Practical GUI Testing of Android Applications via Model Abstraction and Refinement

[1]Tianxiao Gu [1]Chengnian Sun [2]Xiaoxing Ma [2]Chun Cao
[2]Chang Xu [2]Yuan Yao [3]Qirun Zhang [2]Jian Lu [1,4]Zhendong Su
[1]*University of California, Davis, USA* [3]*Georgia Institute of Technology, USA* [4]*ETH Zurich, Switzerland*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, China*
{txgu,cnsun}@ucdavis.edu, {xxm,caochun,changxu,y.yao,lj}@nju.edu.cn, qrzhang@gatech.edu, zhendong.su@inf.ethz.ch

Abstract—**This paper introduces a new, fully automated model-based approach for effective testing of Android apps. Different from existing model-based approaches that guide testing with a static GUI model (*i.e.*, the model does not evolve its *abstraction* during testing, and is thus often imprecise), our approach dynamically optimizes the model by leveraging the runtime information during testing. This capability of model evolution significantly improves model precision, and thus dramatically enhances the testing effectiveness compared to existing approaches, which our evaluation confirms. We have realized our technique in a practical tool, APE. On 15 large, widely-used apps from the Google Play Store, APE outperforms the state-of-the-art Android GUI testing tools in terms of both testing coverage and the number of detected unique crashes. To further demonstrate APE's effectiveness and usability, we conduct another evaluation of APE on 1,316 popular apps, where it found 537 unique crashes. Out of the 38 reported crashes, 13 have been fixed and 5 have been confirmed.**

Index Terms—**GUI testing; mobile app testing; CEGAR;**

I. INTRODUCTION

Mobile application (app) testing heavily involves human effort [1]. For example, a human tester writes code to simulate GUI actions (*e.g.*, clicking a button) to drive the execution of Android apps [2]. This process is not only time-consuming but also error-prone. Moreover, when the GUI changes, the tester has to make nontrivial modifications to their existing test scripts [1]. To mitigate these problems, many automated GUI testing techniques have recently been proposed [3], [4].

Automated GUI Testing for Android Apps. Monkey [5], a GUI fuzzing tool developed by Google, generates purely random events [6] as test input without any guidance. Thus, it does not guarantee steering test exploration to uniformly traverse the GUIs (*i.e.*, low activity coverage [7]), and cannot incorporate user-defined rules such as inputting a password [7] or prohibiting logging out. Additionally, the generated events are low-level, with hard-coded coordinates, and usually excessively long, which complicates reproduction and debugging [3], [8]. To overcome Monkey's limitations, techniques such as evolutionary algorithms [9] and symbolic execution [10] have been adapted to guide input generation. However, such approaches are computationally expensive and do not yet scale in practice [11].

An alternative approach for performing Android GUI testing is model-based [4], [12], [13]. A model is usually a finite state machine, where each state has a set of model actions, and each transition between states is labeled with a model action of the source state. In practice, almost no app comes with a model. Existing testing tools thus build a GUI-based model by abstracting/mapping GUI actions to model actions and GUI views to states, respectively.

A model offers at least three types of benefits to GUI testing. First, a model can be used to guide the exploitation of an app. A testing tool can traverse the model using specific guidance to systematically generate action sequences and then replay the action sequences to test the app [4]. Second, a model-based testing tool generates input sequences composed of high-level model actions rather than low-level events, which can facilitate replaying [14]. Third, a proper abstraction can be applied to the model, which in turn can help mitigate the explosion of GUI actions. Through abstraction, many GUI actions with the same behavior can be mapped to the same model action. Since these GUI actions behave the same, the testing tool does not need to exercise each of them and can instead select a representative GUI action among them when executing the model action.

State Abstraction. Mapping each GUI action to a model action is the most critical step in state abstraction. A state is usually identified by the set of its model actions since states with the same set of model actions can be merged [15]. All existing model-based approaches apply a *static* abstraction based on certain heuristics throughout the testing of an app. Designing a proper abstraction is challenging. First, if the model is overly fine-grained, the testing tool cannot systematically explore the model due to *state explosion*. Second, if the model is overly coarse-grained, the testing tool cannot gather sufficiently accurate knowledge on model actions to realize effective guidance (*i.e.*, difficult to replay model action sequences on the tested app). In particular, an ineffective abstraction may map multiple GUI actions with different behaviors (*e.g.*, leading to different target states) to the same model action. Therefore, a model action cannot be replayed as expected if the testing tool chooses a GUI action behaving differently from the one chosen during model construction. Accordingly, all subsequent actions in the sequence depending on the model action can neither be successfully replayed.

Our Approach. This paper proposes APE, a new, practical model-based automated GUI testing technique for Android apps via effective *dynamic* model abstraction. At the beginning, a default abstraction is used to initiate the testing process. This initial abstraction may be ineffective. Based on the runtime information observed during testing, APE gradually refines the model by searching for a more suitable abstraction, an abstraction that effectively balances the size and precision of the model. This dynamic nature distinguishes our approach from all existing state-of-the-art techniques as they rely solely on static abstractions. Instead of operating on a fixed abstraction granularity, our approach dynamically adjusts the granularity as needed. Specifically, APE represents the dynamic abstraction with *a decision tree*, and tunes it on-the-fly with the feedback obtained during testing. This decision tree-based representation of model abstractions greatly improves testing effectiveness, as demonstrated in our comprehensive evaluation (Section IV).

We compared APE with the state-of-art testing tools (*i.e.*, Monkey [5], SAPIENZ [3], and STOAT [4]) on 15 large, widely-used apps from the Google Play Store. APE complements the state-of-art-tools (see Section IV-C). APE achieved higher code coverage and detected more unique crashes (*i.e.*, unique stack traces defined in Section IV-C) than the other tools on the 15 benchmark apps. Specifically, APE consistently resulted in 26–78%, 17–22% and 14–26% relative improvements over the three tools in terms of average activity coverage, method coverage, and instruction coverage, respectively. Though these apps have been well tested, within one hour APE managed to find 62 unique crashes by testing their GUIs, whereas Monkey found 44, SAPIENZ 40, and STOAT 31. To further demonstrate the usability and effectiveness of APE, we conducted another large-scale evaluation on 1,316 apps from the Google Play Store. APE found 537 crashes from 281 apps in total. We reported 38 crashes to developers with detailed steps to reliably reproduce these crashes, where 13 crashes have already been fixed and 5 crashes have been confirmed (fixes pending).

Contributions. This paper makes the following contributions:

- We propose a novel, fully automated, model-based technique for Android GUI testing. The major difference of our approach from existing techniques is the ability to dynamically evolve GUI models toward ones that discard all irrelevant GUI details while faithfully reflect the runtime states of the app under test.
- We realize dynamic model abstraction via a novel type of decision trees, which can expressively represent a wide range of abstractions and thus enables APE to effectively, dynamically refine and coarsen abstractions to balance model precision and size.
- Our extensive evaluation demonstrates that APE outperforms the state-of-the-art tools. It can automatically explore many more GUIs (*i.e.*, activities) than the other tools. Therefore, according to our evaluation results, it not only increased code coverage but also found more unique crashes.
- We implement the proposed technique into a practical tool, APE, and make it publicly available.[1]

[1]http://gutianxiao.com/ape

Paper Organization. The remainder of this paper is organized as follows. Section II introduces necessary background. Section III presents the approach. Section IV details our extensive evaluation. Section V surveys related work, and Section VI concludes.

II. BACKGROUND

This section introduces relevant background on model-based Android GUI testing and its challenges.

A. GUI of Android Apps

In an Android app, an activity [16] is a composition of *widgets*. These widgets are organized into a tree-like structure, named *GUI tree* in this paper [4], [12], [17]. A widget can be a button, a text box, or a container with a layout. It supports various GUI actions such as clicking and swiping. A widget has four categories of attributes describing its *type* (*e.g.*, class), *appearance* (*e.g.*, text), *functionalities* (*e.g.*, clickable and scrollable), and *the designated order among sibling widgets* (*i.e.*, index). Each attribute is a key-value pair. We use $\mathbf{i}$, $\mathbf{c}$, and $\mathbf{t}$ to denote the key of an index, class, and text attribute, respectively. For example, an index attribute whose value is 0 can be denoted by $\mathbf{i} = 0$.

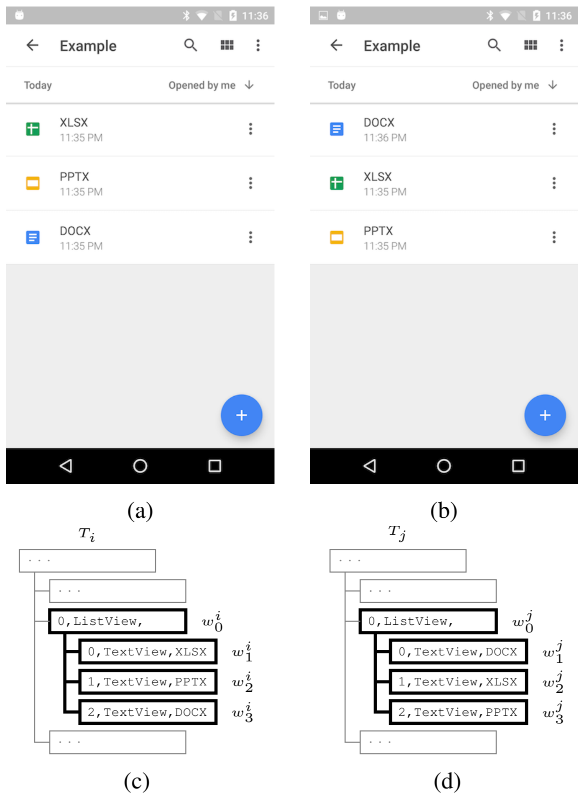


Fig. 1: Figs. 1a and 1b are two GUI snapshots of Google Drive. Figs. 1c and 1d are their corresponding GUI trees.

