LAND: A User-Friendly and Customizable Test Generation Tool for Android Apps*

Jiwei Yan
Tech. Center of Softw. Eng.
Institute of Software, CAS
Beijing, China
yanjw@ios.ac.cn

Linjie Pan
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China

Yaqi Li
Tech. Center of Softw. Eng.
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China

Jun Yan †
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China

Jian Zhang
State Key Lab. of Computer Science
Institute of Software, CAS
Univ. of Chinese Academy of Sciences
Beijing, China

ABSTRACT

Model-based GUI exploration techniques are widely used to generate test cases for event-driven programs (such as Android apps). These techniques traverse the elements of screens during the user interaction and simultaneously construct the GUI model. Although there are a number of automatic model-based exploration tools, most of them pay more attention to the exploration procedure than the model reusing. This paper presents LAND, an effective and user-friendly test generation tool based on GUI exploration of Android apps, which constructs an elaborate window transition model "LATTE" that considers more Android specific characteristics and provides a customizable test generation interface by reusing the model. Experiments on 20 real-world Android apps are conducted to construct their models as well as test cases. The experimental results indicate that LAND can achieve higher code coverage and trigger exceptions in shorter sequence. It is also demonstrated that LATTE can be well reused under different requirements of test suite generation. A demo video of our tool can be found at the website https://www.youtube.com/watch?v=iqtr12eiJ_0.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

Android GUI Model; Targeted Test Generation; Dynamic Modeling

ACM Reference Format:
Jiwei Yan, Linjie Pan, Yaqi Li, Jun Yan, and Jian Zhang. 2018. LAND: A User-Friendly and Customizable Test Generation Tool for Android Apps. In *Proceedings of 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'18).* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3213846.3229500

1 INTRODUCTION

Test case is an essential element in the field of testing. For event-driven programs, such as Android apps, a test case is an event sequence of unlimited length, which is composed of events (e.g., click, scroll, input) in arbitrary order. An Android app can be regarded as a collection of widgets, each of which is defined in an `Activity` class that is provided by Android system to interact with users. The user operations on the widgets will trigger corresponding events and drive the app to transfer from one window to another. The main challenge of GUI testing lies in how to generate effective test cases to achieve higher code coverage and stronger fault detection ability. In order to achieve this goal, GUI exploration techniques, especially model-based ones, are widely adopted via the dynamic information obtaining and event picking.

In recent years, several model-based GUI exploration approaches [5–7] for Android apps have been proposed. Most of them focus on generating test cases during exploration while ignoring the reusing of generated models for further testing. The systematic exploration process for model construction is usually time-consuming. However, the obtained model is rarely or merely used to generate small-scale test suite, leading to the wasting of testing ability and low efficiency of test generation.

Thus, we present LAND, a user-friendly and customizable test generation tool for Android apps, to improve the efficiency of model-based GUI testing. Our tool can work without source code of the application under test (AUT). It constructs the designed model during exploration, which can be reused to satisfy different testing requirements. To design an elaborate model of AUT, more Android specific characteristics like the back stack [1], Activity launch mode

*This work is supported by National Natural Science Foundation of China (Grant No. 61672505), the National Key Basic Research (973) Program of China (Grant No. 2014CB340701), and Key Research Program of Frontier Sciences, CAS, Grant No. QYZDJ-SSW-JSC036.
†Corresponding Author. Email: yanjun@ios.ac.cn

ISSTA'18, July 16–21, 2018, Amsterdam, Netherlands

ACM ISBN 978-1-4503-5699-2/18/07…$15.00
https://doi.org/10.1145/3213846.3229500

as well as dynamic widget status information are considered. Besides GUI information, we also link the transitions in this model to the corresponding executed code snippets via a labeling mechanism. Moreover, a metric named "state similarity" is proposed to balance the accuracy and the cost during the analysis. Finally, users can make use of the constructed model under several requirements of test generation, including activity directed, widget directed, label directed, graph traverse and record-and-replay. The generated test cases are in the form of runnable, readable and editable test scripts.

