Stoat: Guided, Stochastic Model-Based GUI Testing of Android Apps

Ting Su[1,2], Guozhu Meng[2], Yuting Chen[3], Ke Wu[1], Weiming Yang[1], Yao Yao[1], Geguang Pu[1], Yang Liu[2], Zhendong Su[4]
[1]School of Computer Science and Software Engineering, East China Normal University, China
[2]School of Computer Engineering, Nanyang Technological University, Singapore
[3]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[4]Department of Computer Science, University of California, Davis, USA
suting@ntu.edu.sg,ggpu@sei.ecnu.edu.cn,yangliu@ntu.edu.sg,su@cs.ucdavis.edu

ABSTRACT

Mobile apps are ubiquitous, operate in complex environments and are developed under the time-to-market pressure. Ensuring their correctness and reliability thus becomes an important challenge. This paper introduces *Stoat*, a novel guided approach to perform stochastic model-based testing on Android apps. Stoat operates in two phases: (1) Given an app as input, it uses dynamic analysis enhanced by a weighted UI exploration strategy and static analysis to reverse engineer a stochastic model of the app's GUI interactions; and (2) it adapts Gibbs sampling to iteratively mutate/refine the stochastic model and guides test generation from the mutated models toward achieving high code and model coverage and exhibiting diverse sequences. During testing, system-level events are randomly injected to further enhance the testing effectiveness.

Stoat[1] was evaluated on 93 open-source apps. The results show (1) the models produced by Stoat cover 17~31% more code than those by existing modeling tools; (2) Stoat detects 3X more unique crashes than two state-of-the-art testing tools, Monkey and Sapienz. Furthermore, Stoat tested 1661 most popular Google Play apps, and detected 2110 previously unknown and unique crashes (including several crashes in such apps with billions of installations as Wechat, Gmail and Google+). So far, 50 developers have responded that they are investigating our reports. 30 of reported crashes have been confirmed or fixed.

1 INTRODUCTION

Mobile apps have become ubiquitous and drastically increased in number over the recent years. As recent statistics [9] shows, over 50K new Android apps are submitted to Google Play each month. However, it is challenging to guarantee their quality. First, they are event-centric programs with rich graphical user interfaces (GUIs), and have complex environment interplay (*e.g.*, with users, devices, and other apps). Second, they are typically developed under the time-to-market pressure, thus may be inadequately tested before releases. When performing testing, developers tend to exercise those functionalities or usage scenarios that they believe to be important, but may miss bugs that their designed tests fail to expose.

To tackle this challenge, we developed a guided, stochastic model-based testing approach, *Stoat* (STOchastic model App Tester)[2], to improve GUI testing of Android apps. It aims to thoroughly test the functionalities of an app from the GUI model, and validate the app's behavior by enforcing various user/system interactions [20]. Given an app as input, Stoat operates in two phases. First, it generates a stochastic model from the app to describe its GUI interactions. In our setting, a stochastic model for an app is a finite state machine (FSM) whose edges are associated with probabilities for test generation. In particular, Stoat takes a dynamic analysis technique, enhanced by a weighted UI exploration strategy and static analysis, to explore the app's behaviors and construct the stochastic model.