A GUI tree T is a rooted, ordered tree where each node w is a GUI widget that has a set of attributes, denoted by *attributes*(w). The Android SDK has provided a tool [18] to obtain the GUI tree of an activity. Figs. 1c and 1d show the corresponding (simplified) GUI trees for Figs. 1a and 1b, respectively. Since only **bold** parts are of interest in this paper, we assumed that T_i and T_j are rooted at w_0^i and w_0^j, respectively.

B. Attribute Path

A testing tool needs to properly identify widgets to accumulate testing knowledge on them. The memory address of the runtime object representing the widget cannot be used to identify the widget because the same GUI may be created and disposed many times. In a GUI tree T, given a widget w_n, a *node path* $\omega = \langle w_1, w_2, \cdots, w_n \rangle$ is a sequence of tree nodes (*i.e.*, widgets) that are on the traversal path from one of its ancestors w_1 to w_n. This node path is referred to as an *absolute node path* if the first node w_1 is the root of the tree, and otherwise a *relative node path*, which is similar to the concept of absolute and relative file paths in file systems. An absolute node path uniquely identifies a widget in the tree. For example, w_1^i's unique node path in Fig. 1c is $\langle w_0^i, w_1^i \rangle$.

*Definition 1 (***Attribute Path***):* Given a widget w_n and one of its node paths $\omega = \langle w_1, w_2, \cdots, w_n \rangle$, an attribute path $\pi = \langle a_1, a_2, \cdots, a_n \rangle$ is a projection of ω, such that,

$$\forall_{i=1}^{n} a_i \subseteq \textit{attributes}(w_i).$$

That is, each a_i is a subset of the attributes of the corresponding widget w_i.

*Definition 2 (***Full Attribute Path***):* An attribute path $\pi = \langle a_1, a_2, \cdots, a_n \rangle$ is a full attribute path if

- its $\omega = \langle w_1, w_2, \cdots, w_n \rangle$ is an absolute node path, and
- $\forall_{i=1}^{n} a_i = \textit{attributes}(w_i)$.

To distinguish from an arbitrary attribute path, we use σ to denote a full attribute path. In this paper, a widget can be uniquely identified (in the tree) by its full attribute path, because the order of the widgets on the full attribute path resembles the hierarchy of widgets in the tree, and the index attribute of each widget determines the unique location of the widget among its siblings. Therefore, a GUI tree T can essentially be represented with, and is equivalent to, the set of full attribute paths of all its widgets. Table I shows the full attribute path for each widget in Fig. 1.

*Definition 3 (***Attribute Path Reduction***):* An attribute path reducer is a function R that takes as input an attribute path $\pi = \langle a_1, a_2, \cdots, a_n \rangle$, and returns a new attribute path $\pi' = \langle b_m, \cdots, b_n \rangle$, such that,

$$1 \leq m \leq n \ \wedge \ \forall_{i=m}^{n} b_i \subseteq a_i.$$

In other words, π' is a suffix of π, and each element of π' is a subset of the corresponding element of π.

Widget Assumption. A GUI action is identified by its widget σ and action type τ, *i.e.*, $\langle \sigma, \tau \rangle$. For a simpler presentation, we assume that every widget supports only one GUI action and omit action types and functionality attributes throughout the paper. The relation between widgets and GUI actions becomes bijective, and we use them (*i.e.*, σ and $\langle \sigma, \tau \rangle$) interchangeably. Note that this assumption is for illustration only — our technique also supports widgets with multiple actions by using functionality attributes (see Section III-B).

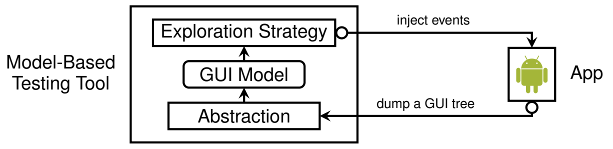


Fig. 2: The typical workflow of model-based GUI testing.

C. Model-Based Android GUI Testing

Fig. 2 depicts the typical workflow of a model-based testing tool [4], [12], [15], [17]. Such a tool interacts iteratively with the app under test. Initially, the testing tool starts with an empty state machine as the model. During each iteration, the testing tool (1) obtains the current GUI tree of the app, (2) identifies an existing, corresponding state, or creates a new state for this GUI tree, and (3) picks a model action and determines a concrete GUI action to interact with the app.

A model-based testing tool aims at exploring the app to discover new widgets and exploiting the app to exercise interactions among the discovered widgets. In practice, modern Android apps usually have a large number of widgets. To mitigate this scalability challenge, a model-based testing tool further attempts to identify equivalent GUI actions and abstract them to a same model action to reduce the search space. However, it is nontrivial to determine whether two widgets are equivalent. A full attribute path usually contains irrelevant information, which prevents two "semantically" equivalent widgets from being discovered. For example, w_1^i in Fig. 1c and w_2^j in Fig. 1d are essentially equivalent but have different full attribute paths, as shown in Table I. Relying solely on full attribute paths, a testing tool will view the two widgets differently, and the model size is consequently increased.

State Abstraction. *State abstraction* refers to the procedure that identifies equivalent GUI trees and actions, and maps them to the same state and the same model action, respectively. In general, state abstraction is realized based on the similarity of full attribute paths of GUI actions [4], [12], [15]. Specifically, state abstraction determines two GUI actions as equivalent if (1) they have the same action type and (2) their full attribute paths can be reduced to the same attribute path π by removing irrelevant attributes under certain reduction rules. The corresponding model action is denoted as π ($\langle \pi, \tau \rangle$ when action type τ is considered). Similarly, state abstraction determines two GUI trees as equivalent if all of their GUI actions can be reduced to the same set of attribute paths. The corresponding model state is denoted as the set of all model actions (*i.e.*, the set of attribute paths π) [15].

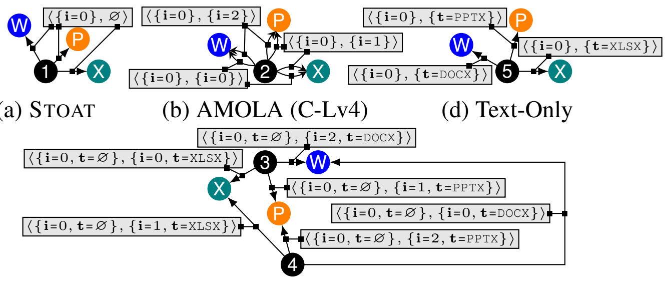
However, an automated testing tool has no prior knowledge of the apps under test and can only define reduction rules heuristically. Note that a testing tool can apply different reductions to different widgets to achieve a proper abstraction granularity. For example, STOAT [4] assumes that child widgets of `ListView` behave equivalently and realizes this heuristic in its state abstraction. Specifically, STOAT removes all attributes from child widgets of `ListView`, and removes only types and texts from other widgets. For the examples in

TABLE I: Model actions (*i.e.*, reduced attribute paths) by different abstractions.

Tree	Widget	Full Attribute Path	STOAT	AMOLA (C-Lv4)	AMOLA (C-Lv5)	Text-Only
T_i	w_0^i	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$
	w_1^i	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{XLSX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}0, \mathbf{t}{=}\texttt{XLSX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{XLSX}\}\rangle$
	w_2^i	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}1, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{PPTX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}1\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}1, \mathbf{t}{=}\texttt{PPTX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{PPTX}\}\rangle$
	w_3^i	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}2, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{DOCX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}2\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}2, \mathbf{t}{=}\texttt{DOCX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{DOCX}\}\rangle$
T_j	w_0^j	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}\rangle$	$\langle\{\mathbf{i}{=}0\}\rangle$
	w_1^j	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{DOCX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}0\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}0, \mathbf{t}{=}\texttt{DOCX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{DOCX}\}\rangle$
	w_2^j	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}1, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{XLSX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}1\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}1, \mathbf{t}{=}\texttt{XLSX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{XLSX}\}\rangle$
	w_3^j	$\langle\{\mathbf{i}{=}0, \mathbf{c}{=}\texttt{LV}, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}2, \mathbf{c}{=}\texttt{TV}, \mathbf{t}{=}\texttt{PPTX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \varnothing\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{i}{=}2\}\rangle$	$\langle\{\mathbf{i}{=}0, \mathbf{t}{=}\varnothing\}, \{\mathbf{i}{=}2, \mathbf{t}{=}\texttt{PPTX}\}\rangle$	$\langle\{\mathbf{i}{=}0\}, \{\mathbf{t}{=}\texttt{PPTX}\}\rangle$

* `LV` and `TV` are abbreviations for `ListView` and `TextView`, respectively.