2 LAND

In this section, we introduce the system architecture and techniques used in our tool LAND.

2.1 System Overview

LAND takes an Android app as input and outputs the constructed LATTE model as well as the generated test cases. Figure 1 gives the high-level overview of LAND, which contains four modules: *Pre-Processing*, *GUI Exploration*, *Test Generation* and *Report Generation*.

- The *Pre-Processing* module instruments the AUT on Dalvik byte-code for target labeling and coverage computing, and it will automatically generate exploration profile according to the features of AUT.
- The *GUI Exploration* module iteratively updates the constructed LATTE model in the loop of executing three sub-modules, including the event choosing and triggering (*Event Executor*), information monitoring (*Monitor*), as well as state abstracting and model constructing (*Model Constructor*).
- Making use of the embedded information in LATTE, the *Test Generation* module reuses the constructed model to generate test cases under several requirements.
- The *Report Generation* module gives detailed code coverage and crash reports for the corresponding exploration.

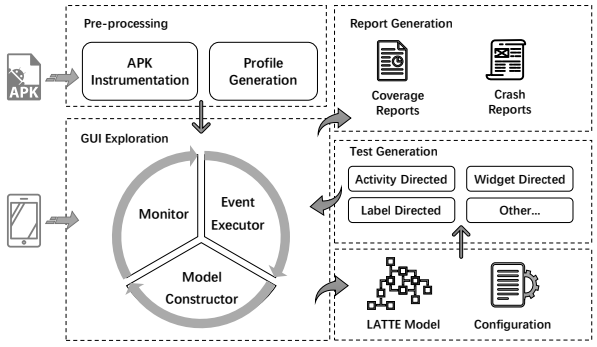


Figure 1: System Overview

2.2 GUI Exploration

Our GUI exploration approach takes an instrumented Android apk file and its profile as input, and outputs the corresponding LATTE model. LATTE is a window transition graph who splits each Activity into one or more states, depending on both the widget and back stack, and records the transition information between windows. Details of LATTE model can be obtained from our previous work [9].

The basic work-flow of this approach is an automatic iterative operation of the GUI exploration and model construction.

We initialize a LATTE model with an empty state and an empty transition set. The first event to be executed automatically by *Event Executor* is app launching, after which the app will be driven to its first state (usually corresponding to the `MainActivity`). During the event execution, the sub-module *Monitor* will record the GUI widget information by *Robotium* [4] script and get the event list to be executed. We also get the instrumentation-related logs through *Android Debug Bridge*. Then, *Model Constructor* will use the collected information to update the model. To avoid exploring too many states and get an acceptable model size, we define a metric to measure the similarity of two states. Any two states will be merged if their similarity is greater than a pre-defined threshold. Besides the package, activity and basic widgets information, the state similarity is also measured according to the widget status and back stack. After that, based on the current model, the first unvisited state, i.e., the first state that still contains unvisited events, will be picked using Depth-First Search (DFS) or Breadth-First Search (BFS) traversal strategies. The first unvisited event (according to the layout) in that state will be the next event to be executed. The app may be driven to a new window after a new event is triggered, and then we repeat the above procedure until all events are traversed.

2.3 Test Generation

After exploration, the constructed LATTE model can be used to generate several types of test cases for further testing. For an app, LAND could construct a LATTE model and provide five strategies to generate different test cases, including activity directed, widget directed, label directed, graph traverse and record-and-replay.

Activity Directed. If the test target is a specific activity, LAND will use the corresponding activity name to retrieve states in LATTE model and return a set of test cases to reach these states. The generated test cases are in the form of *Robotium* runnable script, as shown in Figure 2.

```
@Test
public void test01() {
//state=MainActivity, id=1, isTarget=false, isQuit=false
solo.clickOnView(solo.getView(2131296262));
/* other code */
//state=TargetActivity, id=8, isTarget=true, isQuit=false
solo.clickOnScreen(45,1310);
}
```

Figure 2: A Test Case Example

Widget Directed. When one type of widget, e.g. `Button`, needs to be comprehensively tested, LAND retrieves these widgets in LATTE model and generates test cases to reach them. For specific kind of widget, like `EditText`, which accepts `string` input from users, LAND will reach these widgets and then generate various strings based on black-box (e.g., boundary value analysis) and fuzz testing techniques with some user-given special characteristics (e.g., asterisk) to test them thoroughly.

