Mining Android App Usages for Generating Actionable GUI-based Execution Scenarios

Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk
The College of William and Mary, Williamsburg, VA, USA
{mlinarev, mgwhite, cebernal, kpmoran, denys}@cs.wm.edu

***Abstract*—GUI-based models extracted from Android app execution traces, events, or source code can be extremely useful for challenging tasks such as the generation of scenarios or test cases. However, extracting effective models can be an expensive process. Moreover, existing approaches for automatically deriving GUI-based models are not able to generate scenarios that include events which were not observed in execution (nor event) traces. In this paper, we address these and other major challenges in our novel hybrid approach, coined as MONKEYLAB. Our approach is based on the `Record`→`Mine`→`Generate`→`Validate` framework, which relies on recording app usages that yield execution (event) traces, mining those event traces and generating execution scenarios using statistical language modeling, static and dynamic analyses, and validating the resulting scenarios using an interactive execution of the app on a real device. The framework aims at mining models capable of generating feasible and fully replayable (i.e., actionable) scenarios reflecting either natural user behavior or uncommon usages (e.g., corner cases) for a given app. We evaluated MONKEYLAB in a case study involving several medium-to-large open-source Android apps. Our results demonstrate that MONKEYLAB is able to mine GUI-based models that can be used to generate actionable execution scenarios for both natural and unnatural sequences of events on Google Nexus 7 tablets.**

***Index Terms*—GUI models, mobile apps, mining execution traces and event logs, statistical language models**

I. INTRODUCTION

The mobile handset industry has been growing at an unprecedented 23% compound annual growth rate in revenue since 2009 [51], and the growth between 2012 and 2016 is expected to be around 28% [50]. This global app economy with millions of apps (1.3M+ Android and 1.2M+ iOS apps), 2.3M developers, and billions of devices and users has been a tremendous success [52]. Many of these mobile apps have features that rival their desktop counterparts and belong to several domain categories spanning from games to medical apps. Mobile platforms enable user interaction via touchscreens and sensors (e.g., accelerometer, ambient temperature, gyroscope) that present new challenges for software testing. Furthermore, mobile developers and testers face other emerging challenges such as rapid platform/library evolution and API instability [12], [36], [37], [42], platform fragmentation [25], continuous pressure from the market for frequent releases [28], [31], and limited availability/adequacy of testing tools for mobile apps [32], [34], among many other challenges.

Although several tools are available to support automated execution of Android apps for validation purposes, in practice, testing is still performed mostly manually as recent survey study results show [32]. Limited testing time during the development process tends to prohibit the use of data from these manually generated scenarios for different devices [32] in a continuous development fashion, especially taking into account that some apps may have users coming from as many as 132 unique devices [34]. A significant amount of work is required to generate replay scripts that are coupled to screen dimensions for a single device (i.e., one script is required for each target device). Consequently, these scripts become quickly outdated when significant changes are made to the app's GUI [24], [32].

Existing research tackled some of these issues by deriving models that represent the GUI and behavior of apps. These models are abstract representations that can be decoupled from device dimensions and event locations and can still remain valid when small changes are done to the app (e.g., button location change). For instance, some representative approaches for deriving such models use either dynamic [8], [11], [15], [39], [44], [45], [48], [49] or static analyses [10], [13], [19], [30], [43], [46]. However, current approaches fall short in (i) generating scenarios that are representative of natural (i.e., typical end-user) application usages [49], (ii) taking into account the context which exists in an app's execution history [18], [49], (iii) generating sequences with previously unseen (i.e., not available in artifacts used to derive the model) but feasible events. Moreover, utilizing model-based testing techniques for GUI-based testing in industrial contexts may be particularly challenging because creating such models requires specialized expertise and using these models (for manual and automated testing) assumes a logical mapping between the model and the actual system that was modeled [7].