Fig. 1, STOAT maps all child widgets of `ListView` to the same model action (*i.e.*, $\langle\{\mathbf{i}=0\}, \varnothing\rangle$). As shown in Table I and Fig. 3a, STOAT assigns both Figs. 1a and 1b to the same state (*i.e.*, ❶) since they have the same set of model actions. In contrast, AMOLA supports five static abstractions and its C-Lv5 criterion [12] takes into account both the indices and text content, and identifies a model action for each widget. As shown in Table I and Fig. 3c, AMOLA assigns Figs. 1a and 1b to two states, *i.e.*, ❸ and ❹, respectively.


W
⟨{i=0}, ∅⟩
P
1
X
⟨{i=0}, {i=2}⟩
P
W
⟨{i=0}, {i=1}⟩
⟨{i=0}, {i=0}⟩
2
X
⟨{i=0}, {t=PPTX}⟩
P
W
⟨{i=0}, {t=XLSX}⟩
⟨{i=0}, {t=DOCX}⟩
5
X
⟨{i=0, t=∅}, {i=2, t=DOCX}⟩
⟨{i=0, t=∅}, {i=0, t=XLSX}⟩
3
W
X
⟨{i=0, t=∅}, {i=1, t=PPTX}⟩
P
⟨{i=0, t=∅}, {i=0, t=DOCX}⟩
⟨{i=0, t=∅}, {i=1, t=XLSX}⟩
⟨{i=0, t=∅}, {i=2, t=PPTX}⟩
4

(a) STOAT (b) AMOLA (C-Lv4) (d) Text-Only

(c) AMOLA (C-Lv5)

Fig. 3: Partial models. Each circle is a state and each edge is a state transition labeled with a model action. States Ⓦ, Ⓧ and Ⓟ are *w.r.t.* GUI trees of file viewers for DOCX, XLSX and PPTX, respectively. Other states are *w.r.t.* GUI trees in Fig. 1. Model actions of `ListView` (*i.e.*, w_0^i and w_0^j) are omitted.

A good state abstraction should balance trade-offs between the model size and model precision. On one hand, the state abstraction should be coarse to avoid state explosion by tolerating differences of GUIs that are irrelevant to testing. For example, AMOLA builds very fine-grained models, but the models can quickly explode in size as they incorporate much testing-irrelevant GUI information. In constrast, STOAT does not have this problem as it builds a coarse-grained model.

On the other hand, the state abstraction should precisely model the runtime states of an app in order to interact with the app. For example, AMOLA (C-Lv4) keeps indices only, and thus can distinguish different file items in the same GUI but not among different GUIs. STOAT cannot distinguish different file items even in the same GUI. As shown in Fig. 3a, the model action $\langle\{\mathbf{i}=0\}, \varnothing\rangle$ actually represents a set of inequivalent widgets and leads to three different target states (*i.e.*, Ⓦ, Ⓧ and Ⓟ). When the guidance in STOAT prompts it to open a DOCX file (*i.e.*, reach the state Ⓦ), the actually opened file may be PPTX or XLSX (*i.e.*, the actual target states may be Ⓧ or Ⓟ). This divergent behavior will misguide the testing tool.

A proper way to identify model actions here is to include the file name (text) only. this state abstraction identifies three actions and a single state (*i.e.*, ❺ in Fig. 3d) for GUIs in Figs. 1a and 1b and also avoids mapping inequivalent widgets to the same model action. Unfortunately, none of the existing techniques supports such a proper abstraction. To address these difficulties, we propose a technique that can dynamically optimize the abstraction during testing. Specifically, we start the testing from a default abstraction. During the testing, we systematically and efficiently improve the abstraction toward the one that balances the size and precision of the model.

III. APPROACH

A. Model

Definition 4 (**Model**): A model M in APE is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L})$, where

- $\mathcal{S}$, the set of states. $S \in \mathcal{S}$ is a set of attribute paths.
- $\mathcal{A}$, the set of model actions. Each model action $\pi \in \mathcal{A}$ is an attribute path, and $\mathcal{A} = \bigcup_{S \in \mathcal{S}} S$.
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, the set of state transitions. Each transition (S, π, S') has a source state S, a target state S', and a transition label that is a model action π.
- $\mathcal{L}$: the abstraction function, which reduces a full attribute path σ to an attribute path π, *i.e.*, $\mathcal{L}(\sigma) = \pi$.

We also record every *GUI transition* between iterations. Each GUI transition is a tuple of (T, σ, T'), where T and T' are GUI trees and σ is the chosen GUI action for simulation.

To facilitate presentation, we derive two variants from $\mathcal{L}$, $\mathbb{L}_{\mathcal{L}}$ and $\mathfrak{L}_{\mathcal{L}}$, to process a GUI tree and a GUI transition, respectively. Given a GUI tree T (*i.e.*, a set of full attribute paths), $\mathbb{L}_{\mathcal{L}}$ can be used to find its corresponding state S,

$$S = \mathbb{L}_{\mathcal{L}}(T) = \{\pi | \pi = \mathcal{L}(\sigma) \wedge \sigma \in T\}.$$

And given a GUI transition (T, σ, T'), $\mathfrak{L}_{\mathcal{L}}$ can be used to find its corresponding state transition (S, π, S'),

$$(S, \pi, S') = \mathfrak{L}_{\mathcal{L}}((T, \sigma, T')) = (\mathbb{L}_{\mathcal{L}}(T), \mathcal{L}(\sigma), \mathbb{L}_{\mathcal{L}}(T')).$$

Algorithm 1 sketches the workflow of model-based testing shown in Fig. 2. Initially, the model M is empty. At the beginning of each iteration, S and π refer to the state and the chosen model action in the previous iteration. Next,

Algorithm 1: Model construction.

```
Input: Testing budget $B$, initial abstraction function $\mathcal{L}$.
Output: The model $M$.
1 $(M, S, \pi) \leftarrow ((\varnothing, \varnothing, \varnothing, \mathcal{L}), \varnothing, \varnothing)$        ▷ Initialization.
2 while $B > 0$ do
3     $B \leftarrow B - 1$        ▷ Decrease the testing budget.
4     $T' \leftarrow$ CaptureGUITree()        ▷ Use uiautomator.
5     $M \leftarrow$ UptAndOptModel($M, S, \pi, T'$)        ▷ See Algorithm 2.
6     $S \leftarrow \mathbb{L}_{\mathcal{L}}(T')$        ▷ Get the current state.
7     $\pi \leftarrow$ SelectAndSimulateAction($S$)        ▷ See Section III-D.
8 return $M$
```

function `UptAndOptModel` adds the state $\mathbb{L}_{\mathcal{L}}(T')$ and the state transition $(S, \pi, \mathbb{L}_{\mathcal{L}}(T'))$ to the model and also attempts to optimize the model as needed. Finally, we update S with the current state $\mathbb{L}_{\mathcal{L}}(T')$ and determine a new model action π on $\mathbb{L}_{\mathcal{L}}(T')$. The model action is determined with the function `SelectAndSimulateAction`, which will be informally described in Section III-D. When executing π, we first determine a set of GUI actions that map to π (*i.e.*, $\{\sigma | \sigma \in T \wedge \mathcal{L}(\sigma) = \pi\}$) and then randomly choose one of them to simulate.

B. Dynamic Abstraction Functions

As aforementioned in Section I, the key novelty of APE is its dynamic abstraction function that can dynamically adapt/change the model abstraction to balance model precision and size based on runtime information. However, realizing dynamic abstraction functions is nontrivial. First, a dynamic abstraction function should be *adaptable* at runtime. All previous approaches use a statically crafted abstraction function with a fixed abstraction granularity. Hence, adapting the abstraction needs to change the source code implementing the static abstraction function. Second, a dynamic abstraction function should be *generalizable*. For the examples in Fig. 1, an abstraction function should apply to not only T_i and T_j but also any GUI trees after reordering files. We cannot simply use a hash map between full attribute paths and attribute paths as the abstraction function, because new GUI trees and new attribute paths are continuously being discovered when the testing progresses and they are not in the hash map of such an abstraction function. Third, a dynamic abstraction should also be *human-interpretable* so that users can incorporate critical rules to further improve the dynamic abstraction.

Different from existing approaches, we represent an abstraction function as a decision tree that determines a reducer for a given σ. A decision tree is a rooted tree, where each node is a reducer R and each edge (branch) is labeled with an attribute path π. Here, π is called the *selector* of the branch.

Selector. A branch *selects* a full attribute path σ if σ can be reduced to the selector π of the branch. Given a GUI tree T, π selects a set of full attribute paths in T that can be reduced to π, *i.e.*, $\{\sigma | \sigma \in T \wedge \exists R : R(\sigma) = \pi\}$.

Decision Procedure. To determine the reducer for the given σ, we start from the root node of the decision tree and check whether σ can be *selected* by any branch of the current node. If yes, we move to the target node n of the branch and continue to recursively check the branches of n. Otherwise, the reducer of n is used as the output to reduce σ. And n is referred as *output node* in later discussions. As a function, an important property of the decision tree is that it should determine one and only one reducer for σ. To guarantee this property, we enforce that any σ must be selected by at most one branch.

