2016 IEEE/ACM 38th IEEE International Conference on Software Engineering Companion

FSMdroid: Guided GUI Testing of Android Apps

Ting Su *
Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University, Shanghai, China
tsuletgo@gmail.com

ABSTRACT

GUI testing has been an effective means of validating Android apps. Meanwhile, it still faces a strong challenge about how to explore trails, *i.e.*, unfrequented test sequences, as defects tend to reside on these unfrequented trails. This paper introduces *FSMdroid*, a novel, guided approach to GUI testing of Android apps. The essential idea of *FSMdroid* is to (1) construct an initial stochastic model for the app under test, (2) iteratively mutate the stochastic model and derive tests. The model mutations are guided by an MCMC sampling method such that the resulting test sequences can be diverse and also achieve high code coverage during testing. We have evaluated *FSMdroid* on 40 real-world Android apps. Compared with the traditional model-based testing approaches, *FSMdroid* enhances the diversity of test sequences by 85%, but reduces the number of them by 54%. Furthermore, we uncover 7 app bugs.

1. MOTIVATION AND CONTRIBUTION

Today Android apps have become ubiquitous. Most of them encompass their functional behaviors into GUI interactions. Therefore, before the release of an app, GUI testing is often conducted, in which tests are designed and run in the form of sequences of GUI interaction events.

Many approaches do exist for GUI testing of Android apps [3, 5, 17, 7, 14, 15], which are effective in generating random or common test sequences. However, GUI testing still faces a strong challenge about how to explore trails, *i.e.*, unfrequented test sequences for an app, as these unfrequented trails are usually less covered during testing and defects tend to reside on them.

To tackle this challenge, this paper introduces *FSMdroid*, a model-based approach [13, 18] to GUI testing of Android apps. *FSMdroid* generates test sequences by (1) constructing a stochastic model for an app, and (2) deriving a set of test sequences from that model. Moreover, *FSMdroid* exploits a Markov Chain Monte Carlo (MCMC) sampling algorithm [11, 10] to iteratively mutate the stochastic model, and guide test generation toward yielding high code coverage and exhibiting diverse event sequences. We have implemented *FSMdroid* and evaluated it on 40 real-world Android apps. The results indicate that (1) *FSMdroid* can diversify test sequences, and (2) defects can be revealed on these generated test sequences.

*Ting Su is partially supported by ECNU Program for outstanding doctoral dissertation cultivation No.PY2015032 and China HGJ Project No.2014ZX01038-101-001.

ICSE '16 Companion, May 14 - 22, 2016, Austin, TX,USA

ACM ISBN 978-1-4503-4205-6/16/05.
DOI: http://dx.doi.org/10.1145/2889160.2891043

2. RELATED WORK

When model-based testing is employed to test Android apps, one main activity is to produce an appropriate model representing the GUI interactions. Researchers either manually [19, 14] or automatically [16, 5, 20] construct behavior models for apps. For example, *Android-GUITAR* [1] follows the idea of [16], and uses an *event flow graph*, which is composed of UI events, to describe behaviors of apps. *AndroidRipper* [5] and *ORBIT* [20] use state machines to represent app models. All these models can support model-based GUI testing, while the stochastic model in our study allows the test sequences can be picked in their priorities.

Given a model for an app under test, tests can be derived from the model to test the app. *MobiGUITAR* [6] enforces pair-wise edge coverage to generate tests. *SwiftHand* [12] generates tests when learning models to visit unexplored app states. Usage profiles [9] are also used to generate tests to check commonly-used features. Comparatively, *FSMdroid* iteratively optimizes the generated tests by utilizing the feedback of their code coverage and sequence diversity.

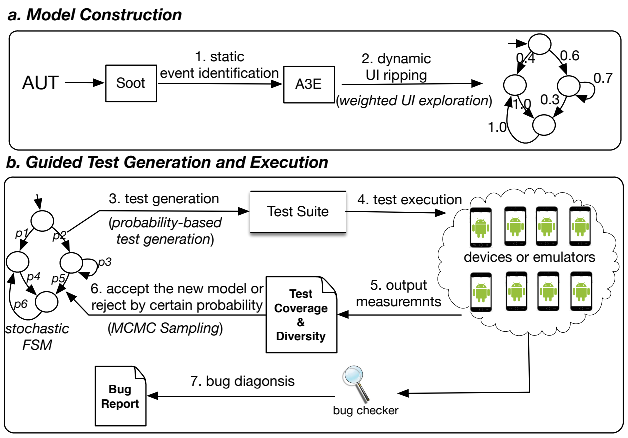
3. APPROACH

As Figure 1 shows, *FSMdroid* takes two main steps to test an app under test: (1) constructing a stochastic model (Figure 1.a); and (2) iteratively mutating the stochastic model, generating test sequences, and running the tests on the app (Figure 1.b). Next presents the key elements.