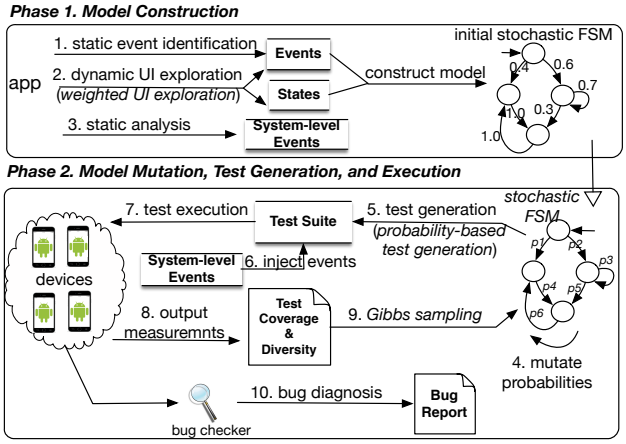
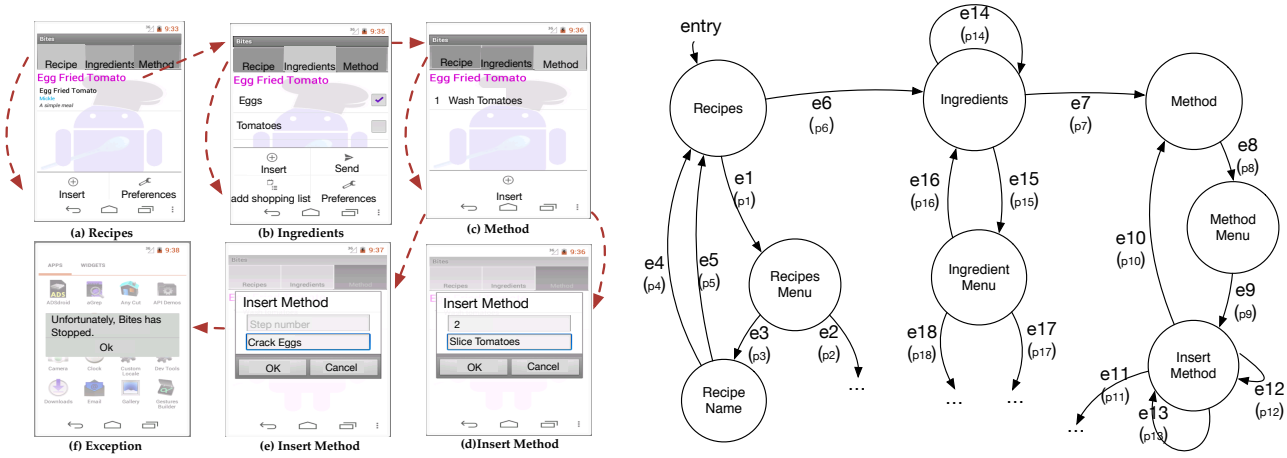


Figure 1: Stoat's workflow.

Second, Stoat iteratively mutates the stochastic model and generates tests from the model mutants. By perturbing the probabilities, Stoat is able to generate tests with various compositions of events to sufficiently test the GUI interactions, and purposely steers testing toward less travelled paths to detect deep bugs. In particular, Stoat takes a guided search algorithm, inspired by Markov Chain Monte Carlo (MCMC) sampling, to search for "good" models (explained in Section 2) — the derived tests are expected to be diverse, as well as achieve high code and model coverage.

Moreover, Stoat adopts a simple yet effective strategy to enhance MBT: randomly inject various system-level events into UI tests during MCMC sampling. It avoids the complexity of incorporating system-level events into the behavior model, and further imposes the influence from outside environment to detect intricate bugs.

[1]Stoat and its demo video are available at https://tingsu.github.io/files/stoat.html. This paper is presented at the Research Tool Competition track in NASAC 2017 (held by CCF, China Computer Federation), Harbin, China.
[2]This work has been published in [15, 16].



(a) Screenshots of a cookbook app *Bites*.

(b) App model of *Bites*.

Figure 2: Example app *Bites* and its app model.

2 APPROACH OVERVIEW

Stoat operates in a unique two-phase process to test an app. Figure 1 shows its high-level workflow.

Phase 1: Model construction. Stoat first constructs a stochastic Finite State Machine (FSM) to describe the app's behaviors, where each node represents an app state (abstracted as an app page, see details in Section 3), and each transition denotes an input event. It uses a dynamic analysis technique, enhanced by a *weighted UI exploration strategy* (step 2 in Figure 1), to efficiently explore an app's GUIs. It infers the input events by (1) analyzing the GUI layout information of each app page; and (2) using a static analysis (step 1) to scan the events that are programmed in the app code, which may be missed in (1). To efficiently construct the app model, Stoat dynamically prioritizes the executions of these events to cover as many app behaviors as possible. In detail, Stoat uses three key elements to improve the UI exploration performance, which constitute the basis of the weighted UI exploration strategy:

- *Frequency of Event Execution.* All events are given opportunities to be executed. The less frequently an event is executed, the more likely it will be selected during subsequent exploration.
- *Type of Events.* Different types of events are not equally selected. For instance, compared with normal UI events (*e.g.*, `click`), navigation events (*e.g.*, `back`, `scroll`, and `menu`), are given different priorities to ensure they are triggered at right timing, otherwise they may drastically undermine the exploration efficiency.
- *Number of Unvisited Children Widgets.* If an event solicits more new UI widgets on the next page, it will be prioritized since more efforts should be spent on pages with new functionalities.

During exploration, Stoat records the execution frequencies of all events, and later uses them to compute the initial probability values of the transitions in the model.

Phase 2: Model mutation, test generation, and execution. To thoroughly test an app, Stoat leverages the model from *Phase 1* to iteratively mutate the transitions' probabilities and guide test generation. In detail, it exploits *Gibbs Sampling* [19], one of Markov Chain Monte Carlo (MCMC) methods [4], to guide the GUI testing. In our setting, we intend to find "good" model samples, from which the test suites can achieve our desired goal, *i.e.*, achieving *high coverage* and containing *diverse event sequences*. Since such test suites are expected to trigger more program states and behaviors, and thus increase the chance of detecting bugs.

Typically, Stoat works as a loop: randomly mutate the transition probabilities of the current stochastic model (step 4), generate the tests from the model *w.r.t.* the probabilities (step 5), randomly inject system-level events (analyzed by static analysis in step 3) into these UI-level tests to enhance MBT (step 6), replay them on the app (step 7) and collect test results, such as *code* and *model coverage* and *event sequence diversities* (step 8).

Informed by the test results, Stoat exploits Gibbs sampling to decide whether the newly proposed model should be accepted or rejected (step 9), the model with better objective value (it implies it can generate better test suites) will be accepted for the next iteration of mutations and samplings; otherwise, it will be rejected with certain probability to avoid local optimal (if rejected, the original model will be reused). Once any bug is detected (*i.e.*, crash or non-responding), further analysis will be performed to diagnose the bug with the corresponding test (step 10).

3 AN ILLUSTRATIVE EXAMPLE

Bites [6] is a simple cookbook app (shown in Figure 2a) that supports recipe creation and sharing. A user can create a recipe by clicking the *insert* menu item in the *Recipes* page (page a). When the user taps the name of a recipe, the app navigates to the *Ingredients* page (page b), where he/she can view or add ingredients, share them via SMS, or add them into a shopping list. The user can also switch to the *Method* page (page c), where the cooking methods can be viewed. By clicking the *insert* menu item, the user can fill in *Step number* and *Method* (page d).

Model construction. Figure 2b shows a part of the constructed behavior model of *Bites*, where each node denotes an app state s and each transition an input event e (associated with a probability value p). Stoat drives the app from one state s to another state s' by emitting an input event e (*e.g.*, `click`, `edit`). For example, Stoat presses the *menu* key on the *Recipes* page, a menu will pop up, and a

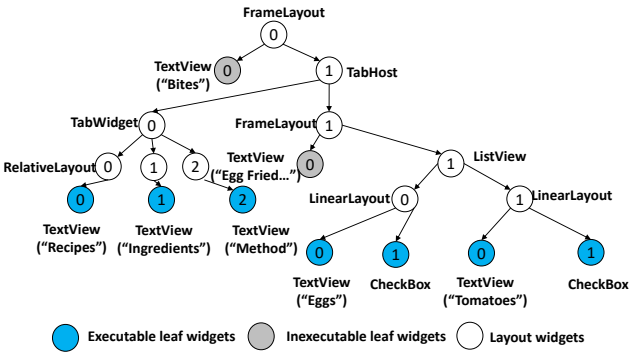


Figure 3: View hierarchy of the *Ingredient* page in *Bites*.