In practice, developers constantly test their apps *manually* by exercising apps on target devices. In fact, manual testing is usually preferred over automated approaches for testing mobile apps [32]. However, oftentimes this execution data is simply thrown on the ground and never used when an app needs to be retested. *Our key hypothesis is that all this data that is produced from regular app usages by developers, testers, or even end-users can be effectively recorded and mined to generate representative app usage scenarios (as well as the corner cases) that can be useful for automated validation purposes.* Furthermore, our intuition is that the models mined from execution (event) traces can be augmented with static analysis information to include unseen but feasible events.

In this paper, we propose a novel hybrid approach

*This work is supported by NSF CCF-1218129 and CCF-1253837 grants. grants. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

for mining GUI-based models from event logs collected during routine executions of Android apps. Our approach (MONKEYLAB) derives feasible and fully replayable GUI-based event sequences for (un)natural app usage scenarios (i.e., actionable scenarios) based on the novel `Record→Mine→Generate→Validate` framework. MONKEYLAB provides stakeholders with an automated approach for scenario generation that can be as powerful as manual testing. MONKEYLAB mines event traces and generates execution scenarios using statistical language modeling, static analysis, and dynamic analysis. To generate (un)natural event sequences, our novel approach (i) augments the vocabulary of events mined from app usages with feasible events extracted statically from the app's source code and (ii) exploits a space of possible events and transitions with different flavors of language models capable of modeling and generating combinations of events representing natural scenarios (i.e., those observed relatively frequently in app usages) and corner cases (i.e., those sequences that were observed relatively infrequently or not observed at all).

We evaluated MONKEYLAB on several medium-to-large Android apps from Google Play and compared MONKEYLAB to other commonly used approaches for generating GUI-based event scenarios. The results demonstrate that MONKEYLAB is able to generate effective and fully replayable scenarios. Moreover, MONKEYLAB is able to generate scenarios that differ from observed executions enabling it to explore other paths that could trigger unexpected app crashes.

In summary, the paper provides the following contributions:

- A `Record→Mine→Generate→Validate` framework for generating actionable scenarios for Android apps. We designed MONKEYLAB to be independent from specific Android devices or API platform versions.
- A novel mechanism for generating actionable scenarios that is rooted in mining event traces, generating execution scenarios via statistical language modeling, static and dynamic analyses, and validating these resulting scenarios using interactive executions of the app on a real device (or emulator). In particular, we explore interpolated *n*-grams and back-off models, and we propose three different flavors (i.e., up, down, and strange) for generating (un)natural scenarios.
- A thorough empirical evaluation and comparison of MONKEYLAB to competitive approaches on Android devices (Google Nexus 7 tablets). Experimental data, videos of the generated scenarios, and other accompanying tools are available in our online appendix [38].

II. BACKGROUND AND RELATED WORK

Automatic generation of GUI-based scenarios (event sequences) has applications not only in automated testing but also in creating app usage documentation [53]. In general, event sequences can be generated automatically by relying on models built (i) statically from app source code, (ii) dynamically from interactive app executions (e.g., GUI ripping) or from execution traces, (iii) manually defined by programmers, or (iv) approaches using random chains of events without any knowledge of the app such as the widely used Android `UI monkey` [23]. Once the model is defined, it can be used to generate sequences of feasible (in theory) events.

Models derived from the app source code rely on static analysis [46], symbolic [13], [30] and concolic execution [10] techniques. Rountev *et al.* [46] proposed a method for statically extracting GUI components, flows of GUI object references and their interactions. The concolic-based testing model proposed by Anand *et al.* [10] generates single events and event sequences by tracking input events from the origin to the point where they are handled.

Models derived using dynamic analysis are mostly based on interactive execution of the app for systematically identifying the GUI components and transitions (i.e., GUI ripping). The execution is usually done heuristically by using some strategy (e.g., depth-first search (DFS) or a non-uniform distribution). Examples of approaches and tools relying on systematic exploration are the work by Takala *et al.* [48], the tools *VanarSena* [45], *AndroidRipper* [8], A^3E [11], *Dynodroid* [39], *SwiftHand* [15], OME^* [44], and *MobiGUITAR* [9] that extracts a state-based model of the GUI, which is used to generate JUnit test cases.