A weighted UI exploration strategy In our study, an app is represented as a stochastic finite state machine (FSM), where each node represents an app state s characterized by the UI widgets, each transition represents a user event e, and each transition is assigned with a probability value p which indicates its selection probability during test generation.

FSMdroid first uses static analysis to identify UI events which can be missed during dynamic analysis. To efficiently reverse-engineer the model, it adopts a weighted exploration strategy, which integrates three key insights: (1) *prioritizing event selections*, that is, events are assigned with different weights according to the frequencies of event execution, the

Figure 1: Overview of the *FSMdroid* approach.



types of widget, and the numbers of children widgets; (2) *tabuing special events*, that is, some events, *e.g.*, *exit* and bug-triggering events, may accidently close the app and terminate the testing; once triggered, they will be tabued in the future to prevent jeopardizing the efficiency of modeling; (3) *merging duplicate states and transitions* at runtime to render the model compact. The execution profiles (*i.e.*, the execution frequencies of events) from the modeling process are used to populate the stochastic FSM model, where each transition is associated with a probability value.

An MCMC guided test generation process Starting from the initial stochastic FSM, *FSMdroid* utilizes an MCMC sampling method to iteratively mutate the model to guide test generation. As Figure 1.b shows, the test generation process consists of *model mutation*, *test derivation*, and *test execution*. These three steps are iteratively performed until code coverage and sequence diversity reach peak values or the testing budget is used up.

Model mutation At each search iteration, we mutate a stochastic FSM M to M' by randomly picking a state s from M, and then changing all its transition probabilities $p_1, \ldots, p_j$ *w.r.t.* Gaussian distribution, respectively. Note that $p_1 + \ldots + p_j = 1$ always holds during mutations.

Test derivation Given the model M, *FSMdroid* takes a probabilistic-based test generation algorithm to derive test sequences from M, where the events associated with a state are selected according to their transition probabilities: the higher probability is, the higher selection chance it gets.

Test execution The generated test suite T from M is executed on the app under test to determine whether the proposed model M should be accepted or not. It is evaluated on the fitness function F

$$F = \alpha_1 * Coverage(T) + \alpha_2 * Diversity(T)$$

where α_1 and α_2 are weights, *Coverage(·)* measures statement coverage, and *Diversity(·)* approximates the distance difference between test sequences. If the fitness value of M' is higher than that of M, we accept the new model M' and continue on. Otherwise we still accept M' with certain probability to avoid local optimum during the search. More precisely, the new model M' (with the fitness value f') will be accepted with the probability below

$$AcceptProbability(M') = min(1, exp(-\beta * (f - f') - \gamma)$$

Table 1: Testing statistics of our guided GUI testing on 40 Android apps.

Tool(Max Eve.)	Statement Cov.	Consumed Eve.
$Monkey(10000)$	35.5%	3769
$A^3E(500)$	20.8%	36
$FSMdroid_{random}(500)$	31.4%	233
$FSMdroid_{weighted}(500)$	45.1%	109
Model (Test Suite)	Test Suite Size	Test Diversity
$M_{init}(T_{init})$	-28%	+34%
$M_{opt}(T_{opt})$	-54%	+85%

where, β and γ are parameters used to scale the underlying density function and make the search more efficiently.

To improve the scalability of the search, test suites are dynamically allocated and parallelly executed on a distributed testing platform. The system runtime logs are recorded during test execution, which can help analyze app bugs.

