Fastbot2: Reusable Automated Model-based GUI Testing for Android Enhanced by Reinforcement Learning

Zhengwei Lv
lvzhengwei.m@bytedance.com
Bytedance
Beijing, China

Chao Peng∗
pengchao.x@bytedance.com
Bytedance
Beijing, China

Zhao Zhang
zhangzhao.a@bytedance.com
Bytedance
Beijing, China

Ting Su†
tsu@sei.ecnu.edu.cn
East China Normal University
Shanghai, China

Kai Liu‡
liukai.0914@bytedance.com
East China Normal University /
ByteDance, Beijing, China

Ping Yang
yangping.cser@bytedance.com
Bytedance
Beijing, China

ABSTRACT
We introduce a reusable automated model-based GUI testing technique for Android apps to accelerate the testing cycle. Our key insight is that the knowledge of event-activity transitions from the previous testing runs, i.e., executing which events can reach which activities, is valuable for guiding the follow-up testing runs to quickly cover major app functionalities. To this end, we propose (1) a probabilistic model to memorize and leverage this knowledge during testing, and (2) design a model-based guided testing strategy (enhanced by a reinforcement learning algorithm). We implemented our technique as an automated testing tool named Fastbot2. The evaluation on two popular industrial apps (with billions of user installations), Douyin and Toutiao, shows that Fastbot2 outperforms the state-of-the-art testing tools (Monkey, Ape and Stoat) in both activity coverage and fault detection in the context of continuous testing. To date, Fastbot2 has been deployed in the CI pipeline at ByteDance for nearly two years, and 50.8% of the developer-fixed crash bugs were reported by Fastbot2, which significantly improves app quality. Fastbot2 has been made publicly available to benefit the community at: https://github.com/bytedance/Fastbot_Android.

CCS CONCEPTS
• Software and its engineering → Software testing and debugging; • Computing methodologies → Reinforcement learning.

ACM Reference Format:

∗Chao Peng and Ting Su are the corresponding authors.
†Kai Liu was a research intern at ByteDance when this work was conducted.
‡Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

1 INTRODUCTION
Mobile apps have drastically increased in number over the recent years [1]. Ensuring app quality is crucial to keeping user loyalty and maintaining business success. To this end, automated GUI testing has become an attractive and cost-effective solution [2, 10, 11].

In practice, an industrial app undergoes frequent updates to catch up with the changing user demands. At ByteDance, we release new updates of major apps on a weekly basis. As a result, continuous testing becomes crucial for quick feedback on app quality (e.g., doing smoke testing) whenever a new internal version is built. However, simply adopting existing testing tools [10], although feasible, is inefficient and ineffective, as they simply rerun each version from scratch and do not leverage the knowledge from previous testing runs to accelerate GUI testing in the current run.

To fill this important gap, we introduce a reusable automated GUI testing technique. Our key idea is to leverage model-based testing (MBT). Among existing testing solutions, MBT is recognized for its unique model construction phase, which is ideal for storing and leveraging the prior knowledge. However, we face two major technical challenges in putting our idea into practice.

The first challenge is how to effectively store the knowledge from previous testing runs. Our key insight is that the knowledge of event-activity transitions, i.e., executing which events reaches which activities, is valuable for guiding the follow-up testing to quickly cover core activities. Thus, we propose a probabilistic model as the basis of MBT to memorize such knowledge from each testing run. This model stores a set of event-activity transitions, each of which records the historical probability of an event to reach an activity. Moreover, to tackle the complexity of industrial apps, we introduce a conception of hyper-event to represent events in this model, which is useful to balance the model scalability and accuracy.

The second challenge is how to effectively leverage the prior knowledge to guide GUI testing. Classic MBT methods traverse the model to generate GUI events (i.e., GUI tests). However, one prominent problem is that such GUI tests are likely to be broken due to the unawareness of the connectivity between different GUI events. To overcome this issue, our key insight is to employ the probabilistic model to achieve on-the-fly, guided model-based testing. Specifically, the probabilistic model (which stores the knowledge of event-activity transitions) provides one-step guidance about which events on the current GUI page could be selected to quickly reach
those not-yet-covered activities in the current testing run. Moreover, to further improve performance, we develop a reinforcement learning algorithm to provide multi-step guidance (also informed by the probabilistic model), which aims to reach those deep activities requiring executing multiple sequential events.