Reducer. A reducer is the aggregation of a set of *primitive reducers*. We first define two types of primitive reducers.

*Definition 5 (***Local Reducer***):* Let A denote a set of attribute keys. Given a full attribute path $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$, a local reducer R_A removes $a_1, a_2, \ldots, a_{n-1}$ and retains attributes in A for a_n.

$$R_A(\sigma) = \langle \{(k, v) | (k, v) \in a_n \wedge k \in A\} \rangle$$

In this paper, we use four types of local reducers: R_c to select the type attributes, R_t to select the appearance attributes, R_i to select the index attribute, and $R_\varnothing$ to select no attribute.

*Definition 6 (***Parent Reducer***):* Given a full attribute path $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$, a parent reducer R_p reuses the output of the parent full attribute path $\langle a_1, a_2, \ldots, a_{n-1} \rangle$ and retains nothing for a_n.

$$R_p(\sigma) = \mathcal{L}(\langle a_1, a_2, \ldots, a_{n-1} \rangle) \oplus R_\varnothing(\sigma)$$

where $\oplus$ is the concatenation of sequences.

*Definition 7 (***Reducer Aggregation***):* Given a full attribute path $\sigma = \langle a_1, a_2, \ldots, a_n \rangle$ and two reducers R and R', suppose that $R(\sigma) = \langle b_m, b_{m+1}, \ldots, b_n \rangle$, $R'(\sigma) = \langle c_k, c_{k+1}, \ldots, c_n \rangle$ and $m \leq k$. The aggregation $R \bowtie R'(\sigma)$ is the reverse element-wise union of $R(\sigma)$ and $R'(\sigma)$:

$$R \bowtie R' = \langle b_m, b_{m+1}, \ldots, b_k \cup c_k, b_{k+1} \cup c_{k+1}, \ldots, b_n \cup c_n \rangle.$$

We have defined five primitive reducers in this paper and obtain 2^4 reducers in total (denoted by $\mathbb{R}$) since $R_\varnothing \bowtie R = R$. We also define a partial order over all reducers in $\mathbb{R}$. We say that a reducer R' is finer than another reducer R (*i.e.*, $R' \sqsubset R$) if R' consists of more primitive reducers than R.

Multiple Action Types. Recall that we have assumed that a widget supports only one action type in Section II-A. To support multiple action types, we can represent a GUI action as $\langle \sigma, \tau \rangle$ and a model action as $\langle \pi, \tau \rangle$. A selector π selects all $\langle \sigma, \tau \rangle$ in which σ can be reduced to π. The decision tree can still be used to realize $\mathcal{L}(\langle \sigma, \tau \rangle) = \langle \pi, \tau \rangle$.

$$\begin{tikzcd} \boxed{R_i \bowtie R_p} \arrow[r, "\langle \{\mathbf{c}=\mathtt{LV}\}, \varnothing \rangle"] & \boxed{R_p} \end{tikzcd}$$

Fig. 4: The decision tree of STOAT's abstraction function ($\mathcal{L}^s$).

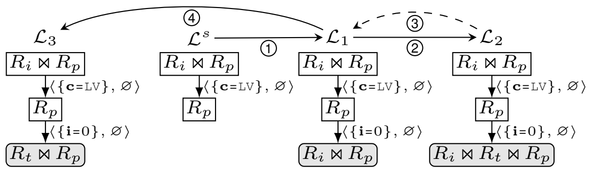
Example. The decision tree in Fig. 4 implements STOAT's abstraction function (denoted by $\mathcal{L}^s$), which we will use as the starting abstraction function to illustrate dynamic optimization of our approach in Section III-C. Take widgets w_0^i and w_1^i in Fig. 1c as examples. $\mathcal{L}^s$ assigns $R_i \bowtie R_p$ to w_0^i because w_0^i is *not* a child of a `ListView` and *cannot* be selected by $\langle \{\mathbf{c} = \mathtt{LV}\}, \varnothing \rangle$. $\mathcal{L}^s$ assigns R_p to w_1^i because w_1^i is a child of a `ListView` and can be selected by $\langle \{\mathbf{c} = \mathtt{LV}\}, \varnothing \rangle$. Since w_0^i is the root and has no parent, R_p selects nothing for it.

The attributes path of w_0^i is actually created by R_i, which is $\langle\{\mathbf{i}=0\}\rangle$. Since w_1^i is a child of w_0^i, R_p reuses $\langle\{\mathbf{i}=0\}\rangle$ for w_1^i. The final attribute path of w_1^i is $\langle\{\mathbf{i}=0\}, \varnothing\rangle$.

C. Optimizing Abstraction Functions

APE starts with an initial abstraction function and continuously refines and coarsens the abstraction function toward a proper model precision. Specifically, refinement either replaces the reducer R of certain leaf output node in the decision tree with a not coarser reducer R' (*i.e.*, $R' \not\sqsupset R$) or inserts a new branch with a finer reducer R' (*i.e.*, $R' \sqsubset R$) to certain output node, while coarsening reverts the current abstraction function $\mathcal{L}$ to the previous one $\mathcal{L}'$ that $\mathcal{L}$ is refined from. Hence, the initial abstraction function represents the lower bound of abstraction and should be coarse enough.

An extremely coarse abstraction function would map all GUI actions to the same model action and all GUI trees to the same state. Such a function builds a very simple model, and this model does not have non-deterministic transitions, because every GUI tree is mapped to the same state. To avoid this trivial case, we first refine the abstraction function until no model action abstracts more than α GUI actions. Second, a state transition (S, π, S') is *non-deterministic* if there is another transition (S, π, S''), where $S' \neq S''$. Such a pair of non-deterministic transitions usually indicates that the abstraction of the source state S or the model action π is overly coarse and needs to be refined. Hence, we refine the abstraction function aiming at eliminating every non-deterministic transition. Third, the previous refinement monotonically increases the model size, and may eventually lead to a state explosion. Thereby, we revert the abstraction function $\mathcal{L}$ to the previous one $\mathcal{L}'$ if $\mathcal{L}$ refines a state created by $\mathcal{L}'$ into β new states. Here, α and β are two configurable threshold values.



$\mathcal{L}_3$
$\mathcal{L}^s$
$\mathcal{L}_1$
$\mathcal{L}_2$
①
②
③
④
$R_i \bowtie R_p$
$\langle\{\mathbf{c}=\texttt{LV}\}, \varnothing\rangle$
R_p
$\langle\{\mathbf{i}=0\}, \varnothing\rangle$
$R_t \bowtie R_p$
$R_i \bowtie R_t \bowtie R_p$

Fig. 5: Example of the optimization of abstraction functions.

Example. Fig. 5 depicts the evolution from Stoat's abstraction function $\mathcal{L}^s$ to the text-only abstraction function (*i.e.*, $\mathcal{L}_3$) discussed in Section II-C. As shown in Table I and Fig. 3a, suppose that APE visits the model action $\langle\{\mathbf{i}=0\}, \varnothing\rangle$ only twice, on T_i and T_j, and simulates two GUI actions to click widgets w_2^i and w_1^j, respectively. Since clicking w_2^i and w_1^j leads to two different target states (*i.e.*, P and W), APE can detect that $\langle\{\mathbf{i}=0\}, \varnothing\rangle$ is non-deterministic. Suppose that APE prefers to use $R_i \bowtie R_p$ to refine $\langle\{\mathbf{i}=0\}, \varnothing\rangle$ and map w_2^i and w_1^j to $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=1\}\rangle$ and $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=0\}\rangle$, respectively. Since the initial decision tree may incorporate user-defined rules, APE does not modify its nodes (*i.e.*, rectangle nodes) and inserts a new branch into $\mathcal{L}^s$ to create $\mathcal{L}_1$. $\mathcal{L}_1$ only *temporally* eliminate the non-deterministic transitions, because APE cannot detect that $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=1\}\rangle$ or $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=0\}\rangle$ is non-deterministic when both $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=1\}\rangle$ and $\langle\{\mathbf{i}=0\}, \{\mathbf{i}=0\}\rangle$ have a single transition each, *i.e.*, by clicking w_2^i and w_1^j, respectively. APE will detect the non-determinism (see Fig. 3b) when there are more transitions. Suppose that APE further refines $\mathcal{L}_1$ to $\mathcal{L}_2$ with a finer reducer $R_i \bowtie R_t \bowtie R_p$. Similar to AMOLA C-Lv5, $\mathcal{L}_2$ cannot tolerate file reordering and leads to the state explosion. Then, APE reverts $\mathcal{L}_2$ to $\mathcal{L}_1$ and blacklists $\mathcal{L}_2$. A later refinement will refine $\mathcal{L}_1$ to $\mathcal{L}_3$, the text-only abstraction discussed in section II-C.