Dynamic analysis has also been used for collecting real execution traces and inferring some grammar-based rules or statistical models describing observed events. Representative approaches of models extracted from execution traces collected apriori include *SwiftHand* [15], which uses statistical models extracted from execution traces, and the approach by Elbaum *et al.* [19] which generates test cases from user-session data. TAUTOKO [17] mines typestate models from test suite executions and then mutates the test cases to cover previously unobserved behavior. Recent work by Tonella *et al.* [49] derives language models from execution traces.

Hybrid models, such as *ORBIT* [54], have also been proposed, which combine static analysis and GUI ripping. Moreover, *Collider* [30] combines GUI models and concolic execution to generate test cases that end at a target location. *EvoDroid* [40] extracts interface and call graph models automatically, and then generates test cases using search-based techniques. Yeh *et al.* [13] track the execution paths of randomly generated input events by using symbolic execution. Afterward, the paths are used for detecting faults and vulnerabilities in the app.

Event sequences can also be generated by relying on random distributions. For instance, the Android `UI monkey` [23] is widely used for generating random input events without prior knowledge of the app's GUI. Although a tool like `monkey` helps automate testing, it is not robust since it is prone to providing a high ratio of invalid inputs. *Dynodroid* [39] also includes a strategy for generating GUI and system level events via a uniform distribution where available GUI events are discovered interactively using GUI ripping.

Regardless of the underlying analyses to build the model, model-based GUI testing is inherently difficult, because the current state of an app (available components and possible

actions) can change continually as new GUI events are executed. For example, consider a state machine-based GUI model in which each node (state) is a screen (i.e., activity) of an Android app, and the transitions are actions on specific GUI components (e.g., click the OK button). Some of the real transitions are feasible under certain conditions (e.g., delete a task in a non-empty list of a TODO app), and some states that are reachable only after executing natural (and unnatural) sequences of GUI events (e.g., customized scroll behavior on an empty list) are hard to model using state machines. In addition, although systematic-based exploration techniques could execute events on all of the GUI components of the app, the execution follows a predefined heuristic on the GUI (e.g., DFS) that neither represents natural execution scenarios nor considers execution history. Existing approaches also assume a high rate of generated event sequences that are simply infeasible [18], [49]. For instance, some events in the sequence can be infeasible in the context of a given test case (e.g., an invalid action on a component or an action on a component that is not displayed in the current GUI state). Another issue, which is pertinent to the models extracted from execution traces, is their inability to generate sequences with unseen (i.e., not observed in the traces) but feasible events.

III. Mining and Generating Actionable Execution Scenarios with MonkeyLab

Given that manual execution of Android apps for testing purposes is still preferred and relied upon over automated methods [32], we set out to build a system for generating inputs (event sequences) to mobile apps on Android that would simulate the convenience, naturalness, and power of manual testing (and utilize all the data that is produced during these app usages) while requiring little effort from developers. Also, we identified a set of four key goals that we felt such a system must satisfy in order to be useful:

1) The solution for generating executable scenarios needs to resemble manual testing, where the process of collecting test scripts allows developers/testers to interact with the app naturally by using gestures and/or GUI-based actions. One solution to this issue is represented by the event log collection capability used by *RERAN* [20]. However, *RERAN* replays only what it records and does not allow an arbitrary recombination of events for replay; moreover, scripts recorded with *RERAN* cannot be easily deciphered by testers since they are low-level, hardware-specific events that are coupled to screen locations (e.g., events collected on a Nexus 4 will not work on a Galaxy S4).
2) The solution should have context awareness of an app's execution history, where current possible actions are generated and executed as event streams to facilitate the highest possible testing coverage. In this regard, our key insight is to rely on statistical language models (n-grams) to generate event sequences by using program structure and previous executions of an app. Our motivation behind using language models (LMs) is in their capacity to represent event streams while maintaining contextual awareness of the event history. MonkeyLab uses LMs for representing sequences of events as tokens (more specifically, *Object:Action* lexemes) that can be extracted statically from the app and also mined from the app's execution. Extracting the tokens only from execution traces carries the risk of missing possible feasible events that are not observed in traces. Therefore, combining static and dynamic information may further augment the vocabulary used by LMs in such a way that some unnatural (those that did not appear in traces) but feasible tokens are included in the model, effectively taking into account both the execution history and unobserved events.
3) The solution should be able to generate *actionable scenarios* as streams of events that can be reproduced automatically on a device running the app. These scenarios should not be low-level event streams but rather human readable streams that developers can easily comprehend and are not coupled to locations in a screen. This goal can be met by translating high-level readable tokens (e.g., click OK button) to low-level event streams that can be executed automatically on a target device. These high-level readable tokens are derived from language models formulated by mining from app usages and source code.
4) The solution should not impose significant instrumentation overhead on a device or an app in order to be able to reproduce automatically generated scenarios. In order to meet this goal, we rely on the replay capability provided by the input commands for remote execution in Android [5]. However, any framework for remote execution of GUI events can be used such as Robotium [4].

Having these goals in mind, the proposed approach for generating actionable scenarios is described by the following framework: (i) developers/testers use the app naturally; the event logs representing scenarios executed by the developers/testers are `Recorded`; (ii) the logs are `Mined` to obtain event sequences described at GUI level instead of low-level events; (iii) the source code of the app and the event sequences are `Mined` to build a vocabulary of feasible events; (iv) language models are derived using the vocabulary of feasible events; (v) the models are used to `Generate` event sequences; (vi) the sequences are `Validated` on the target device where infeasible events are removed for generating *actionable scenarios* (i.e., feasible and fully reproducible). The framework and the architecture are depicted in Figure 1 as well as real examples of event logs collected in the `Record` phase, event sequences derived after the `Mine` and `Generate` phases, and actionable scenarios generated with the `Validate` phase, for the Android app (Mileage). In the following subsections, we describe each step of the framework and the components required for each step.

A. `Record`: Collecting Event Logs from the Crowd

Collecting event logs from developers/testers should be no different from manual testing (in terms of experience and overhead). Therefore, developers/testers using MonkeyLab rely on the `getevent` command for collecting event