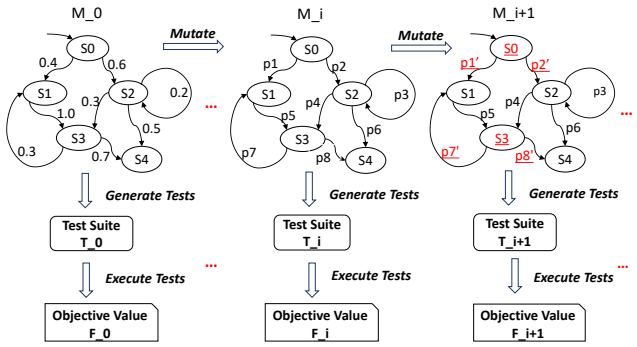


Figure 4: Gibbs sampling-guided model mutation and test generation.

new app state (corresponding to *Recipes Menu*) is created. By doing in this way, Stoat constructs the app model by connecting the states and the events. For example, *Recipes* can be navigated to *Ingredients* when e_6 occurs (*i.e.*, *click* a recipe item on the *Recipes* page) with the probability p_6.

Here, an app state s is abstracted as an app page, which is represented as a widget hierarchy tree. Figure 3 shows such a tree of the *Ingredient* page, where the non-leaf nodes in white denote layout widgets (e.g., `LinearLayout`), the leaf nodes in blue denote (executable) widgets (e.g., `Button`). Each node has its own index value and a description of its widget type and other properties (*e.g.*, text, clickable, long-clickable, scrollable, *etc.*). These information can be used to encode an page as a string to differ app states. When a page's structure (and properties) changes, a new state is created. For example, in Figure 2a, the *Recipes* page and the *Ingredients* page correspond to two app states since they have different view hierarchy trees. If the app exits/crashes, the ending state is treated as a final state (*e.g.*, page f).

A probability value p is assigned to each transition e, denoting the selection weight of e in test generation. The initial probability values are determined by the execution frequency of each event during model construction — p is initially assigned the ratio of e's observed execution times over the total execution times of all events *w.r.t.* s ($e \in s$). For example, if e_1 and e_6 have been executed 4 and 6 times during model construction, their initial probability values, *i.e.*, p_1 and p_6, will be 0.4 and 0.6.

Guided, Stochastic Model-based Testing. Stoat adopts the Gibbs sampling technique to guide model mutation and test generation. Figure 4 illustrates this procedure. Starting from the initial model M_0, Stoat works as follows: assume the model M_i is the current model, to generate a new model mutant M_{i+1}, Stoat randomly decides which states should be mutated. Assume Stoat chooses the two states s_0 and s_3 to mutate, take s_0 as an example, it will mutate its two transition probabilities p_1 and p_2 to p_1' and p_2' by randomly increasing or decreasing them by Δ (Δ is a constant between 0 and 1, *e.g.*, 0.1), but ensure $p_1' + p_2' = 1$ holds.

After the mutation, Stoat generates a test suite T_{i+1} from the new model M_{i+1} by following a probabilistic strategy: It starts from the entry state s_0, and follows the probability values to select an event from the corresponding app state until the maximum sequence length or the ending state is reached. The higher the event probability value is, the more likely the event will be selected. For example, if p_1' is 0.4 and p_2' is 0.6, Stoat will choose e_1 with 40% while e_2 with 60%. To decide whether M_{i+1} is a better model, Stoat executes T_{i+1} on the app, and collect the objective value F_{i+1} (the results of code and model coverage and test diversity). If F_{i+1} is higher than F_i from M_i, M_{i+1} will be accepted and M_i will be discarded, and the next mutation will start on M_{i+1}. Otherwise, $M_i + 1$ will be accepted with the probability of F_{i+1}/F_i, if rejected, M_i will be reused. By continuing this working loop, the test suites can be optimized towards our desired goal.

During this procedure, Stoat also randomly injects system-level events into UI-level tests from the model. For example, *Bites* can be activated by SMS and Browser to read the recipes shared by others. During testing, Stoat simulates these system-level events by sending specific Broadcast Intents to *Bites*.

In *Bites*, Stoat finally found four unique crashes, two are detected in model construction, and the other two in guided test optimization. One is `NumberFormat`, two are `CursorIndexOutOfBounds`, and the last one is `NullPointer` exception.