Algorithm 2: Update and optimize the model.

```
1  Function UptAndOptModel($M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), S, \pi, T'$)
       Input: The new GUI tree $T'$, the model $M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L})$, the
              previous state $S$, and the previous action $\pi$.
       Output: The new model.
2      $S' \leftarrow \mathbb{L}_{\mathcal{L}}(T')$
3      $M \leftarrow (\mathcal{S} \cup \{S'\}, \mathcal{A} \cup S', \mathcal{T} \cup \{(S, \pi, S')\}, \mathcal{L})$
4      repeat $M \leftarrow$ ActionRefinement($M, T'$) until $M$ is not updated
5      $M \leftarrow$ StateCoarsening($M, T'$)
6      return StateRefinement($M, (S, \pi, S')$)
7  Function ActionRefinement($M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), T'$)
8      foreach $\pi' \in \mathbb{L}_{\mathcal{L}}(T')$ do
9          if $|\overline{\mathcal{L}}(\pi') \cap T'| > \alpha$ then
10             $R \leftarrow$ GetReducer($\pi'$)
11             foreach $R' \in \{R' | R' \in \mathbb{R} \wedge R \not\sqsubseteq R'\}$ do
12                 $\mathcal{L}' \leftarrow \mathcal{L} \cup \{R \rightarrow R'\} \mid \mathcal{L} \cup \{(R, \pi', R')\}$
13                 $\Pi \leftarrow \{\mathcal{L}'(\sigma) | \sigma \in \overline{\mathcal{L}}(\pi') \cap T'\}$
14                 if $|\Pi| > 1$ then
15                     return RebuildModel($M, \{\mathbb{L}_{\mathcal{L}}(T')\}, \mathcal{L}'$)
16     return $M$
17 Function StateCoarsening($M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), T'$)
18     $\mathcal{L}' \leftarrow$ GetPrev($\mathcal{L}$)
19     $\mathbb{S} \leftarrow \{\mathbb{L}_{\mathcal{L}}(T) | T \in \overline{\mathbb{L}_{\mathcal{L}'}}(\mathbb{L}_{\mathcal{L}'}(T'))\}$
20     if $|\mathbb{S}| > \beta$ then return RebuildModel($M, \mathbb{S}, \mathcal{L}'$) else return $M$
21 Function StateRefinement($M = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{L}), (S, \pi, S')$)
22     foreach $(S, \pi, S'') \in \{(S, \pi, S'') | (S, \pi, S'') \in \mathcal{T} \wedge S'' \neq S'\}$ do
23         foreach $\pi' \in S$ do
24             $R \leftarrow$ GetReducer($\pi'$)
25             foreach $R' \in \{R' | R' \in \mathbb{R} \wedge R \not\sqsubseteq R'\}$ do
26                 $\mathcal{L}' \leftarrow \mathcal{L} \cup \{R \rightarrow R'\} \mid \mathcal{L} \cup \{(R, \pi', R')\}$
27                 $\mathbb{S}_1 \leftarrow \{\mathbb{L}_{\mathcal{L}'}(T) | (T, \sigma, T') \in \overline{\mathfrak{L}_{\mathcal{L}'}}((S, \pi, S'))\}$
28                 $\mathbb{S}_2 \leftarrow \{\mathbb{L}_{\mathcal{L}'}(T) | (T, \sigma, T'') \in \overline{\mathfrak{L}_{\mathcal{L}'}}((S, \pi, S''))\}$
29                 if $\mathbb{S}_1 \cap \mathbb{S}_2 = \varnothing$ then
30                     return RebuildModel($M, \mathbb{S}, \mathcal{L}'$)
31     return $M$
```

We use three requirements with threshold α and β to measure the balance between model precision and model size.

1) No model action abstracts more than α GUI actions in a GUI tree, meaning that we should refine each model action until it abstracts fewer than α GUI actions.
2) Non-deterministic transitions should be as few as possible, meaning that we should keep optimizing the abstraction function until no non-deterministic transition can be eliminated.
3) No model state created by a previous $\mathcal{L}$ is refined into β new states by the refined version of $\mathcal{L}$.

Algorithm 2 aims at balancing the model precision and size. We first introduce inverse mappings to facilitate explaining it.

- $\overline{\mathcal{L}}(\pi)$ is the set of GUI actions that are abstracted to π.

- $\overline{\mathbb{L}_{\mathcal{L}}}(S)$ is the set of GUI trees that are abstracted to S.
- $\overline{\mathfrak{L}_{\mathcal{L}'}}((S, \pi, S'))$ is the set of GUI transitions that are abstracted to (S, π, S').

Function `UptAndOptModel` first adds the new state S' and transition (S, π, S') into the model. Then, it checks whether S' and (S, π, S') violates any requirements and if yes further attempts to optimize S' and (S, π, S') to incrementally optimize the model via function `ActionRefinement`, `StateRefinement` and `StateCoarsening`. APE keeps using the model if it fails to optimize S' or (S, π, S').

Action Refinement. Function `ActionRefinement` refines each model action that abstracts more than α GUI actions in the GUI tree T' (*i.e.*, $|\overline{\mathcal{L}}(\pi') \cap T'| > \alpha$). For such a π', the function tries to find a new abstraction function that refines π' into multiple model actions. Function `GetReducer` returns the reducer R that creates π'. Since any reducer R' that is not coarser than R (*i.e.*, $R' \not\sqsupseteq R$) may refine π', a new abstraction function is created by either replacing R with a not coarser R' (denoted by $\mathcal{L} \cup \{R \to R'\}$) or appending a new branch with a finer R' to the output node of R (denoted by $\mathcal{L} \cup \{(R, \pi', R')\}$). If $\mathcal{L}'$ refines π' into multiple model actions, we replace $\mathcal{L}$ by $\mathcal{L}'$ and rebuild the model accordingly. Since we only do replace on leaf nodes and append at most one branch a time, no node can be selected by multiple branches. The refined decision tree still guarantees to be a valid function. Function `RebuildModel` takes the current model, the set of affected states and the new abstraction function as input, removes all affected model actions, states and transitions, and applies the new abstraction function to re-add states and transitions for all affected GUI trees and GUI transitions. Since the model may be rebuilt, S' at line 8 may be stale. Hence, function `ActionRefinement` repeats until M is not updated, *i.e.*, no model action needs refinement.

State Coarsening. Function `StateCoarsening` (1) applies a previous $\mathbb{L}_{\mathcal{L}'}$ to T' to obtain the old state $\mathbb{L}_{\mathcal{L}'}(T')$, (2) retrieves all GUI trees that can be abstracted to the old state, *i.e.*, $\overline{\mathbb{L}_{\mathcal{L}'}}(\mathbb{L}_{\mathcal{L}'}(T'))$, (3) obtains the current states $\mathbb{S}$ of these GUI trees, and (4) reverts $\mathcal{L}$ to $\mathcal{L}'$ if $|\mathbb{S}| > \beta$, which means $\mathcal{L}$ is much finer than $\mathcal{L}'$.

State Refinement. To eliminate non-deterministic transitions (S, π, S') and (S, π, S''), function `StateRefinement` tries to refine each model action of S following the same steps as `ActionRefinement`, except that $\mathcal{L}'$ must refine S to different states or refine π to different model actions. We have implemented the latter case but omit its details in Algorithm 2 since it is just a variant of `ActionRefinement`.

Parameter Selection. A model action will be refined if it leads to non-deterministic transitions no matter what α is. The threshold α is set to 3, which is good in practice. If α is too large, the exploration strategy should sample each model action more times. If α is too small, the testing tool cannot tolerate minor differences. The threshold β is calculated based on the number of primitive reducers of the finest reducer in the decision tree. The largest value of β is 8 by default.

Algorithm Decisions. When two refinements can both eliminate the non-deterministic transitions, we prefer the refinement that creates fewer states and then fewer model actions in the new model at first, *e.g.*, prefer $\mathcal{L}_1$ to $\mathcal{L}_2$ in Fig. 5. Since refinement may conflict with coarsening, the strictly balanced model under α and β may not even exist. When the conflict arises, we prefer coarsening to suppress the state explosion because the corresponding GUIs have been explored sufficiently when the explosion is detected.

D. Exploration Strategy

We propose to combine random and greedy into the depth first search. First, we found that some non-determinism can be tolerated by keeping the locality of the exploration. For example, APE can tolerate non-determinism caused by access time via keeping clicking the same file in Fig. 1. Hence, we discover connected sub-graphs and try to explore all model actions in a connected sub-graph before jumping to other states. Next, we only greedily visit every *newly* added unvisited model action, where model actions are matched by their attribute paths. Third, we randomly visit other model actions, and give a higher priority to model actions that are not visited or abstract more GUI actions. When failing to replay a transition, APE detaches the target states to avoid unnecessary replaying attempts. When reaching the state again by random actions, APE attaches the state to the model again. Hence, APE evolves not only state abstraction but also state connectivity.

E. Implementation

APE is built on top of Monkey and runs on both emulators and real devices. It is designed to be a drop-in replacement of Monkey and requires no customization of both the app under test and Android devices. We have tested APE's compatibility on Android 6 and 7 devices because they dominate the market share when we started to develop APE [19]. GUI trees are dumped with the accessibility API [20], the same one used in [18]. The initial decision tree uses R_c for every widget. Initially, there is only one decision tree for all states. If a state needs refinement, we build a new decision tree for it in a copy-on-write manner. Hence, we can apply different abstractions to widgets with the same full attribute paths when necessary.