Label Directed. In practice, testers often pay more attention to some specific parts of codes, for example, the methods related to the functionality that they want to analyze. To embed the code

information into our model, we introduce a label set. In general, each element in the set corresponds to a code snippet. For instance, we can set each label to represent a distinct method of the app (*method label*), all the methods in a class (*class label*), or instructions with a specific keyword (*keyword label*). The mapping rule for the labels and the code snippets is designed according to the actual testing or analysis requirements. With the mapping rule, the labeling procedure is implemented via code instrumentation in our approach. For example, the keyword label `Camera` represents all code instructions containing the keyword "Camera", e.g., code instruction `Landroid/hardware/Camera;->open(`.

For label directed test generation, the set of transitions whose label set contains at least one element in the target label set will be retrieved. Then we leverage several graph algorithms on the LATTE model to extract the dependency relationships between states and transitions. Finally, we try to find event sequences that cover these transitions with some heuristics.

Graph Traverse and Record-and-Replay. LAND also provides two other strategies that aim at generating test cases for the whole app. For the graph traverse generation, LAND will traverse the LATTE model to generate test cases that can cover all transitions. And for the record-and-replay strategy, all the operations that have been executed during the GUI exploration will be recorded and transferred to a set of test scripts for replaying.

2.4 Usage

In this section, we will introduce how to use LAND to construct a LATTE model and generate test cases for an Android app. The major configuration items in LAND are shown in Figure 3.

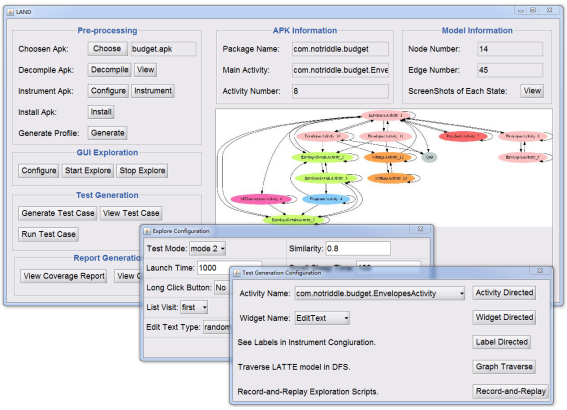


Figure 3: Main Interface of LAND

When LAND is launched, the user first selects an apk to be tested. Then LAND repackages it into a new apk file apk_1 via decompilation and instrumentation. The automatic exploration script that implemented using *Robotium* will also be packaged as an apk file apk_2. After clicking the button "Install", both apk_1 and apk_2 will be installed on the device (or emulator). Then the user can start the exploration by clicking "Start Explore". Both the instrumentation and exploration procedures can be configured by users with a set of editable items. While exploring, the current model will be transformed into *dot* format and be visualized using software *graphviz*, which will be displayed on the window in real-time.

