatus
0,4,NetworkStatus
4,4,NetworkStatus
6,4,NetworkStatus

Figure 2: Network Status Events Creator

3.1.1 App Manager: The internal App Manager takes care of installing and uninstalling the app as required. It also extracts permissions requested by the app to determine which *MobiCoMonkey* supported contextual factors will be applicable for contextual testing. Additionally, it extracts UI components of an activity by utilizing `uiautomator dump` when requested. To get the full list of User Interface (UI) elements, it scrolls from top to bottom of the screen until no new UI elements are found in the same activity. As a result, it becomes possible to create mapping of UI elements to each activity regardless of the screen size and view mode.

3.1.2 Contextual Scenario Generator: The contextual scenario generator is utilized when user defined scenarios are not provided for testing. Based on permissions, it generates random contextual scenarios based on a user provided seed value. Accordingly, the same scenarios can always be generated by utilizing the seed value. While generating, it considers minimum interval and maximum interval between contextual factor changes based on user configuration, and the total duration of each scenario. For example, if the



Type	Activity	Message	Events
Fatal	.MainActivity	IInputConnectionWrapper: finishComposingText on inactive InputConnection	User Rotation 1 4 ROTATION_POTRAIT
Error	.RequestRideActivity		id/ride_name KEYCODE_F
Warning	.DriveRideActivity		NetworkStatus 1 4 lte
	.AvailableRoutesActivity		id/ride_name KEYCODE_X
	.RequestSuccessActivity		id/ride_name KEYCODE_8
	.DriveSuccessActivity		NetworkDelay 2 3 edge
			id/ride_name KEYCODE_P
			id/ride_mobile KEYCODE_2
			User Rotation 2 7 ROTATION_LANDSCAPE

Figure 3: MobiCoMonkey Result Viewer

minimum and maximum intervals are configured to be 5 and 12, two Network Status related events it may generate are:

```
NetworkStatus 0 5 hscsd
NetworkStatus 1 10 lte
```

Each line represents one contextual event. The first value describes the type of contextual event, the second value is the index, the third value is the interval after which the next Network Status event will be applied and the fourth value is the current contextual event to be applied. The combined duration of all contextual events for a particular type matches the total duration of scenario. All the other contextual scenarios are produced similarly.

3.1.3 Executor: This can utilize various approaches for contextual testing. For example, it can run in guided approach, where it will run only selected activity screens of an app. Otherwise, it will iterate through all the activities of the provided app. Before executing any contextual events from the scenario, it starts `logcat` service in the Emulator with a clean state. Next, for each activity it runs the contextual scenario events. Consequently, each executed event are saved in a log file for later analysis. To illustrate, a sample log can be:

```
03-19 00:36:35 NetworkStatus 0 8 lte
03-19 00:36:43 NetworkStatus 1 5 gsm
03-19 00:37:05 UserRotation 0 8 ROTATION_REVERSE_POTRAIT
```

The generated log file follows a structure similar to logs generated by logcat[5]. The executor can also additionally run an optional text input field testing thread. Utilizing a top-down approach for each activity, it inputs random text characters in text input fields. Additionally, it stores progress of text fields covered while doing so. Therefore, even if the view is suddenly rotated, it can continue inserting text input sequentially. The executor furthermore watches for fatal status in case of app crash. In such case, it stops execution and saves available logs.

3.1.4 Log Analyzer: It combines logs generated by the logcat emulator and *MobiCoMonkey* executor. The logs are sorted based on temporal ascending order. From the logcat generated log, it extracts the logs of the app at `Warning`, `Error` and `Fatal` level. Generally, these logs are produced by either the system and/or the developer as part of debugging. `Warning` is defined as a non-obstructive malfunction which takes place when something does not execute properly. Next, `Failure` or `Error` is found when an internal activity crashes, while the app resumes execution. For example, `android.view.WindowLeaked` is found when a dialog is dismissed improperly. Lastly, `Fatal` is found when the app crashes fatally and forfeits execution.

3.1.5 GUI Components. For ease of use, several Graphical User Interface components are prepared as part of MobiCoMonkey using `JavaFX`. These components are namely MobiCoMonkey GUI, Contextual Scenario Creator, and Result Viewer as shown in Figure 1, 2 and 3 respectively.

