Automatic Android Application GUI Testing—A Random Walk Approach

Haoyin LV
School of Information Engineering, Longdong University, Gansu, China
Email: ldxylhy2012@163.com

***Abstract*—Android is one of the most popular operating system for smart phone and tablets. The fast growing of applications in the market with complex functionalities makes the testing of them a demanding task. Random testing is an effective measure to help testers to expose the bugs within the application. However, the built-in Monkey tool is still too primitive to expose bugs effectively. In this paper, we propose our improved monkey-testing tool, which improves the fault-detection capability of the monkey tool significantly with a series of optimizations. The empirical study shows that our improved monkey tool is more effective than the monkey on real-life Android applications.**

***Index Terms*—Random Testing; Test Case Generation; Android; Random Walk**

I. Introduction

In global smart phone market, the Android platform has reached around 85% of market share, followed by iOS and Windows Phone. There are more than 1.2 million Android applications within the application market. For application developers, it is crucial to ensure the quality of their applications to retain and enlarge their end user community. As a result, application developers use testing to eliminate the bugs affecting user experiences (such as crash, ANR, exceptions). Random testing is one effective to expose system-level bugs within the application. It has the merit of requiring no source code or program specification, and can be fully automatic.

The monkey tool is the built-in random testing tool within the Android framework. It can execute either in fully random manner to generate user or system events or it can execute a given test script. However, several problems with the monkey tool seriously reduce its effectiveness.

Firstly, many of the events generated by Monkey are invalid or redundant. Compared with a human tester, the monkey has one obvious disadvantage: it has no knowledge of the GUI structure of the application under test. As a result, Monkey may generate invalid events by sending touch or click events to a UI widget that is non-responsive to any events. Furthermore, Monkey may also generate redundant events by sending several equivalent events within the boundary of the same widget. Thus, these invalid or redundant events make the monkey tool ineffective to expose the faults earlier in the testing process.

Secondly, the randomly generated events from the monkey tool may be ineffective to expose faults in the application. In previous work [2], researchers found that evenly spread the test case within the input domain can increase the rate of fault detection significantly when compared with random testing. The monkey tool only ensures the random selection of events in each UI state rather than the evenly spread of test cases within the entire input domain. Thus, if we can evenly spread the test cases (A test case is a sequence of input events.) within the entire input domain of an Android application, the fault detection rate of the test generation.

To address the above problems, we propose our random walk-based tool as a significant improvement over the monkey tool. Our tool not only “understand” the dynamic UI structure of the application under test, but it also makes use of the idea of random walk on GUI transition graph to evenly spread the event sequences to expose faults earlier in the testing process.

The main contributions of our work are as follows: (1) It enable the improved monkey tool to generate only those valid and non-redundant user inputs for the application under test. (2) It improves the rate of fault detection using the idea of random walk to evenly spread the test cases within the input domain. (3) It performs a systematic empirical study using a set of Android applications downloaded from application market to validate our proposed testing tool.

We organize the rest of paper as follows: Section II presents how we generate valid and non-redundant events. Section III describes our test case generation algorithm based on random walk on the GUI transition graph. Section IV presents our empirical study with results analysis. Section V describes related work, followed by the conclusion in Section VI.

II. Effective Events Generation

The test case of Android application is composed of a sequence of events. Thus, to generate effective test case, it is crucial to generate effective events first.

There two types of events consumed by an Android application: user event and system event. The user of the application initiates the user events while Android OS produces the system events (e.g., broadcast events).

From the perspective of effective events generation, there are several problems with the monkey tool: (1) The user events generated by the monkey tool are often invalid and redundant, owing to its unawareness of the application GUI structure. (2) The monkey tool does not support the generation of system events, i.e., broadcast events.

In the following sections, we will discuss in detail how we can effectively generate each type of event.

A. Generation of User Events

The user events are the most important type of events consumed by an application. When comparing a monkey tester and a human tester, we will find the most important advantage of human tester is that he/she understands the GUI structure of the application under test and will only send events to the UI widgets that respond to user events. Furthermore, a human tester is aware of the area taken by a widget. Thus, it will not repeatedly send equivalent events falling into the same widget. Thus, to make our testing tool smart, we must enable it to "watch" and "understand" the application under test first.