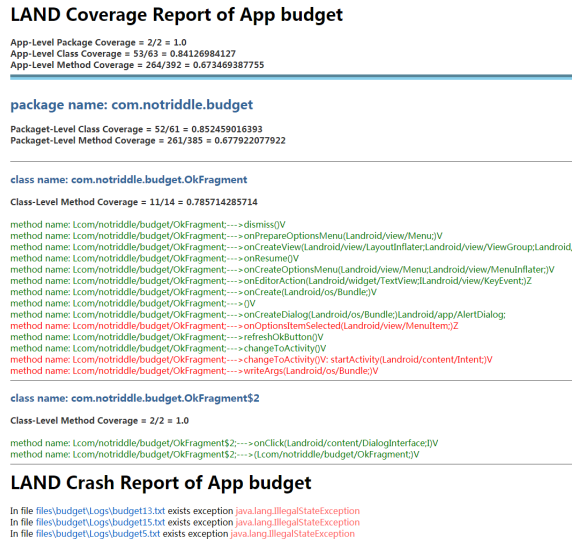


Figure 4: Coverage and Crash Reports

When finished, the user can obtain the coverage reports as well as crash reports. Figure 4 shows one coverage report in different granularity, including the app-level, package-level and class-level. The covered methods during app execution are marked in green color and the uncovered ones in red. It also displays a crash report that contains the name of exceptions and corresponding scripts. For each state in the LATTE model, LAND will record one screen-shot, so that testers can view and understand the model intuitively.

Finally, testers can generate customized test cases to satisfy their requirements. These generated test cases also can be executed by clicking the button whose label is "Run Testcase".

3 EVALUATION

To evaluate the effectiveness of our tool, we collect 20 real-world apps to construct LATTE models and generate test cases for them. Among these apps, ten are open-source ones downloaded from F-droid [2] and the rest are from commercial markets (without source code). Due to the limit of the space, we only show the summary information of these apps. Their sizes vary from 0.14MB to 14.86MB, and their functionalities are varied (like browser, media, and finance). The maximal number of classes and methods are 3752 and 25461, respectively. All of our experiments are carried out on a mobile smartphone with 1.82GHz CPU and 3GB RAM.

3.1 Code Coverage

Code coverage of AUT is taken as a significant measurement index, which can be calculated by analyzing the basic byte-code information and the run-time executed code information. To evaluate the effectiveness of our approach, we pick two popular automatic testing tools Monkey [3] and Dynodroid [8] as well as two well-known model-based traversal tools A^{3}E [6] and PUMA [7] for comparison.

During exploration, some tools fail to report the test results for some instances. As shown in Figure 5, we give the number of apps that can be successfully executed in our benchmark. For Dynodroid, it fails to explore unsupported apps or inconsistent versions of app (cannot execute on SDK 2.3). For PUMA, on the one hand, some

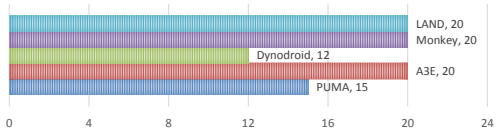


Figure 5: Number of Available Apps for Exploration

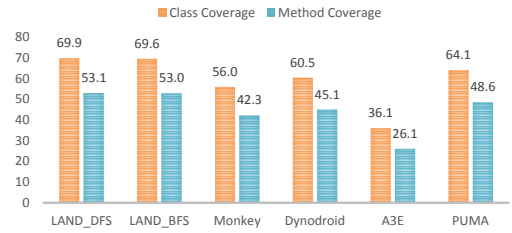


Figure 6: Coverage Comparison on Dalvik Byte-Code

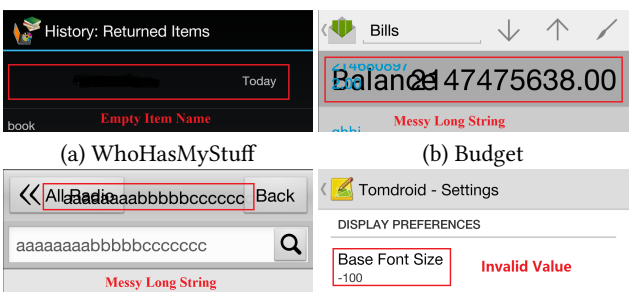
apps are unsupported, which causes the tool to throw exceptions during run-time. On the other hand, too short launch time leads to a wrong model of some commercial apps and causes a low coverage.

Figure 6 shows the detailed information about class and method coverage results. LAND, Monkey, and A^{3}E work well for all the 20 apps, in which A^{3}E may stop at the first window when testing some commercial apps and thus have low coverage. We adopt two traversal algorithms, DFS and BFS, in the exploration of LAND. The results demonstrate that LAND can be applied to more apps and is able to reach higher coverage in most cases.