4 TOOL IMPLEMENTATION

Stoat is implemented as a fully automated app testing framework, which reuses and extends several tools: Android `UI Automator` [8, 12] and Android Debug Bridge (`ADB`) for automating test execution; Soot [7] for static analysis to identify potential input events; Androguard [17] for analyzing the system-level events that the apps are particularly interested in.

Currently, Stoat supports three sources of system-level events: (1) 5 user actions (*i.e.*, screen rotation, volume control, phone calls, SMSs, app switch); (2) 113 system-wide broadcast intents (*e.g.*, battery level change, connection to network) to simulate system messages; (3) the events that the apps are particularly interested in, which are usually declared by the tags `<intent-filter>` and `<service>` in their `AndroidManifest.xml` files.

Stoat supports `click`, `touch`, `edit` (generate random texts of numbers or letters), and `navigation` (*e.g.*, `back`, `scroll`, `menu`) for model construction. During Gibbs sampling, Stoat generates a test suite with the maximum size of 30 tests and each with a maximum length of 20 events at each sampling iteration. Stoat instruments open-source apps by Emma [14] to collect line coverage; closed-source apps by Ella [2] to collect method coverage; and extends cosine similarity [18] to compute the diversity of a test suite. To improve scalability, Stoat is designed as a server-client mode, where the server can parallelly control multiple Android devices.

5 EVALUATION RESULTS

Stoat runs on a 64-bit Ubuntu 14.04 physical machine with 12 cores (3.50GHz Intel Xeon(R) CPU) and 32GB RAM, and uses Android emulators to run tests. Each emulator is configured with 2GB RAM and X86 ABI image (KVM powered), and the KitKat version (SDK 4.4.2, API level 19). To evaluate effectiveness, we run Stoat on 93 open-source Android apps from F-droid [10], and compare it with the state-of-the-arts in terms of model construction, code coverage and fault detection.

Model Construction. Stoat is compared with MobiGUITAR [1] and PUMA [11]. Both tools produce similar FSM models. We run each tool on one emulator, and test each app for 1 hour, and measure the *code coverage* to approximate the completeness of the constructed models, which is the basis of model-based testing. We also record the number of states and edges in the models to measure the complexity. Intuitively, the higher the code coverage, the more compact the model is, the more effective the tool is.

Results: Stoat is more effective than MobiGUITAR and PUMA. On average, it covers 17~31% more code than MobiGUITAR, and 23% more than PUMA. The models produced by Stoat are more compact without state explosion, which is more effective for Gibbs sampling.

Code Coverage and Fault Detection. We compare Stoat with these GUI testing tools: (1) Monkey (random fuzzing), (2) A^3E [3] (systematic UI exploration), and (3) Sapienz (genetic algorithm) [13]. They have the best performance in their own approach categories [5]. Specifically, Monkey emits a stream of random input events, including both UI and system-level events, to maximize code coverage. A^3E systematically explores app pages and emits events by a depth-first strategy, which is also widely adopted in other GUI testing tools. Sapienz uses Monkey to generate the initial test population, and adapts genetic algorithms to optimize the tests to maximize code coverage while minimizing test lengths. We allocate 3 hours for each tool to thoroughly test each app on one single emulator.

Results: Stoat is more effective than Monkey, A^3E and Sapienz in code coverage and fault detection. On average, A^3E, Monkey, Sapienz, and Stoat achieve 25%, 52%, 51%, and 60% line coverage, respectively. In detail, Stoat achieves nearly 35% higher coverage than A^3E. Stoat has detected 68 buggy apps with 249 unique crashes, which is much more effective than A^3E (8 crashes), Monkey (76 crashes), and Sapienz (87 crashes). Stoat has detected all the crashes that were found by A^3E. We find the crashes detected by Stoat have much less overlap with Monkey and Sapienz — Stoat detected exclusive 227 and 224 crashes than Monkey and Sapienz, respectively.

To further evaluate the usability and effectiveness of Stoat in testing real-world apps, we run Stoat on 1661 most popular commercial apps from Google Play with various categories.