IV. EVALUATION

We evaluate APE on 15 widely-used apps from the Google Play Store and compare it with three state-of-art testing tools, *i.e.*, Monkey, SAPIENZ [3] and STOAT [4]. On average, APE achieved higher code coverage and detected more crashes than all the others. Specifically, APE consistently resulted in 26–78%, 17–22% and 14–26% *relative* improvements over the other three tools in terms of average activity coverage, method coverage, and instruction coverage, respectively. Moreover, APE detected 61 crashes, while Monkey detected 44, SAPIENZ 40, and STOAT 31.

To further demonstrate APE's usability and effectiveness, we applied APE to test 1,316 apps in the Google Play Store for 30 minutes each. APE detected 537 unique crashes in 42 error types from 281 of 1,316 apps. We reported 38 crashes to developers with steps to reproduce the crashes, where 13 have been already fixed, and 5 have been accepted (fixes pending).

A. *Experimental Setup*

Setup. All tools can test apps directly without any modification to the apps and Android. However, the publicly available version of SAPIENZ only supports Android 4.4 emulators, and therefore we ran all comparative experiments on emulators. We ran SAPIENZ and STOAT on the same version of Android (*i.e.*, Android 4.4) as described in [3], [4], respectively, and ran APE and Monkey on Android 6. All emulators were run on a MacBook Pro 2016 (Intel Core i7 3.3 GHz, 16GB, Mac OS 10.13). We ran all tools with their default configurations. For Monkey and SAPIENZ, we additionally followed previous work [11], [21] to set a 200 milliseconds delay. We reused artifacts released by STOAT and pushed them to emulators before testing each app. We followed previous work [3], [11], [12] to give all testing tools except STOAT an hour to test each app. We gave STOAT an hour for model construction and two additional hours for model-based fuzzing to assess its full capability. Each experiment was repeated five times. We report the average coverage and total crashes in five runs. The method and instruction coverages were collected every minute using our VM-based tracing tool.

Benchmark Collection. We selected 15 widely-used apps that are compatible with x86 emulators, as shown in Table II. Specifically, we first randomly selected 11 apps from the editor's choice collection of the Google Play Store,[2] where four apps were also used by [21]. Second, we randomly selected another four well-maintained (*i.e.*, updated since 2017) open-source benchmark apps used by SAPIENZ and STOAT.

TABLE II: Statistics of benchmark apps.

#	Application	Activity (#)	Method (#)	Instruction (#)	Install (#)
1	Citymapper	55	70,182	1,126,836	5,000,000
2	Duolingo	57	63,303	1,150,165	100,000,000
3	Flowx	16	22,081	449,899	100,000
4	Google Drive	87	55,130	1,233,004	1,000,000,000
5	Google Translate	35	36,126	787,527	500,000,000
6	Nuzzel	42	32,587	533,618	100,000
7	30 Day Fitness	34	48,292	668,396	10,000,000
8	Zillow	85	109,338	1,554,545	10,000,000
9	Flipboard	69	49,535	985,299	500,000,000
10	VLC	22	28,264	372,488	100,000,000
11	Tunein Radio	52	90,865	1,480,064	100,000,000
12	Amaze	6	41,475	752,809	500,000
13	Book Catalogue	39	6,683	127,685	100,000
14	Any Memo	30	56,474	669,027	100,000
15	My Expenses	32	58,389	1,091,503	500,000
		661	768,724	12,982,865	

B. *Coverage of Benchmark Apps*

As shown in Table III, APE is clearly the most effective tool among the four in terms of coverage. Specifically, APE achieved the best per-app coverage on almost all apps and consistently resulted in 26–78%, 17–22% and 14–26% relative improvements over the other three tools in terms of average activity coverage, method coverage, and instruction coverage respectively. Moreover, the coverage of APE also grows much faster than the other three tools, as shown in Fig. 6. For STOAT, we reported data in both model construction and the entire

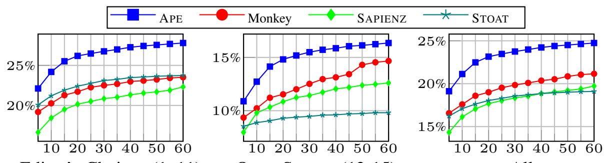
[2]https://play.google.com/store/apps/topic?id=editors_choice

TABLE III: Results on benchmark apps.

#	Activity (%)				Method (%)					Instruction (%)					Crashes (#)				
	Ape	Mo	Sa	St^1	**Ape**	Mo	Sa	St^1	St^3	**Ape**	Mo	Sa	St^1	St^3	**Ape**	Mo	Sa	St^1	St^3
1	**41**	35	31	26	**45**	43	38	41	42	**37**	35	30	33	34	**0**	7	1	0	0
2	**24**	17	22	19	**29**	26	26	25	26	**23**	20	20	19	20	**3**	0	2	0	4
3	**63**	53	60	28	**44**	40	39	32	34	**38**	34	32	25	26	**1**	1	1	0	4
4	**30**	27	26	12	**41**	39	37	37	38	**33**	31	29	29	30	**0**	2	0	0	0
5	**50**	44	46	37	**34**	33	30	30	30	**25**	24	23	22	22	**1**	0	1	2	2
6	**27**	15	26	15	**24**	17	22	20	20	**21**	14	19	17	17	**5**	3	3	0	0
7	**50**	28	40	30	**22**	15	19	19	20	**20**	13	17	17	17	**0**	0	1	0	0
8	**30**	26	19	–	**21**	19	15	–	–	**19**	17	13	–	–	**5**	7	2	–	0
9	**33**	18	18	–	**35**	26	21	–	–	**28**	20	15	–	–	**4**	0	2	–	0
10	**47**	35	37	26	**34**	29	28	26	30	**36**	31	30	27	31	**3**	3	2	1	2
11	**27**	20	21	–	**25**	21	20	–	–	**24**	20	16	–	–	**2**	1	3	–	0
12	**67**	63	60	50	**15**	14	12	12	13	**12**	11	9	9	10	**20**	16	16	2	5
13	**63**	43	56	17	**28**	23	20	10	12	**25**	22	18	9	11	**0**	0	0	0	0
14	**76**	47	71	27	**15**	11	13	10	12	**16**	11	14	10	12	**15**	1	3	3	13
15	**45**	39	36	30	**18**	18	14	14	15	**13**	14	10	10	11	**3**	3	3	1	1
	39	29	31	22	**28**	24	23	24	25	**24**	21	19	20	21	**62**	44	40	9	31

* Mo, Sa, and St are abbreviations for Monkey, SAPIENZ, and STOAT, respectively. St^1 and St^3 represent data in the model construction and the entire testing, respectively.

testing. STOAT did not report covered activities during model-based fuzzing because STOAT has an internal null intent fuzzer which directly starts activities with empty intents. In practice, STOAT can achieve 100% activity coverage on emulators. Due to the intrusive null intent fuzzing, STOAT always resulted in not responding on three apps (#8, #9 and #11). Hence, the average coverage of STOAT counted 12 apps. All tools obtained relatively low coverage on some apps, *e.g.*, app #6–8 and #11–15. The reason is that a proper environment is needed to further improve the coverage. For example, #app 15 requires premium user accounts to unlock certain functionalities.



Editor's Choices (1–11) Open Source (12-15) All

Fig. 6: Progressive instruction coverage. X axis is time in minutes and Y axis is instruction coverage.

C. *Detected Crashes of Benchmark Apps*

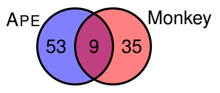
We counted only fatal errors [22] that crashed app processes during GUI exploration. This is different from the evaluation method of STOAT. First, STOAT counted as crashes many exceptions that do not terminate app processes, *e.g.*, window leaked exceptions [4], [23]. Second, STOAT also targeted to detect crashes that were triggered by null intent fuzzing [24] in addition to GUI testing. We believe that the class of crashes found by GUI testing is more important than that found by null intent fuzzing, because the former class can usually be triggered with legitimate user interactions whereas the latter class has been largely prohibited by certain security mechanisms on real devices [25] and is mostly reproducible on emulators only. Moreover, the results of our evaluation method are also consistent with their latest study [26].

TABLE IV: Crashes detected by GUI testing.

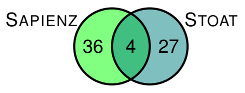
Tool	Java Exceptions (#)			Native Crashes (#)			All (#)		
	BR	UC	BA	BR	UC	BA	BR	UC	BA
APE	161	51	9	67	11	5	228	62	11
Monkey	90	34	8	43	10	6	133	44	10
SAPIENZ	101	33	10	102	7	7	203	40	13
STOAT^1	47	9	5	0	0	0	47	9	5
STOAT^3	315	31	7	0	0	0	315	31	7

* BR, UC, and BA are abbreviations for bug reports, unique crashes and buggy apps, respectively.