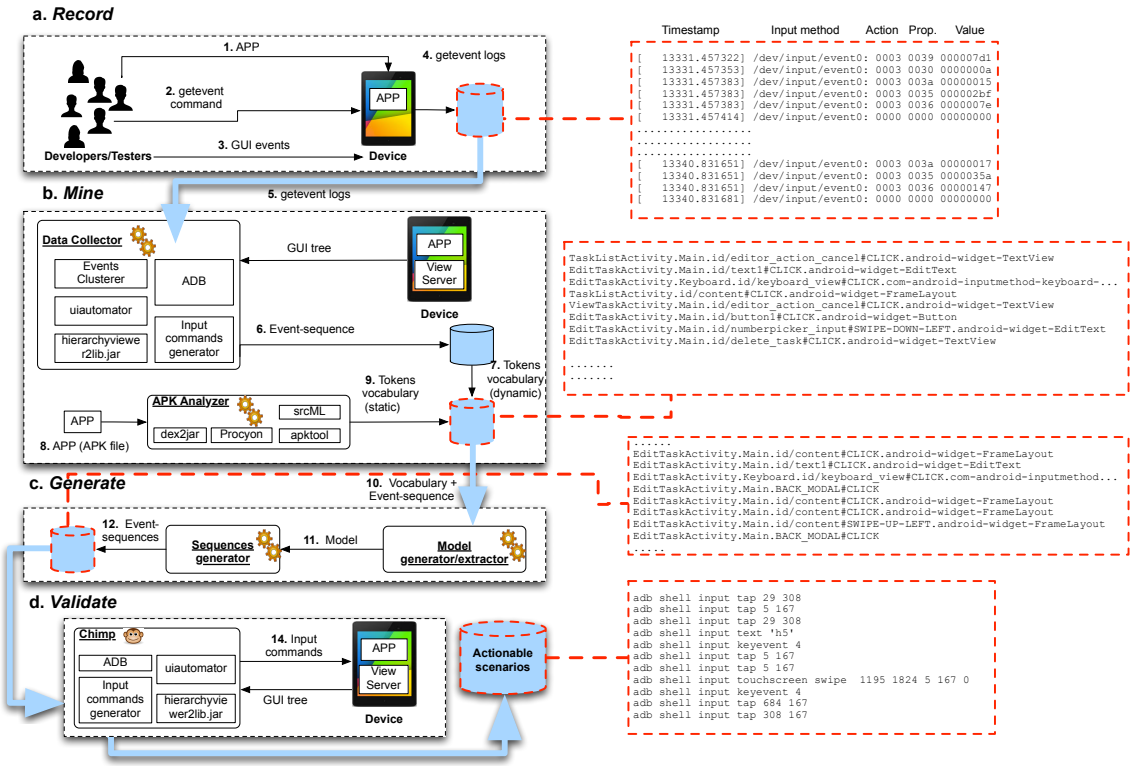

Fig. 1. MonkeyLab architecture and the `Record`→`Mine` →`Generate`→`Validate` framework

sequences, similar to *RERAN* [20]. The `getevent` command produces logs that include low-level events (see `getevent` log example in Fig. 1), representing click-based actions as well as simple (e.g., swipe) and complex gestures; those logs are collected during an app execution. After MONKEYLAB is enabled, developers/testers exercise/test the app as in manual testing. After having executed the app, the logs are generated with the `getevent` command and copied to the logs repository. Since our log collection approach poses no overhead on developers/testers (they just use apps as usual), this setup permits collecting logs on a large scale. In fact, *this log collection approach can be easily crowd-sourced, where logs are collected from daily app usages by ordinary users*.

B. `Mine`: Extracting Event Sequences from Logs and the App

The goal of the `Mine` phase is to extract the vocabulary of events (i.e., feasible events) and translate the `getevent` logs required to build the language models. The vocabulary of events is extracted from the source code of the app and also from the event logs. However, the low-level events are coupled to specific locations on the screen and do not describe the GUI components in the app; in addition, a single GUI event is represented by multiple lines in a log (see Fig. 1). A line in a `getevent` log has a timestamp, input method (e.g., screen or physical keyboard), an action (e.g., `003 = click`), a property related to the action (e.g., `0035` = x-axis position), and the property value. Therefore, to eliminate the dependency of the actions on the screen coordinates in specific devices, we translated the `getevent` logs to a GUI-based level representation: we model an event e_i—represented by multiple lines in a `getevent` log—as the tuple e_i :=<*Activity*$_i$, *Window*$_i$, *GUI-Component*$_i$, *Action*$_i$, *Component-Class*$_i$> (see examples of event sequences in Fig. 1). We included the *Window* element to distinguish actions on the activity (i.e., screen) and actions on the displayable Android keyboard. For the *Action* element, we included `click`, `long-click`, and `swipe`. This representation considers the fact that Android apps are composed mostly of Activities representing screens of the application. Each Activity is composed of GUI components that are rendered dynamically according to the app's state. Thus, the number of GUI components that are visible in an activity can be different across time. Additionally, the set of feasible actions for each component depends on the component type. For instance, `click` and `long-click` event handlers can be attached to a button but not to a `swipe` gesture handler. The *Component-Class* field is necessary to validate GUI-components in the app at runtime. This representation of e_i tuples is used for both generating the vocabulary of events from app code and event sequences from `getevent` logs.