Usability and Effectiveness. Stoat was run on 3 physical machines with 18 emulators and 6 phones (allocate 3 hours per app). It takes nearly one month to test all these apps, and detected 2110 unique previously-unknown crashes from 691 apps: 452 crashes from model construction, 1927 crashes from Gibbs sampling, and 269 crashes are detected in both phases. We have sent all the bug reports to the developers. So far, 50 developers have replied that they are investigating our reports (excluding auto-replies). 30 of the reported crashes have been confirmed or fixed. In particular, Stoat detected total 4 crash bugs in WeChat, Gmail, and Google+ that affect billions of users — all of these bugs have been confirmed and fixed.

6 CONCLUSION

We have introduced Stoat, a novel, automated model-based testing approach to improving GUI testing. Stoat leverages the behavior models of apps to iteratively refine test generation toward high coverage as well as diverse event sequences. Our evaluation results on large sets of apps show that Stoat is more effective than state-of-the-arts. We believe that Stoat's high-level approach is general and can be fruitfully applied in other testing domains.

REFERENCES

[1] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. 2015. MobiGUITAR: Automated Model-Based Testing of Mobile Apps. *IEEE Software* 32, 5 (2015), 53–59. DOI: http://dx.doi.org/10.1109/MS.2014.55

[2] Saswat Anand. 2017. ELLA. (2017). Retrieved 2017-2-18 from https://github.com/saswatanand/ella

[3] Tanzirul Azim and Iulian Neamtiu. 2013. 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*. 641–660.

[4] Siddhartha Chib and Edward Greenberg. 1995. Understanding the Metropolis-Hastings Algorithm. (1995).

[5] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet? (E). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*. 429–440. DOI: http://dx.doi.org/10.1109/ASE.2015.89

[6] Bites Developers. 2017. Bites. (2017). Retrieved 2017-2-18 from https://code.google.com/archive/p/bites-android/

[7] Soot Developers. 2017. Soot. (2017). Retrieved 2017-2-18 from https://github.com/Sable/soot

[8] Google. 2017. Android UI Automator. (2017). Retrieved 2017-2-18 from http://developer.android.com/tools/help/uiautomator/index.html

[9] AppBrain Group. 2017. AppBrain. (2017). Retrieved 2017-2-18 from http://www.appbrain.com/stats/

[10] F-droid Group. 2017. F-Droid. (2017). Retrieved 2017-2-18 from https://f-droid.org/

[11] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. 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 (MobiSys '14)*. ACM, New York, NY, USA, 204–217. DOI: http://dx.doi.org/10.1145/2594368.2594390

[12] Xiaocong He. 2017. Python wrapper of Android UIAutomator test tool. (2017). Retrieved 2017-2-18 from https://github.com/xiaocong/uiautomator

[13] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 94–105.

[14] Vlad Roubtsov. 2017. EMMA. (2017). Retrieved 2017-2-18 from http://emma.sourceforge.net/

[15] Ting Su. 2016. FSMdroid: Guided GUI Testing of Android Apps. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016 - Companion Volume*. 689–691.

[16] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, Stochastic Model-based GUI Testing of Android Apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*. ACM, New York, NY, USA, 245–256.

[17] Androguard Team. 2017. Androguard. (2017). Retrieved 2017-2-18 from https://github.com/androguard/androguard

[18] Wikipedia. 2017. Cosine similarity. (2017). Retrieved 2017-2-18 from https://en.wikipedia.org/wiki/Cosine_similarity

[19] Wikipedia. 2017. Gibbs Sampling. (2017). Retrieved 2017-2-18 from https://en.wikipedia.org/wiki/Gibbs_sampling

[20] Qing Xie and Atif M. Memon. 2006. Studying the Characteristics of a "Good" GUI Test Suite. In *17th International Symposium on Software Reliability Engineering (ISSRE 2006), 7-10 November 2006, Raleigh, North Carolina, USA*. 159–168. DOI: http://dx.doi.org/10.1109/ISSRE.2006.45