4. EVALUATION

We implemented *FSMdroid* on top of Soot [4] (a Java static analysis framework) and A^3E [8], and evaluated it on 40 real-world apps from F-droid [2] (a popular Android apps repository). In particular, we compared the weighted UI exploration strategy ($FSMdroid_{weighted}$) with Android *Monkey* [3] (a random testing approach for Android apps) and A^3E (a systematic GUI ripping approach with the classic dept-first exploration strategy, which is also widely adopted in other GUI rippers [16, 5, 20]). We also implemented random exploration ($FSMdroid_{random}$) in *FSMdroid*.

We also compared the effectiveness of the tests from three models of an AUT: the naive model without transition probabilities (say M_{naive}), the initial stochastic model (say M_{init}), and the optimal stochastic model found in the search (say M_{opt}). Three test suites $T_{naive}, T_{init}, T_{opt}$ are respectively derived from the three models to achieve the same highest coverage by taking the probabilistic-based test generation algorithm in Section 3.

Some evaluation results are summarized in Table 1. By observing the results, we make three findings. First, the weighted exploration strategy can achieve higher code coverage and consume fewer UI events than the other two approaches, which indicates it is more effective and efficient when building app models. In particular, it can improve 10% and 25% statement coverage than those of *Monkey* and A^3E, respectively, but only consumes 3% events of *Monkey*.

Second, the optimized stochastic model can achieve high code coverage more quickly as well as derive more diverse tests. In particular, while achieving the same highest coverage, the test suites T_{init} and T_{opt} are in average 28% and 54% smaller than T_{naive}, respectively. Moreover, T_{init} and T_{opt} in average further improve 34% and 85% test diversity (measured by the event difference between tests) than T_{naive}, respectively. It indicates *FSMdroid* is more effective than traditional model-based testing approaches.

Third, our approach can indeed reveal app bugs on the generated tests. In particular, we have found 7 app bugs, and 3 of them are newly found, the other 4 bugs are independently found by *FSMdroid* and also reported by app users. These bugs include *NullPointerException*, *NumberFormatException*, *IndexOutOfBoundsException*, and *Nonresponding Hang*. Thus we believe *FSMdroid* can enhance the existing GUI testing approaches.

5. REFERENCES

[1] Android guitar. http://sourceforge.net/apps/mediawiki/guitar/index.php?title=Android_GUITAR. Accessed January, 2016.

[2] F-droid. https://f-droid.org/. Accessed January, 2016.

[3] Monkey. http://developer.android.com/tools/help/monkey.html. Accessed January, 2016.

[4] Soot. https://github.com/Sable/soot. Accessed November, 2015.

[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 *IEEE/ACM International Conference on Automated Software Engineering, ASE'12, Essen, Germany, September 3-7, 2012*, pages 258–261, 2012.

[6] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Software*, 32(5):53–59, 2015.

[7] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 59, 2012.

[8] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 641–660, 2013.

[9] P. A. Brooks and A. M. Memon. Automated gui testing guided by usage profiles. In *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, pages 333–342, 2007.

[10] Y. Chen and Z. Su. Guided differential testing of certificate validation in SSL/TLS implementations. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015, Bergamo, Italy, August 30 - September 4, 2015*, pages 793–804, 2015.

[11] S. Chib and E. Greenberg. Understanding the metropolis-hastings algorithm, 1995.

[12] W. Choi, G. C. 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, OOPSLA 2013, Indianapolis, IN, USA, October 26-31, 2013*, pages 623–640, 2013.

[13] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pages 31–36. ACM, 2007.

[14] C. S. Jensen, M. R. Prasad, and A. Møller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, pages 67–77, 2013.

[15] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: segmented evolutionary testing of android apps. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 599–609, 2014.

[16] A. M. Memon, I. Banerjee, and A. Nagarajan. GUI ripping: Reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, WCRE 2003, Victoria, Canada, November 13-16, 2003*, pages 260–269, 2003.

[17] N. Mirzaei, S. Malek, C. S. Pasareanu, N. Esfahani, and R. Mahmood. Testing android apps through symbolic execution. *ACM SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[18] M. Shafique and Y. Labiche. A systematic review of model based testing tool support. *Carleton University, Canada, Tech. Rep. Technical Report SCE-10-04*, 2010.

[19] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based GUI testing of an android application. In *Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011*, pages 377–386, 2011.

[20] W. Yang, M. R. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *Fundamental Approaches to Software Engineering - 16th International Conference, FASE 2013, Rome, Italy, March 16-24, 2013. Proceedings*, pages 250–265, 2013.