Activities related to utilizing *MobiCoMonkey* can be categorized to three segments, which are Setup & Configuration, Execution, and Result Analysis. These are described in sub sections 3.2, 3.3, and 3.4 respectively.

3.2 Setup & Configuration

To start, a developer has to download and follow instructions from the `GitHub` [3] repository to initially set configuration values of *MobiCoMonkey* in a *config* file. It is assumed that the user already has Android SDK, Android Virtual Device and Android Emulator installed in system. The configuration file is utilized by the *MobiCoMonkey* GUI after user browses and selects the address of the *config* file. Next, the user can utilize the *Event Generator* Window to create a custom scenario of Contextual Events. For each entry of contextual events, the user has to insert the type of event, the nature of the event and the duration of the event. For example, if the user wants to add a Network Status of GSM stage, he simply has to select it from the drop-down menu and input the delay before the next event. While saving the file, the generator will convert it into a Comma Separated Value(CSV) format that can be easily read by the *MobiCoMonkey Executor* module. The user can also simply ignore the Event Generator module and click on 'Run Test'. In such case, the executor module will generate contextual scenarios based on a user defined seed value automatically and will inject those for a prefixed amount of time.

3.3 Execution

After clicking on *Run Test* at first, the app manager module will check for existing installation of the app installer file (*apk*) after initiating the Emulator and will install a fresh version if necessary. Second, the app manager will extract several meta-data related to the app, such as requested permissions and name of activities included in app. Based on extracted permissions, the executor will filter the events in the provided scenario. For example, an app that does not request permission to access the Internet or any data network might not be affected by a sudden contextual change in Network Status or Network Speed. Third, based on the extracted names of activities, the executor will extract further information related to User Interface, such as available text input fields. As a result, it is possible to traverse through activities in a top-down approach, while inserting textual elements in each input fields.

During execution, it creates detailed log similar to *logcat*[5] *detail format*. This log contains information related to android activities, UI components, and injected events. Additionally, it also utilizes the Android internal *logcat* to monitor the errors, warnings and fatal errors. All these information are stored continuously to prevent loss of data. Furthermore, in case of a fatal error it immediately stops execution and stops collecting unnecessary data.

3.4 Result Viewing

Using the *MobiCoMonkey Result Viewer* as shown in Figure 3, the information are categorized to `Warning`, `Error`, and `Fatal` levels. It then allows further filtering to android activities screens so that user is able to view details only from the necessary activities. By navigating to the appropriate activity, the user is able to view the events that occurred during testing. With it, the time-based adjacent, injected contextual events are provided so that user may deduce whether the derived execution event is influenced by the injected contextual events.

4 CONCLUSION

The strength of *MobiCoMonkey* lies in its simple, modular approach for handling different tasks of contextual mobile testing. An inexperienced user without any idea about contextual testing can use it similar to *Monkey*, where it injects random contextual events to an app. On the other hand, an experienced user can design custom contextual scenarios while utilizing any other kind of testing, such as UI testing or Unit testing. Furthermore, it allows the user to utilize it without modifying any existing tools in use or modifying the applications. As a result, it can be integrated as part of agile software methodology with very little overhead. In future, *MobiCoMonkey* can be extended for further analysis of app behavior so that causation and correlation can be derived from injected contextual events and observed behavior of the app. Additional contextual factors, such as GPS, Accelerometer and Bluetooth can be considered for future scope extension.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, and Porfirio Tramontana. 2011. A GUI Crawling-Based Technique for Android Mobile Application Testing. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 252–261.