We implemented our technique as an automated testing tool named Fastbot2. Our evaluation on our two popular industrial apps, Douyin and Toutiao, shows that Fastbot2 outperforms the two state-of-the-art MBT tools, Stoat [9] and Ape [5], and the random testing tool Monkey [4] in both activity coverage and bug finding in the context of continuous testing. To sum up, our work makes two major contributions: (1) We propose a reusable automated model-based GUI testing technique enhanced by reinforcement learning to satisfy the practical needs of continuous testing, which has not been considered by prior work. (2) Our implementation, Fastbot2, outperforms the state-of-the-art. It has also been successfully deployed in the CI pipeline at ByteDance and received positive feedback on its ability of improving app quality.

2 FASTBOT2

Figure 1 shows Fastbot2’s workflow. Fastbot2 takes as input an APK file of the app, outputs coverage and crash reports. Fastbot2 includes two major phases. The first phase does setup before testing: compiling the APK to gather static widget text labels of (Step “a1”), installing the app on a pool of mobile devices (Step “a2”), and loading the historical data if available to populate the probabilistic model (Step “a3”, cf. Section 2.2.1). The second phase does guided GUI exploration (cf. Section 2.3). Fastbot2 dumps the current GUI page from the app (Step “b1”), identifies and abstracts available hyper-events from the current page (Step “b2”, cf. Section 2.2.2), selects the event which is likely to increase activity coverage (Step “b3”), executes this UI event (Step “b4”) and updates the historical testing data, the probabilistic model (Step “b5”) and the reinforcement learning agent (Step “b6”). These steps (“b1”∼“b6”) will be iteratively conducted until the time budget is used up.

2.1 An Illustrative Example

Figure 2(a) gives an illustrative example taken from Toutiao, a popular daily news app. We use this example to ease the exposition of our approach in Sections 2.2 and 2.3. In Toutiao, users can view news on Activity 1, click the news title (e.g., “Title 1”) to reach Activity 2 to view the content, click the news author (“Author”) to view all news from this author on Activity 3 and watch the live video on Activity 4 if available by clicking the profile photo.

2.2 Probabilistic Model and Hyper-events

2.2.1 Probabilistic Model. We propose a probabilistic model as the basis of MBT to memorize the knowledge of event-activity transitions from previous testing runs. Specifically, this model $M$ is formally defined as a 3-tuple $M = (E, A, \delta)$, where

- $E$ is the set of hyper-events created from UI widgets.
- $A$ is the set of activities of the app under test.
- $\delta$ is the transition function, i.e., $E \rightarrow \mathcal{P}(A \times [0, 1])$. $\mathcal{P}$ is a powerset function and each transition is of the form $e \rightarrow (A, p)$, meaning that the probability of a hyper-event $e$ reaching an app activity $A$ is $p$, where $e \in E$ and $A \in A$.

The probabilistic model $M$ is constructed from historical exploration data. The probability of reaching activity $A_i$ by executing the hyper-event $e$ (denoted by $P(e, A_i)$) is calculated by:
where \( N(e, A_i) \) denotes the number of times of \( e \) reaching \( A_i \), and \( N(e) \) denotes the total execution times of \( e \) in all the previous testing runs. If the hyper-event \( e \) can reach \( k \) activities (e.g., \( A_1, \ldots, A_k \)), \( \sum_{i=1}^{k} P(e, A_i) = 1 \) holds.

**Example.** Figure 2(b) gives an example of the initial probabilistic model (see the left part) loaded from previous testing runs before starting the current testing run. For example, Activity 2 can be reached by executing the hyper-event \( e_1 \) on Activity 1, \( e_1 \) can reach Activity 2 and 5 with probability values 60% and 40%, respectively.

### 2.2.2 Hyper-events

We propose the concept of hyper-event to represent the events in the probabilistic model. A hyper-event is created from each UI widget according to its properties. Specifically, we only consider the following four properties of a widget: the activity which the widget belongs to, the widget’s text\(^1\), resource-id, and the supported action types (e.g., click, long click). In other words, if some widgets have the same four properties, we assume they have the similar functionality and only one hyper-event will be created. We ignore all the other minor widget properties (e.g., a widget’s type) when creating hyper-events with the aim of balancing between model scalability and accuracy.

**Example.** In Figure 2(a), on Activity 1, Fastbot2 will create only one hyper-event "e1" for the widgets named “Title 1” and “Title 2” because these two widgets have the identical widget properties: the same activity, the same empty text (“Title 1” and “Title 2” are the texts dynamically loaded from the app server without static text labels), the same resource-id, and the same click action type. In this way, we will create three hyper-events, i.e., “e1” (representing “Title 1” and “Title 2”), “e2” (representing “Author 1” and “Author 2”), “e3” (representing “Remarks 1” and “Remarks 2”), on Activity 1.