1) Understanding Dynmic GUI State: To make our smart-monkey tool understand the dynamic GUI structure of the application under test, our monkey tool interact with the *ViewServer* of the Android system to get the GUI information of the current active view. The ViewServer resides on the port 4939 of the Android device to provide service, which can be started or shutdown through adb commands.

More specifically, our smart monkey tool connects to the ADB (Android Debug Bridge) server on the device through ADB client via local loop TCP connection. Our smart monkey can issue the DUMP command to the ViewServer through the ADB tool. Upon receiving the DUMP command, the view server will return a list of all the widgets with their positions and properties in the current active activity. In fact, the Hierarchy Viewer tool in the Android SDK uses the same approach to acquire the hierarchical Widgets information from the view server.

Although the ViewServer provides a convenient way for our smart monkey tool to inspect the GUI state of the android application, it is too slow to return the information due to its heavy use of reflection mechanism. Since our smart monkey inspect the GUI state dynamically, we must optimize the GUI state "DUMP" process.

We observe that not all properties returned by the "DUMP" command from the view server is useful for testing. We only need those properties that are directly useful for our smart monkey tool. So we only keep a subset of properties for each control, including the name, position, width, height, and its relative position with parent control. We also keep other properties useful for state determination, including isClickable(), isLongClickable() to check whether the control can respond to click events. The VerticalScrollBarEnabled() and HorizontalScrollBarEnabled() are also kept to determine its responsiveness to scroll events. We revised the ViewServer implementation to realize a new command DUMPT based on the work of DUMPQ, which is a revision of DUMP command implementation to get lightweight GUI information. But our DUMPT acquires a different subset of properties from the DUMPQ implementation to support effective testing. Our performance evaluation shows our implementation can achieve a $5\times$ to $10\times$ performance boost than the DUMP command.

2) Generating Valid User Event: After our smart monkey tool can "understand" the current GUI state, it can proceed to generate valid and events.

To generate valid user events means the smart monkey must only send the generated events to those GUI widgets handling them. Sending an event that no widget will handle is a waste of testing effort.

We classify the user event sent to an application into 5 categories:

For KeyPress event, we consider the press on the "Menu" and "Return" key in each program state is valid.

For Tap or Long Tap event, we consider a tap or long tap on a widget whose isClickable() property is true as a valid event.

For Input event, we consider the input of characters into an editable widget as a valid event.

For move event, a move on a widget whose isVerticalScrollBarEnabled() or isHorizontalScrollBarEnabled() is true is considered valid. A move on a screen (including 4 possible directions) is also a valid event.

We further want the event generated by our smart monkey tool to be non-redundant. The test case of Android application is a sequence of events. Thus, given a specific test case length, the input space of the application is determined by the input domain for a single event. For the monkey tool, the events generated are coordinate-based. Thus, for the most tap and long tap events, the input domain consists of all the points within the current screen. However, for our smart monkey tool, it understands that all the tap or long tap events on the same widget are equivalent. Thus, we can perform equivalence partitioning on the domain for the tap and long tap events. For a certain UI state, we only need send one tap and one long tap event to each control that handles them. All other events are redundant and can be eliminated for one UI state during test case generation.

B. Generation of System Events

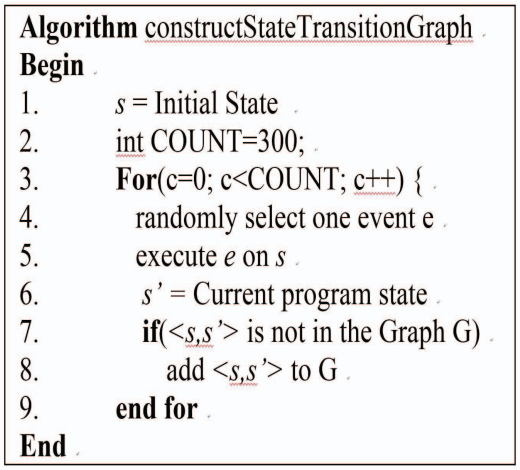
The system event our smart monkey tool supports is the broadcast event. For an android application, not all broadcast events are interesting. To be notified by a broadcast event, an application must register the corresponding broadcast receivers in the AndroidManifest.xml file in its project.

Thus, to generate valid broadcast event, we adopt the apktool to extract the AndroidManifest.xml file from the Android installation file (i.e., the .pkg file) into the sdcard. Then we parse the AndroidManifest.xml file to extract the broadcast events the application listens. Finally, our smart monkey will instantiate the corresponding broadcast event.