[2] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 258–261. https://doi.org/10.1145/2351676.2351717

[3] Amit Seal Ami. 2018. LordAmit/mobile-monkey . (jan 2018). Retrieved January 21, 2018 from https://github.com/LordAmit/mobile-monkey

[4] Android.com. 2017. Settings.Global | Android Developers . (may 2017). Retrieved May 28, 2017 from https://developer.android.com/reference/android/provider/Settings.Global.html

[5] Android.com. 2017. Logcat Command-line Tool | Android Studio . (may 2017). Retrieved May 21, 2017 from https://developer.android.com/studio/command-line/logcat.html

[6] Android.com. 2017. UI/Application Exerciser Monkey | Android Studio. (may 2017). Retrieved May 28, 2017 from https://developer.android.com/studio/test/monkey.html

[7] Young-min Baek and Doo-hwan Bae. 2016. Automated Model-Based Android GUI Testing using Multi-level GUI Comparison Criteria. *ASE ’16 (31st IEEE/ACM International Conference on Automated Software Engineering)* (2016), 238–249. https://doi.org/10.1145/2970276.2970313

[8] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *2013 35th International Conference on Software Engineering (ICSE)*. 72–81. https://doi.org/10.1109/ICSE.2013.6606553

[9] Google.com. 2017. Firebase Test Lab for Android | Firebase. (may 2017). Retrieved may 28, 2017 from https://firebase.google.com/docs/test-lab/

[10] Chieh-jan Mike Liang, Nicholas D Lane, Niels Brouwers, Li Zhang, Börje F. Karlsson, Hao Liu, Yan Liu, Jun Tang, Xiang Shan, Ranveer Chandra, and Feng Zhao. 2014. Caiipa: Automated Large-scale Mobile App Testing through Contextual Fuzzing. *MobiCom* (2014), 519–530. https://doi.org/10.1145/2639108.2639131

[11] M. Linares-VÃąsquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. 2017. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622. https://doi.org/10.1109/ICSME.2017.47

[12] M. Linares-VÃąsquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk. 2017. How do Developers Test Android Applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 613–622. https://doi.org/10.1109/ICSME.2017.47

[13] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013* (2013), 224. https://doi.org/10.1145/2491411.2491450

[14] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-device Bug Reporting for Android Applications. In *Proceedings of the 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft ’17)*. IEEE Press, Piscataway, NJ, USA, 215–216. https://doi.org/10.1109/MOBILESoft.2017.36

[15] K. Moran, M. Linares-VÃąsquez, C. Bernal-CÃąrdenas, C. Vendome, and D. Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 33–44. https://doi.org/10.1109/ICST.2016.34

[16] Henry Muccini, Antonio Di Francesco, and Patrizio Esposito. 2012. Software Testing of Mobile Applications: Challenges and Future Research Directions. In *Proceedings of the 7th International Workshop on Automation of Software Test (AST ’12)*. IEEE Press, Piscataway, NJ, USA, 29–35.

[17] Rayhanur Rahman, Amit Seal Ami, and Kazi Sakib. 2017. MobileMonkey - A Contextual Stress Testing Framework for Android Application. *International Journal of Computer Applications* 172, 9 (Aug 2017), 1–7. https://doi.org/10.5120/ijca2017915210

[18] Lenin Ravindranath, Suman Nath, Jitendra Padhye, and Hari Balakrishnan. 2014. Automatic and scalable fault detection for mobile applications. In *Proceedings of the 12th annual international conference on Mobile systems, applications, and services - MobiSys ’14*. ACM Press, New York, New York, USA, 190–203. https://doi.org/10.1145/2594368.2594377

[19] Raimondas Sasnauskas and John Regehr. 2014. Intent fuzzer: crafting intents of death. In *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA) - WODA+PERTEA 2014*. ACM Press, New York, New York, USA, 1–5.

[20] Techcrunch.com. 2013. Users Have Low Tolerance For Buggy Apps - Only 16% Will Try A Failing App More Than Twice | TechCrunch. (mar 2013). Retrieved may 28, 2017 from https://techcrunch.com/2013/03/12/users-have-low-tolerance-for-buggy-apps-only-16-will-try-a-failing-app-more-than-twice

[21] Xamarin.com. 2017. Mobile App Testing On Hundreds Of Devices - Xamarin Test Cloud. (may 2017). Retrieved may 28, 2017 from https://www.xamarin.com/test-cloud

[22] Hui Ye, Shaoyin Cheng, Lanbo Zhang, and Fan Jiang. 2013. DroidFuzzer. *Proceedings of International Conference on Advances in Mobile Computing & Multimedia - MoMM ’13* (2013), 68–74. https://doi.org/10.1145/2536853.2536881