### 2.3 Model-based Guided UI Exploration

The key idea of Fastbot2 is to reuse the prior knowledge stored in the probabilistic model to effectively guide GUI testing. To achieve this, the key step is to decide which UI event on the current GUI page should be selected so as to quickly increase activity coverage. This step corresponds to Step “b3” in Figure 1. Specifically, given a probabilistic model \( M \), Fastbot2 can be the set of already covered activities in the current testing run and \( \mathcal{E}_k \) be the set of hyper-events from the current GUI page. The expectation of improving activity coverage by executing \( e_i \) \( e_i \in \mathcal{E}_k \) can be computed as \( \mathbb{E}(e_i) = \sum \mathbb{E}(e_i, A) \) for \( 0 \leq i \leq |\mathcal{E}_k| \).

Here, \( \mathbb{E}(e_i) \) represents the expectation value of probability that those not-yet-covered activities in the current testing run will be covered after the hyper-event \( e_i \) is executed. The higher \( \mathbb{E}(e_i) \), the more likely it is to improve activity coverage. Thus, Fastbot2 selects the hyper-event \( e_i \) by probability \( P_M(e_i) \):

\[
P_M(e_i) = \exp \left( \frac{\mathbb{E}(e_i)}{\alpha} \right) / \sum_{e_j \in \mathcal{E}_k} \exp \left( \frac{\mathbb{E}(e_j)}{\alpha} \right)
\]

where, \( \alpha \) is a hyperparameter which adjusts the randomness of this mode. This equation is adapted from the softmax formula. We also require that \( e_i \) should be selected no more than \( K \) times to ensure fairness. In practice, we set \( \alpha \) as 0.8 and 0.3 as 1. By using the probabilistic model as prior information, the model exploitation mode can quickly improve activity coverage in a short time.

**Example.** In Figure 2(a), three hyper-events are available on Activity 1. Since all these three events have been included in the probabilistic model \( M \) (see the left part of Figure 2(b)), Fastbot2 activates the model exploitation mode to select events. According to \( M \), event \( e_1 \) and \( e_2 \) are more likely to reach unexplored activities (i.e., Activity 2, 3, 5), while event \( e_3 \) has 90% probability to stay in Activity 1. Thus, Fastbot2 is likely to select \( e_1 \) or \( e_2 \). Assume \( e_1 \) is selected and then Activity 2 is covered. In Activity 2, event \( e_4 \) (the back button) has 100% probability to return back to Activity 1, while event \( e_5 \) and \( e_6 \) have not been included in \( M \). As this time, Fastbot2 activates the model expansion mode and randomly selects \( e_5 \) or \( e_6 \). Assume \( e_5 \) is selected and then Activity 3 is covered. Meanwhile, \( M \) is updated by adding \( e_5 \rightarrow \) Activity 3 with probability value 100% (see the right part of Figure 2(b)).

#### 2.3.2 Learning-based event selection

However, the probabilistic model can only express one-step guidance information. Fortunately, reinforcement learning technique is able to spread one-step into multiple-step guidance information.

**Q-table expansion.** The key component of the RL agent is the Q-table, which contains the Q-values (which indicate the possibility of executing each hyper-event to reach a new activity). During testing, no matter which event selection strategy is used, the Q-value of the selected hyper-event \( e_i \) on the current GUI page, i.e., \( Q(e_i) \), is updated to \( Q(e_i) + \alpha(\mathcal{G}_{t+1,n} - Q(e_i)) \), where \( \mathcal{G}_{t+1,n} \) is the n-step cumulative reward calculated by an N-step Sarsa method [3]:

\[
\mathcal{G}_{t+1,n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^n Q(e_{t+n})
\]

Here, \( \gamma \) is the discount factor. \( r_{t+1} \) is the immediate reward earned after the event \( e_1 \) is executed, which is defined as

---

\(^1\)Here, the text means the static text labels stored in the resource files of the APK file. If the text is dynamically loaded from the app server, we treat its text as empty.

\(^2\)After we decide which hyper-event should be selected, if the selected hyper-event represents multiple UI widgets, we will randomly pick one UI widget to exercise.
We evaluate the effectiveness of Fastbot2 to test one version on 10 devices in parallel for 1 hour, and test the next version by reusing historical data from all previous runs. For the other tools, we ran them on each version on 10 devices in parallel as this is the typical usage scenario of these tools. We compare the achieved activity coverage and the number of uncovered unique crashes by each tool.