In this section, we have described how we can generate valid and non-redundant user events as well as effective system events. Since a test case for android application consists of a sequence of events, we will discuss how we optimize the test case generation process for our smart monkey tool in the next section.

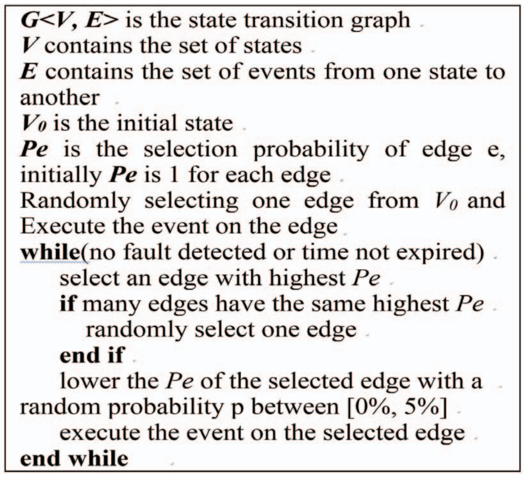
III. The Test Case Generation Algoritm Based on Random Walk

The generation of effective test suite for Android application consists of two steps: GUI transition graph construction and



```
Algorithm constructStateTransitionGraph
Begin
1.      s = Initial State
2.      int COUNT=300;
3.      For(c=0; c<COUNT; c++) {
4.        randomly select one event e
5.        execute e on s
6.        s' = Current program state
7.        if(<s,s'> is not in the Graph G)
8.          add <s,s'> to G
9.      end for
End
```

Fig. 1. The algorithm for constructing state transition graph.



```
G<V, E> is the state transition graph
V contains the set of states
E contains the set of events from one state to another
V0 is the initial state
Pe is the selection probability of edge e, initially Pe is 1 for each edge
Randomly selecting one edge from V0 and Execute the event on the edge
while(no fault detected or time not expired)
    select an edge with highest Pe
    if many edges have the same highest Pe
        randomly select one edge
    end if
    lower the Pe of the selected edge with a random probability p between [0%, 5%]
    execute the event on the selected edge
end while
```

Fig. 2. The biased random walk test case generation algorithm.

random walk on the generated transition graph for test case generation. The GUI state transition graph are dynamically built with random exploration while event sequence generation is based on the adaption of random walk traversal on the graph.

A. Dynamically Building GUI Transition Model

An Android application can be seen as a reactive system: it reacts to each event in a test case by transits from one state to another. To effectively generate test cases, our smart monkey must not only understand each GUI state, but it must also learn how the GUI states transits from each other based on different events.

In this work, we define that two states are equivalent if they have the same Acitivity ID. Note that, there may be different definitions of GUI state equivalence, whose impact is may be explored in future work.

Our smart monkey dynamically learns the GUI state transition model with a bootstrapping process as shown in Fig. 1. At the start of the testing process, the smart monkey knows nothing about the application under test. Then it starts from the initial state of the application under test, finds its corresponding sets of valid events, and randomly picks one to send to the application. If the application states have changed, it will add a new node in the state transition graph of the application under test. For each state, smart monkey maintains a hash map of valid events. When a new event is sent to a certain application state, a new edge to a new state or an existing state will be added into the state transition graph. Thus, during the testing process, the GUI state transition model (graph) becomes more and more complete. The GUI state transition model construction process ends when certain number of events is executed starting from an initial state. In our experimental setting, we set the number of events as 300. Basically, we use a fast random testing as the bootstrapping process.

B. The Test Case Generation Algorithm Based on Random Walk

We use a biased random walk algorithm to generate test cases from the state transition graph built from the modeling process in the last section such that the test cases generated can spread evenly within the input domain.

As shown in Fig. 2, we use a biased random walk algorithm to generate event sequence against a program p. We start from the initial state of the state transition graph built in the last section. This initial state corresponds to the entry point activity of the application. Each edge is given a probability *Pe* initialized with 1. Our algorithm first randomly selects one edge starting from the initial state and executes its corresponding event against the application. Then the algorithm goes into an loop until a fault is found. Within the loop, the algorithm tries to select an edge with highest *Pe*. When there are tie cases, it just randomly selects one edge. If an edge is selected, then its *Pe* is lower with a small probability between [0%, 5%] such that it may have a relatively smaller chanced to be selected again. Then the event corresponding to the newly selected edge is executed. The program halts when a fault is found or a predetermined time expires. We use a biased random walk on the transition graph such that those explored edges will have a lower probability to be explored again. In this way, we can more evenly spread the test cases within the input domain.

