FSMdroid: Guided GUI Testing of Android Apps

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

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
types of widget, and the numbers of children widgets; (2) 
tabu ng 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 tabu in 
the future to prevent jeopardizing the efﬁciency of model-
ing; (3) merging duplicate states and transitions at runtime 
to render the model compact. The execution prof les (i.e., 
the execution frequencies of events) from the modeling pro-
cess 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 exe-
cuted on the app under test to determine whether the pro-
posed model \( M' \) should be accepted or not. It is evaluated 
on the ﬁtness function \( F \)

\[
F = \alpha_1 \cdot \text{Coverage}(T) + \alpha_2 \cdot \text{Diversity}(T)
\]

where \( \alpha_1 \) and \( \alpha_2 \) are weights, \( \text{Coverage}(\cdot) \) measures state-
ment coverage, and \( \text{Diversity}(\cdot) \) 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

\[
\text{AcceptProbability}(M') = \min(1, \exp(-\beta \cdot (f - f') - \gamma))
\]

where, \( \beta \) and \( \gamma \) are parameters used to scale the underlying 
density function and make the search more efﬁcient.

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 \(^{A^E} \) [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 
\(^{A^E} \) (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 ran-
dom exploration (FSMdroid_random) in FSMdroid.

We also compared the effectiveness of the tests from three 
models of an AUT: the naive model without transition prob-
abilities (say \( M_{\text{naive}} \)), the initial stochastic model (say \( M_{\text{init}} \)), 
and the optimal stochastic model found in the search (say 
\( M_{\text{opt}} \)). Three test suites \( T_{\text{naive}}, T_{\text{init}}, T_{\text{opt}} \) 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 cov-
verage 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 \(^{A^E} \), 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 \( T_{\text{init}} \) and \( T_{\text{opt}} \) are in average 28% and 
54% smaller than \( T_{\text{naive}} \), respectively. Moreover, \( T_{\text{init}} \) and 
\( T_{\text{opt}} \) in average further improve 34% and 85% test diver-
sity (measured by the event difference between tests) than 
\( T_{\text{naive}} \), 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 in-
dependently 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