We show statistics of crashes detected by GUI testing in Table IV. APE detected the most unique crashes by GUI testing. To identify unique crashes, we followed STOAT to use the full normalized stack traces [4], *i.e.*, stack traces [22] without irrelevant information such as messages of exceptions or `pid` of processes [27]. Since benchmark apps were developed in Java, native crashes [27] were mostly triggered by defects of the framework rather then the apps. For example, almost a half of native crashes detected by APE, Monkey and SAPIENZ were caused by some defect of the `WebView` [28].



(a) Android 6.0



(b) Android 4.4

Fig. 7: Pairwise comparison of unique crashes (stack traces).

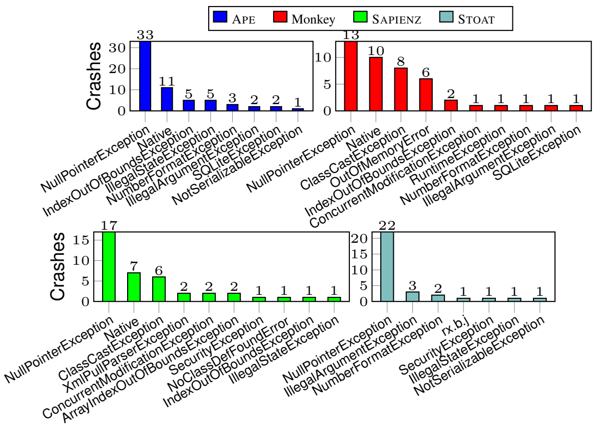


Fig. 8: Distribution of crashes types by GUI testing.

Fig. 7 depicts the pairwise comparison of unique normalized stack traces for tools on the same version of Android. We do not compare crashes detected by tools on different versions of Android via normalized stack traces because different versions of Android have different Android framework code. Particularly, Android 4.4 employs the dalvik VM while Android 6.0 employs the ART runtime [29]. The two runtime environments have different thread entry methods. Based on data in Figs. 7 and 8, each of the compared tools complements the others in crash detection and has its own advantages.

APE. APE can exhaustively exercise parts of the app locally and also discover more activities than the other tools with a dynamic GUI model. For app #14, APE detected more crashes than others because to trigger these crashes it needs to navigate in the file system thoroughly to search for some particular files and then make some complicated interactions on the files. This requires that the model should be fine enough otherwise the searching of the file may terminate prematurely. For app #10, APE detected a crash on an activity that was not discovered by all other tools in five runs.

STOAT. STOAT supports the most types of events and can find deep crashes via model-based fuzzing [4]. We plan to incorporate the powerful fuzzing strategy of STOAT into APE in future work. Due to the limitation of its implementation, STOAT did not detect crashes triggered via out-of-order events [4] as the other three tools.

Monkey. Monkey spent almost all testing budgets to generate events, *e.g.*, no restart and no model construction. Hence, Monkey detected some crashes that can only be triggered under certain stress. As shown in Fig. 8, only Monkey has detected six `OutOfMemoryError`.

SAPIENZ. SAPIENZ inherits from Monkey the same ability to trigger some particular types of crashes (Fig. 8). For example, Monkey and SAPIENZ found many `ClassCastException` and `ConcurrentModificationException`, where all of them were triggered via trackball and directional pad events. However, these crashes are not important because trackballs and directional pads are generally not available on modern Android phones. Moreover, SAPIENZ aims at minimizing event sequences at the testing time and thereby missed *deep crashes* that can only be detected via long meaningful interactions [4]. Actually, we can reduce sequences of interest afterwards [14] without sacrificing the ability to detect deep crashes.

D. *Comparative Analysis of Results on Benchmark Apps*

1) Model-based vs. Model-free: A model enables APE to effectively generate events (*e.g.*, avoid clicking non-interactive widgets) in terms of higher activity coverage with fewer events in comparison with other tools. On average, APE, SAPIENZ and Monkey generated 21,819, 33,376 and 71,125 events in an hour, respectively. STOAT is built on top of a high-level testing framework and thus did not report such events.

A model enables APE to tolerate non-determinism related to coordinate changes, while model-free tools such as Monkey and SAPIENZ rarely can. To reduce non-determinism, SAPIENZ additionally cleared the local app data before replaying every test script. However, non-determinism caused by data in file systems or from server is still a potential threat to replaying. Besides, SAPIENZ can be easily misguided due to the lack of connectivity information between GUIs. For example, app #8 has a welcome activity that requires users to customize the app. Since local app data were cleared, this activity always appeared during replaying every test script. Events that skip or finish all customizations should always be included in every test script, which is difficult to guarantee without connectivity information. In contrast, APE can easily overcome this limitation by traversing the model.

TABLE V: Statistics of models built by APE and STOAT.

	APE			STOAT		
	min	median	max	min	median	max
State (#)	131	347	725	12	87	517
Action (#)	1,277	6,531	16,141	21	820	1,307
Transition (#)	836	2,282	3,240	21	834	1,340

2) Dynamic Model vs. Static Model: APE can first try fine states and actions to explore the apps thoroughly and coarsen states to avoid unnecessarily repeated exploration of the same GUI, and produced finer models than STOAT, as shown in Table V. Note that APE can detach model states to mitigate state explosions caused by certain non-determinism during systematic exploration (see Section III-D). Our results are consistent with previous work [12]. That is, fine models are generally better than coarse models (see Table III).

E. Large-Scale Evaluation on Apps from Google Play Store

We additionally ran APE on real Nexus 5 phones to test 1,316 apps from the Google Play Store for 30 minutes each. The results are quite encouraging. APE detected 537 unique crashes in 42 error types from 281 of 1,316 apps. We reported 38 crashes to developers with steps to reproduce the crashes, where 13 have already been fixed, and 5 have been accepted (fixes pending). The results show that APE is an industry-strength practical tool in testing real-world apps.

TABLE VI: Distribution of major crash types.

Error Type	#
`NullPointerException`	226
`IllegalStateException`	58
`IllegalArgumentException`	47
`NumberFormatException`	45
`Native Crash`	33
`RuntimeException`	27
`ArrayIndexOutOfBoundsException`	13
`IndexOutOfBoundsException`	12

Table VI presents the major crash types, *i.e.*, those with more than 10 crashes each. Since we conducted the evaluation on real Nexus phones, the results excluded trivial or spurious crashes that can only be detected on emulators or by null intent fuzzing, and also excluded crashes that can only be detected on third-party Android builds (*e.g.*, on Samsung phones [30]), due to defects in third-party framework or library code.

V. RELATED WORK

We survey representative related work in this section. Ensuring the quality of mobile apps is challenging and involves manual work in practice [1]. Many pieces of existing work address specific problems of Android apps, such as context [31], [32], concurrent and asynchronous features [33], [34], fragmentation [30], [35], adverse conditions [36]–[38], and energy [39]–[41]. Android apps generally comprise of various foreground and background components, which require different testing techniques. Hence, some testing tools focus on background services [42], where the majority aims at testing the GUI [4], [7], [10], [12], [15], [17], [43]–[45] and many of them also have limited support for both [4], [43].

Many tools attempt to improve back-box tools by ripping the GUI [7], [43], [46]. Moreover, traditional model-based [12], [17], [44], symbolic execution based [10], [47], [48], and search based approaches [9] are more guided but face scalability issues on large apps [11]. Recently, SAPIENZ, a novel search-based approach, has significantly advanced the-state-of-the-art-and-practice [3], [21], [49]. The success of SAPIENZ makes us plan to leverage search-based techniques to improve APE's exploration strategy.

Many efforts have been devoted to mitigating the scalability issues, *e.g.*, hacking the app or framework to optimize performance [13], [36], [50]. For model-based testing, we can also apply a coarse-grained model [13], [15] or a probabilistic model [4] but a finer model is generally more effective [12]. APE applies a novel technique to adaptively refine and coarsen the model, which is inspired by counter-example guided abstraction refinement [51]. Currently, APE uses basic properties (*i.e.*, the number of model actions and states) to check counterexamples. We plan to incorporate GUI model checking [52], [53] in future work, *e.g.*, checking counterexamples with temporal logic. With fine models, APE may identify overly many actions. This problem can be mitigated by some action summary techniques [21], [54].

VI. CONCLUSION

This paper presents APE, a practical, fully automated model-based tool for effective testing of Android apps. APE has a general, decision tree-based representation of abstraction functions, in order to build a good testing model. It also has a novel evolution mechanism to dynamically, continuously update the model toward a good balance between the model size and model precision.

Compared to the-state-of-the-art techniques that are either modelless or based on static testing models, APE consistently outperforms them on 15 widely-used benchmark apps in terms of code coverage (14–78% relative improvements) and the number of detected bugs (61 unique crashes *v.s.* 31–44 ones). We have also applied APE to test 1,316 apps from the Google Play Store, resulting in 537 crashes. 38 crashes were reported to developers with steps for reproducing them — 13 have been fixed and 5 have been confirmed.

All our evaluations demonstrate that APE is effective, practical and promising. Currently, APE has been adopted by our industry partner, and integrated into their testing process.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Key R&D Program of China (Grant No. 2018YFB1004805) and the National Natural Science Foundation of China (Grant No. 61690204). Tianxiao Gu, Qirun Zhang, and Zhendong Su were supported in part by United States NSF Grants 1528133 and 1618158, and Google and Mozilla Faculty Research Awards.