1) Mining GUI events statically from APKs (APK Analyzer): The *APK-Analyzer* component (Fig. 1-b) uses static analysis to extract from the app source code a list of feasible GUI events. The list is used to augment the dynamically built vocabulary from user event streams. To achieve this, GUI components are extracted from decompiled APKs before links between these components to activities, windows, and actions/gestures are constructed. To extract this information, the *APK-Analyzer* (i) uses the *dex2jar* [2] and *Procyon* [47] tools for decompilation and (ii) converts the source files to an XML-based representation using *srcML* [6]. We also rely on *apktool* [1] to extract the resource files from the app's APK.

The ids, types, and hierarchy of the GUI components were extracted from the XML files of the APK resources.

The next major step in building the static vocabulary is linking each GUI component to its respective actions/gestures. Rather than parsing the source code to determine which gestures are instantiated, the *APK-Analyzer* assigns inherent gestures to the types of GUI components. For standard Android component types, the linking is done with expected gestures, e.g., `Button` would be linked with `click` and `long-click`. For custom components, the *APK-Analyzer* parses source code to determine gesture handlers and event listeners, which are attached to the custom components. After linking actions/gestures and types to the components is complete, the *APK-Analyzer* links the components to the Activities in which they appear. The list of events extracted statically from the app is represented by a set of tuples e_i. It should be noted that the APK Analyzer cannot generate a static vocabulary from obfuscated code nor code where components are instantiated dynamically.

2) Mining GUI events from event logs (Data Collector): The e_i tuples require high-level information (i.e., activity, GUI component, window) that is not provided by the `getevent` logs. Thus, we implemented a component (*Data collector* in Fig. 1-b), which is able to translate `getevent` logs into sequences of e_i tokens. The *Data collector* replays the logs in a ripping mode in order to dynamically collect the GUI information related to the event. The sentences (i.e., lines) in a log L_k are grouped to identify individual GUI events. Then each group of sentences (c) representing a GUI event is translated into a natural language description (e.g., <`click`, $\{x = 10, y = 200\}$>). For identifying the corresponding component in the GUI, we queried the `Android View Server` running on the device. The `View Server` provides a tree representation of the GUI components with some attributes such as the top-left location of a component and the dimensions. Given the event location, we traversed the GUI tree looking for the component area (top-left corner plus dimensions) that contains the event's location. For identifying the current activity we used the `adb shell dumpsys window windows` command, and then we looked for the property `mCurrentFocus|mFocusedApp`. For identifying whether the keyboard is displayed, we queried the list of current windows in the `View Server`. With the e_i tuple, we built an input event command [5] that can be remotely executed using the `Android Debugger Bridge` (e.g., `adb shell input tap x y`). Finally, we executed the input command to update the app state and continued with the next group c. The output of the procedure is the list T_k of e_i tuples extracted from the L_k `getevent` log. The T_k sequences and unique e_i tuples are copied to the sequence and token repositories, respectively.

C. `Generate`: Event Sequences with Language Models

The vocabulary of event tokens extracted from the app code and the event logs, both represented as GUI-level events, are the universe of feasible events in the app and are natural use cases respectively. They represent the GUI model of the app including activities, components, feasible actions, and natural transitions. This data can be used to build statistical models for generating streams of events as execution scenarios of the app. In fact, the `Generate` phase of our framework uses LMs [16], [29], [33] to generate (un)natural sequences of events for that purpose. These LMs are trained with the vocabulary and event sequences extracted in the `Mine` phase.