IV. Empirical Study

In this section, we perform an experimental study to evaluate the effectiveness of our proposed random walk-based test case generation technique.

A. Research Questions

RQ1: Is our technique more effective than the monkey tool in reducing the number of test cases required to expose first fault to test mobile application?

RQ2: Can our random walk technique reduce the time needed to find first fault than random to test mobile application?

B. Experiment Setup

As shown in Table I, we use six real-life open source applications running on Android 5.0 to evaluate our improved test case generation technique. The Alarm Clock, HNDroid,

TABLE I
SUBJECT PROGRAMS.

Application	Description
MusicPlay	A music application
Alarm Clock	An customized alarm clock
HNDroid	A news application
Reader	A reader application
Dialer	A application for making and answering calls
Photostream	A photo management tool

Reader, Dialer, Photostream, and MusicPlay applications are all popular third-party open source Android applications.

We implement our random walk test case generation algorithm as a tool for testing Android applications. To make it easy for us to compare with monkey, we implement the tool in java and install it within the Android application framework similar to the monkey tool. In this way, both tools can be started with an command through ADB interface.

Our experiment is performed on a PC having a i7 Quad-Core 3.2 GHz processor, 16 GB of RAM, and running Ubuntu OS.

We use two effectiveness metrics. The first one is *F-measure*, the *number of test cases needed to detect the first failure*. It reflects the fault detection ability of the generated test cases. The second one is the *time used to find the first fault*. This metric also considers the time needed for test case generation.

We applied both the random walk tool and the monkey tool to test one application at a time. For both strategies, we repeatedly generate new test cases until a fault is found. To eliminate the impact of pseudo-random generator, we repeat the experiment 50 times with different seeds to average out the results. If the application crashes or throws an exception, we consider a fault is exposed.

C. Results and Analysis

We first compare the number of test cases to first fault (F-measure) between Random Walk tool and monkey tool as shown in Fig. 3. The x-axis denotes different applications while the y-axis is the number of test cases required to detect the first fault. We can see that for all subject programs, the random walk tool uses much fewer test cases than random to expose the first fault. The results shows that the adoption of the idea of biased random walk on the state transition graph of Android can significantly improve the effectiveness of the generated test cases.

From the results of the first research question, we can find the test cases generated by our Random Walk tool is more effective than monkey to detect fault. However, the random walk tool is also costly to generate test cases. As a result, we need compare the end-to-end time of exposing the first fault between the two techniques.

The comparison of the time needed to find the first fault between random walk and monkey is shown in Fig. 4. The x-axis represents subject programs while and the y-axis represents the time in seconds used to find the first fault. The

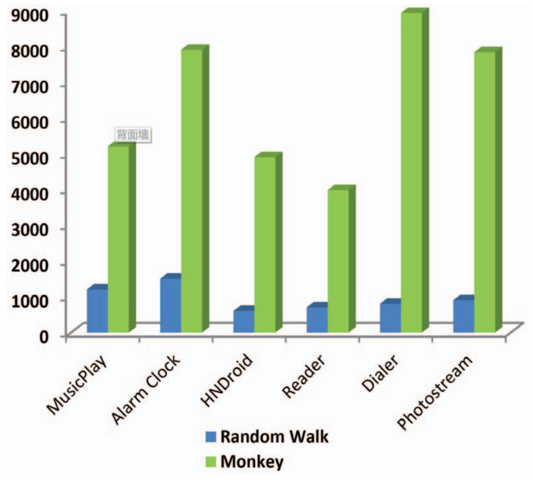


Fig. 3. Comparison of F-measure between Random Walk and monkey tool.