RQ2: Ablation Study. Do model-based and learning-based event selection strategies both contribute to Fastbot2’s overall performance?

We test Douyin (v.19.7) and Toutiao (v.8.7.0) on 10 devices for 1 hour with the model-based strategy enabled only, the learning-based strategy enabled only, and both strategies enabled to evaluate their respective impact on Fastbot2’s overall performance.

In RQ1 and RQ2, 10 different Huawei, OPPO and Google Pixel Android devices are used to mitigate the device fragmentation issue.

3.1 RQ1: Test Effectiveness

Figures 3(a) and 3(b) show activity coverage achieved: bars give numbers of activities covered by different tools on each app version, and curves give accumulated numbers of activities covered after testing ten consecutive versions. Figures 3(c) and 3(d) give similar information on the number of unique crashes revealed by these tools (we deduplicate crashes according to stack traces [9]).

We can see that Fastbot2 achieved the highest activity coverage on each single version of both apps (except Toutiao’s v8.7.3) and the highest accumulated activity coverage across ten continuous versions for both apps. It indicates that reusing the knowledge from previous testing runs can effectively improve activity coverage within the same time budget. On the other hand, We can see that Fastbot2 uncovered many more crashes on Douyin than all other tools (all the crashes were confirmed as real bugs). Fastbot2 uncovered 2 fewer crashes than Ape on Toutiao. We find that most crashes found by these two tools on Toutiao are similar native crashes (i.e., crashes triggered by the app but reside in native C++ libraries). The overall result indicates that reusing the knowledge from previous testing runs can also effectively improve Fastbot2’s bug finding ability.

Figures 4(a) and (b) use venn diagrams to compare the differences between Fastbot2 and other tools in terms of the accumulated activity coverage after testing ten app versions. The overlapped part denotes the number of activities covered by both tools, while other parts denote the number of activities covered by the two tools alone, respectively. We can see that Fastbot2 is indeed effective as it can cover many more unique activities than other tools.

We note that Fastbot2 performed much better on Douyin than Toutiao, compared to Ape. This is because Douyin is much more complicated than Toutiao. For example, Douyin has more complicated features, e.g., online shopping in the livestreaming room, video recording and editing. In contrast, Toutiao has simpler features (e.g., news reading). As a result, Toutiao is more likely to reach saturated coverage within 1-hour testing by Fastbot2 and Ape.

We also note that Stoot did not perform well as we expected. After inspection, we find Stoot only generated around 300 events during 1-hour testing. The main reason is that Stoot kept querying the GUI tree and only generated the next event until the current GUI page became stable. However, many features of Douyin and Toutiao, e.g., advertisements, profiles and user comments, are dynamically changing, which makes Stoot waste a lot of time for waiting.
We focus on discussing the industrial practice of automated Android GUI testing. Facebook Sapienz [6] adopts search-based testing to improve code coverage and fault detection. They also extract information from crowd-based testing to enhance Sapienz [7]. WeChat WCTester [12, 13] adopts Monkey-based random testing. WCTester allows human testers to specify blacklisted widgets and define GUI event sequences to improve coverage. However, our work is significantly different from the prior work. First, FASTBOT2 mainly adopts model-based testing (enhanced by a learning-based algorithm). Second, FASTBOT2 reuses the knowledge from the historical exploration data to resolve the practical needs of continuous testing, which have not been considered by prior work.

6 CONCLUSION

This paper presents a reusable automated model-based GUI testing technique for Android enhanced by reinforcement learning to satisfy the practical needs of continuous testing. Our implementation FASTBOT2 outperforms the three state-of-the-art testing tools in both activity coverage and bug finding in the scenario of continuous testing on two popular apps Douyin and Toutiao. FASTBOT2 has been successfully deployed in the CI pipeline at ByteDance and received positive feedback on its ability of improving app quality.

REFERENCES


4 INDUSTRIAL DEPLOYMENT

To date, FASTBOT2 have been deployed in the Continuous Integration pipeline at ByteDance for nearly two years. FASTBOT2 is automatically triggered by nightly builds to obtain quick feedback on app quality when new code changes occur. We have received positive feedback from app development teams. For example, among all the developer-fixed bugs for Toutiao from September 1 to October 31, 2021, 50.8% of these bugs were uncovered by FASTBOT2. Additionally, FASTBOT2 can cover 80% of the hot-spot activities in Toutiao that are frequently visited by online users. These results corroborate FASTBOT2’s strong effectiveness.

5 RELATED WORK

We focus on discussing the industrial practice of automated Android GUI testing. Facebook Sapienz [6] adopts search-based testing to...