References

[1] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real Challenges in Mobile App Development," in *Proceedings of 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2013)*, 2013, pp. 15–24.

[2] Android, "Android Testing Support Library," https://developer.android.com/topic/libraries/testing-support-library/index.html, accessed: 2018-04-01, 2018.

[3] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective Automated Testing for Android Applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016, pp. 94–105.

[4] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "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)*, 2017, pp. 245–256.

[5] Android, "UI/Application Exerciser Monkey," https://developer.android.com/studio/test/monkey.html, accessed: 2018-04-01, 2018.

[6] "UI Events," https://developer.android.com/guide/topics/ui/ui-events, accessed: 2018-04-01.

[7] X. Zeng, D. Li, W. Zheng, F. Xia, Y. Deng, W. Lam, W. Yang, and T. Xie, "Automated Test Input Generation for Android: Are We Really There Yet in an Industrial Case?" in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 2016, pp. 987–992.

[8] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI Event Traces," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2016)*, 2016, pp. 422–434.

[9] 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 2014)*, 2014, pp. 599–609.

[10] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated Concolic Testing of Smartphone Apps," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE 2012)*, 2012, pp. 59:1–59:11.

[11] S. R. Choudhary, A. Gorla, and A. Orso, "Automated Test Input Generation for Android: Are We There Yet?(e)," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015)*, 2015, pp. 429–440.

[12] Y.-M. Baek and D.-H. Bae, "Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016, pp. 238–249.

[13] T. Gu, C. Cao, T. Liu, C. Sun, J. Deng, X. Ma, and J. Lü, "AimDroid: Activity-Insulated Multi-level Automated Testing for Android Applications," in *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME 2017)*, 2017, pp. 103–114.

[14] W. Choi, K. Sen, G. Necula, and W. Wang, "DetReduce: Minimizing Android GUI Test Suites for Regression Testing," in *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, 2018, pp. 445–455.

[15] W. Choi, G. 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 and Applications (OOPSLA 2013)*, 2013, pp. 623–640.

[16] Android, "Activity," https://developer.android.com/reference/android/app/Activity.html, accessed: 2018-04-01, 2018.

[17] W. Yang, M. R. Prasad, and T. Xie, "A Grey-box Approach for Automated GUI-model Generation of Mobile Applications," in *International Conference on Fundamental Approaches to Software Engineering (FASE 2013)*. Springer, 2013, pp. 250–265.

[18] "UI Automator Viewer," https://developer.android.com/topic/libraries/testing-support-library/index.html#uia-viewer, accessed: 2018-04-01.

[19] Android, "Android Dashboard," https://developer.android.com/about/dashboards/index.html, accessed: 2018-04-01, 2018.

[20] "Accessibility," https://developer.android.com/guide/topics/ui/accessibility/, accessed: 2018-04-01.

[21] K. Mao, M. Harman, and Y. Jia, "Crowd Intelligence Enhances Automated Mobile Testing," in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, 2017, pp. 16–26.

[22] "Android Crashes," https://developer.android.com/topic/performance/vitals/crash, accessed: 2018-04-01.

[23] "Window Leak," https://stackoverflow.com/questions/2850573/activity-has-leaked-window-that-was-originally-added, accessed: 2018-04-01.

[24] R. Sasnauskas and J. Regehr, "Intent Fuzzer: Crafting Intents of Death," in *Proceedings of the 2014 Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA)*, 2014, pp. 1–5.

[25] "Export Activity," https://developer.android.com/guide/topics/manifest/activity-element#exported, accessed: 2018-04-01.

[26] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale Analysis of Framework-specific Exceptions in Android Apps," in *Proceedings of the 40th International Conference on Software Engineering (ICSE 2018)*, 2018, pp. 408–419.

[27] "Android Native Crashes," https://source.android.com/devices/tech/debug/native-crash, accessed: 2018-04-01.

[28] "Android WebView," https://developer.android.com/guide/webapps/webview, accessed: 2018-04-01.

[29] "ART and Dalvik," https://source.android.com/devices/tech/dalvik/, accessed: 2018-04-01.

[30] L. Wei, Y. Liu, and S. Cheung, "Taming Android Fragmentation: Characterizing and Detecting Compatibility Issues for Android Apps," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*, 2016, pp. 226–237.

[31] C.-J. M. Liang, N. D. Lane, N. Brouwers, L. Zhang, B. F. Karlsson, H. Liu, Y. Liu, J. Tang, X. Shan, R. Chandra, and F. Zhao, "Caiipa: Automated Large-scale Mobile App Testing Through Contextual Fuzzing," in *Proceedings of the 20th Annual International Conference on Mobile Computing and Networking (MobiCom 2014)*, 2014, pp. 519–530.

[32] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "CrashScope: A Practical Tool for Automated Testing of Android Applications," in *Proceedings of the 39th International Conference on Software Engineering Companion (ICSE-C 2017)*, 2017, pp. 15–18.

[33] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently Manifesting Asynchronous Programming Errors in Android Apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*, 2018, pp. 486–497.

[34] J. Wang, Y. Jiang, C. Xu, Q. Li, T. Gu, J. Ma, X. Ma, and J. Lu, "AATT+: Effectively Manifesting Concurrency Bugs in Android Apps," *Science of Computer Programming*, vol. 163, pp. 1 – 18, 2018.

[35] H. Khalid, M. Nagappan, E. Shihab, and A. E. Hassan, "Prioritizing the Devices to Test Your App on: A Case Study of Android Game Apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*, 2014, pp. 610–620.

[36] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, Effectively Detecting Mobile App Bugs with AppDoctor," in *Proceedings of the Ninth European Conference on Computer Systems (EuroSys 2014)*, 2014, pp. 18:1–18:15.

[37] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic Execution of Android Test Suites in Adverse Conditions," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015)*, 2015, pp. 83–93.

[38] Z. Shan, T. Azim, and I. Neamtiu, "Finding Resume and Restart Errors in Android Applications," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*, 2016, pp. 864–880.

[39] Y. Liu, C. Xu, S. Cheung, and J. Lu, "GreenDroid: Automated Diagnosis of Energy Inefficiency for Smartphone Applications," *IEEE Transactions on Software Engineering*, vol. 40, no. 9, pp. 911–940, Sept 2014.

[40] D. Li, Y. Lyu, J. Gui, and W. G. J. Halfond, "Automated Energy Optimization of HTTP Requests for Mobile Applications," in *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*, 2016, pp. 249–260.

[41] M. Linares-Vásquez, G. Bavota, C. E. B. Cárdenas, R. Oliveto, M. Di Penta, and D. Poshyvanyk, "Optimizing Energy Consumption of GUIs in Android Apps: A Multi-objective Approach," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*, 2015, pp. 143–154.

[42] L. L. Zhang, C.-J. M. Liang, Y. Liu, and E. Chen, “Systematically Testing Background Services of Mobile Apps,” in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, 2017, pp. 4–15.

[43] A. Machiry, R. Tahiliani, and M. Naik, “Dynodroid: An Input Generation System for Android Apps,” in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2013)*, 2013, pp. 224–234.

[44] 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 and Applications (OOPSLA 2013)*, 2013, pp. 641–660.

[45] A. Sadeghi, R. Jabbarvand, and S. Malek, “PATDroid: Permission-aware GUI Testing of Android,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*, 2017, pp. 220–232.

[46] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, “Using GUI Ripping for Automated Testing of Android Applications,” in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012)*, 2012, pp. 258–261.

[47] C. S. Jensen, M. R. Prasad, and A. Møller, “Automated Testing with Targeted Event Sequence Generation,” in *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA 2013)*, 2013, pp. 67–77.

[48] H. van der Merwe, B. van der Merwe, and W. Visser, “Execution and Property Specifications for JPF-android,” *SIGSOFT Software Engineering Notes*, vol. 39, no. 1, pp. 1–5, Feb. 2014.

[49] N. Alshahwan, X. Gao, M. Harman, Y. Jia, K. Mao, A. Mols, T. Tei, and I. Zorin, “Deploying Search Based Software Engineering with Sapienz at Facebook,” in *International Symposium on Search Based Software Engineering (SSBSE 2018)*, 2018, pp. 3–45.

[50] W. Song, X. Qian, and J. Huang, “EHBDroid: Beyond GUI Testing for Android Applications,” in *Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*, 2017, pp. 27–37.

[51] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-Guided Abstraction Refinement,” in *Proceedings of the 12th International Conference Computer Aided Verification (CAV 2000)*, 2000, pp. 154–169.

[52] M. L. Bolton, E. J. Bass, and R. I. Siminiceanu, “Using Formal Verification to Evaluate Human-Automation Interaction: A Review,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 43, no. 3, pp. 488–503, May 2013.

[53] M. B. Dwyer, V. Carr, and L. Hines, “Model Checking Graphical User Interfaces Using Abstractions,” in *Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 1997)*, 1997, pp. 244–261.

[54] M. Ermuth and M. Pradel, “Monkey See, Monkey Do: Effective Generation of GUI Tests with Inferred Macro Events,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA 2016)*, 2016, pp. 82–93.