We also record the number of exceptions detected by Monkey, Dynodroid, and LAND, which is 14, 5, and 22, respectively. The results indicate that the higher coverage helps LAND to find more crashes than other GUI traversal tools under our benchmark. We also find that the average length of event sequences generated by LAND to trigger exceptions is less than 10, which means we can use a short event sequence to trigger the exception.

3.2 Target Directed Test Generation

Firstly, we use widget directed test generation to test `EditText` widgets. By testing, we observe that some apps crash by throwing exceptions, e.g, app *Budget* will throw `NumberFormatException` when the input string is too large to be cast into a number, and app *aGrep* will throw `PatternSyntaxException` when the input text begins with "*" as a regular expression. Besides, some apps display unfriendly interfaces, including empty item name, messy string displaying and blank window caused by abnormal font size setting. Some of these abnormal displays are shown in Figure 7.



(a) WhoHasMyStuff (b) Budget

(c) Cradio (d) TomDroid

Figure 7: Abnormal Displays

Then, we use label directed test generation to test the label related transitions with randomly picked labels. Experiments are carried out between LAND and Monkey to compare the minimal sequence length they need to cover the given target. To get the minimal sequence length of Monkey, we implement a script to repeatedly run Monkey with the event limits increased by 1000 in each iteration, until the given target is covered. The results show that both LAND and Monkey can trigger all transitions that cover all labels. However, the average length of test cases generated by LAND that cover all labels is 5, while the number for Monkey is 11000, which shows that using LATTE we can generate effective and compact test cases for user-specified testing.

4 RELATED WORK

Some researchers focus on constructing the model by static analysis. S. Yang et al. [10] provided a model called Window Transition Graph, with a more accurate static callback analysis. A recent work [11] constructs Activity Transition Graph with consideration of the launch-mode of each activity to capture transitions more precisely. Note that they built models statically that might miss the changes of GUI screen during run-time.

Some researchers leverage dynamic techniques to construct the model. Amalfitano et al. [5] implemented AnroidRipper to explore the GUI widgets of the app. However, the approach always creates a new state in the model after an event is triggered without calculating the window similarity, which may cause state explosion. Azim et al. [6] statically extracted the Static Activity Transfer Graph of the app, and used dynamic GUI exploration to complement it. However, they regarded the activity as the minimum unit which may cause the loss of accuracy. Besides, all these works pay less attention to the model reusing and specific test generation.

5 CONCLUSION

This paper describes a user-friendly test generation tool for Android apps, which constructs an elaborate LATTE model considering more Android specific characteristics and can be configured for user customized test generation. In the future, we will enrich our LATTE model and enhance our approach by supporting non-crash failures detection through assertion instrumentation.

REFERENCES

[1] Back-Stack. https://developer.android.com/guide/components/activities/tasks-and-back-stack.
[2] F-Droid. https://f-droid.org.
[3] Monkey. http://developer.Android.com/tools/help/monkey.html.
[4] Robotium. http://code.google.com/p/robotium/.
[5] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon. Using GUI ripping for automated testing of Android applications. In *ASE*, pages 258–261, 2012.
[6] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of Android apps. In *OOPSLA*, pages 641–660, 2013.
[7] S. Hao, B. Liu, S. Nath, W. G. J. Halfond, and R. Govindan. PUMA: programmable ui-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*, pages 204–217, 2014.
[8] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: an input generation system for Android apps. In *ESEC/FSE*, pages 224–234, 2013.
[9] J. Yan, T. Wu, J. Yan, and J. Zhang. Widget-sensitive and back-stack-aware GUI exploration for testing android apps. In *QRS*, pages 42–53, 2017.
[10] S. Yang, H. Zhang, H. Wu, Y. Wang, D. Yan, and A. Rountev. Static window transition graphs for Android. In *ASE*, pages 658–668, 2015.
[11] Y. Zhang, Y. Zhang, and J. Xue. Launch-mode-aware context-sensitive activity transition analysis. In *ICSE2018 Accepted*.