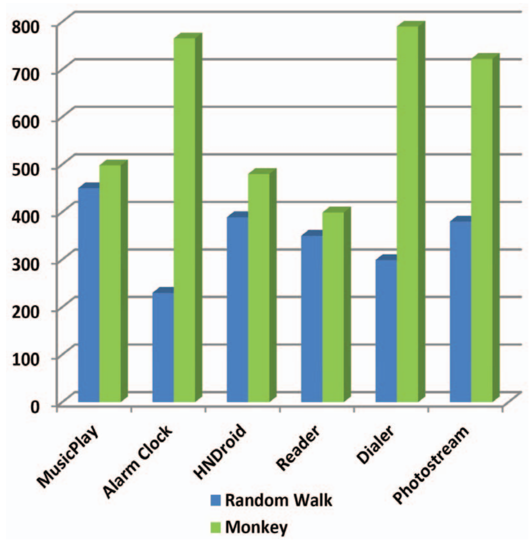


Fig. 4. Comparison of time to first fault between ART and Random.

time here includes both the test case generation time and the test execution time. We can see that random walk consistently uses less time than the monkey tool to expose the first fault for all subject programs. In particular, the results for alarm clock, dialer, and the PhotoStream shows random walk uses much less time than monkey to detect the first fault.

Thus, we can answer the second research question that our random walk tool uses less time than the monkey tool to detect the first fault.

From the experimental result, we can answer our research questions as follows: the random walk test case generation tool can significantly reduce the overall time as well as the number of test cases to detect first fault than the monkey tool.

V. RELATED WORK

There are many works on testing and verifying Android applications [4–7], [9].

In [4], Liu et al. adopted the adaptive random test case generation technique for Android application. To use the traditional ART algorithm, they defined the distances between two test cases, i.e., the distance between two event sequences. To be specific, they defined each event as a binary tuple: one is the event type while the other is the event value. And the event sequence distance is defined as the combination of sequence type distance and event distance. Their experimental results show that the ART algorithm for test case generation can significantly reduce the time to first failure.

Dynodroid [1] is another test input generation system for Android application. It relies on MonkeyRunner to send user events and system events to the target device. It has three event selection strategies: selecting events with smaller occurrence frequency, selecting events randomly, and selecting events randomly but deducing its selection probability after it is selected.

Shuai Hao et al. implemented a tool called PUMA [3] for random exploration of simulated user clicks. The strength of PUMA is not in the strategy of exploring apps, but in its overall design. PUMA is a dynamic analysis framework that can be extended according to analysis requirements. Users can perform different kinds of dynamic analysis by extending the PUMA framework, such as accessibility violation detection and Ad fraud detection.

VI. Conclusion and Future Work

Effective test case generation for mobile applications can help improve its quality. In this walk, we propose to model an Android application as a state transition graph and then use a biased random walk algorithm to generate test cases such that event sequences can be evenly spread within the input domain. We performed an experimental study on 5 open source Android applications to evaluate the effectiveness of our random walk tool. Our evaluation results show that our ART test case generation technique can both reduce the number of test cases and the time required to expose first failure when compared with random technique.

In future work, we will study the impact the definition state equivalence on our proposed test case generation technique. We will also perform larger empirical study on real-life applications to evaluate our proposed technique. Since there are also some other test case generation techniques other than monkey, we will also compare with them in future work.

References

[1] Aravind MacHiry, Rohan Tahiliani, and Mayur Naik. "Dynodroid: An input generation system for Android apps," *FSE'13: ACM Symposium on Foundations of Software Engineering*. 2012.

[2] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. "Adaptive random testing: the ART of test case diversity," *Journal of Systems and Software*, 2009. doi:10.1016/j.jss.2009.02.022.

[3] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services, ser. MobiSys'14*. New York, NY, USA: ACM, pp. 204–217, 2014.

[4] Zhifang Liu, XiaopengGao, and Xiang Long. "Adaptive random testing of mobile application," *IEEE Computer Engineering and Technology*, pp. 297–301, 2010.

[5] A. Memon, I. Banerjee, and A. Nagarajan, "GUI Ripping: reverse engineering of graphical user interfaces for testing," in *Proceedings of the 10th Working Conference on Reverse Engineering, ser. WCRE 03. Washington, DC, USA: IEEE Computer Society*, 2003, p. 260–.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR – a tool for automated model-based testing of mobile apps," *IEEE Software*, vol. PP, no. 99, pp. NN–NN, 2014.

[7] 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, pp. 623–640, 2013.

[8] 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.

[9] Y. Liu, C. Xu, and S. Cheung, "Verifying android applications using java pathfinder," *The Hong Kong University of Science and Technology*, Tech. Rep., 2012.