Our motivation behind using LMs is their capacity to represent event streams while maintaining a contextual awareness of the event history. Hindle *et al.* [26] demonstrated the usefulness of applying LMs to software corpora because real programs written by real people can be characterized as *natural* in the sense that they are even more repetitive than natural languages (*e.g.*, English). Tonella *et al.* [49] exploited this interpretation of naturalness by casting it to streaming events as test cases. *Our hypothesis is that event-sequences extracted from* `getevent` *logs emit a natural set of event streams, which map to core functionality, as well as an unnatural set of event streams*. Language models inherently capture natural behavior, but we also consider the complement of this natural space, thereby improving the interpretive power of these statistical models. Our novel interpretation of software LMs systematically segments the domain of a LM over dynamic traces and imputes testing semantics to each segment. For example, a sampling engine may select tokens from the natural space to improve coverage, yet it may sample from the unnatural space to crash the app. In the following subsections we provide formal definitions behind LMs, LM flavors that we defined in the context of our work as well as details on using LMs for generating event sequences.

1) Language Models: A statistical language model is a probability distribution over units of written/spoken language,which measures the probability of a sentence $s = w_1^m = w_1 w_2 ... w_m$ based on the words (a.k.a., tokens) probabilities:

$$p(s) = p(w_1^m) = \prod_{i=1}^{m} p(w_i | w_1^{i-1}) \approx \prod_{i=1}^{m} p(w_i | w_{i-n+1}^{i-1}) \quad (1)$$

In Eq. (1), the Markov assumption approximates the joint distribution by measuring the conditional probabilities of token subsequences known as n-grams. After "discounting" the model's maximum likelihood estimates, there are generally two methods for distributing the probability mass gleaned from the observed n-grams: back-off and interpolation. The Katz back-off model [33] takes the form

$$p_{\mathcal{B}}(w_i|h) = \begin{cases} \alpha(w_i|h) & c(hw_i) > k \\ \beta(h) p_{\mathcal{B}}(w_i | w_{i-n+2}^{i-1}) & \text{otherwise} \end{cases} \quad (2)$$

where $h = w_{i-n+1}^{i-1}$ is the history, $\alpha(w_i|h)$ is the discounted maximum likelihood estimate of word w_i given h, $\beta(h)$ is the back-off weight, $c(hw_i)$ is the number of times hw_i appears in the training corpus, and k is a constant, which is typically zero. If the history was observed in training, then a back-off model says the conditional probability of a word given its history is equal to the discounted estimate of the n-gram, where $\alpha(w_i|h)$ is computed using a smoothing technique such as Good-Turing

estimation [21]. Otherwise, the model truncates the history and recursively computes the probability. Back-off models only consider lower-order n-grams when $c(hw_i) = 0$ for every w_i in the vocabulary $\mathcal{V}$. On the other hand, interpolation considers lower-order n-grams whether or not $c(hw_i) > 0$. The general interpolated model takes the form

$$p_{\mathcal{I}}(w_i|h) = \alpha(w_i|h) + \beta(h)p_{\mathcal{I}}(w_i|w_{i-n+2}^{i-1}) \quad (3)$$

where $\alpha(w_i|h)$ is computed using a smoothing technique such as modified Kneser-Ney estimation [14].

2) Language Model Flavors: Back-off (`BO`) and interpolation (`INTERP`) are different approaches for computing probabilities, but each LM is a way to characterize the naturalness of a sentence. Moreover, we propose that each model can be used to generate unnatural sentences as well, and these unnatural sentences have clear software testing semantics. Parenthetically, both `BO` and `INTERP` can be decomposed to different flavors—`up`, `down`, and `strange`—that are simple transformations for driving a sampling engine to specific segments of the language model's domain.

`up` corresponds to the natural distribution over tokens, but there are subtle differences in the implementation depending on whether the language model uses `BO` or `INTERP`. For example, suppose we are given the following conditional probabilities from a LM: $\theta = \{\alpha(w_a|h) : 0.20, \alpha(w_b|h) : 0.50, \alpha(w_c|h) : 0.10, \alpha(w_d|h) : 0.10\}$. For `BO` models, `up` will sample a uniform variate using the cumulative sum of the smoothed estimates. If $c(hw) = 0$, then `up` will back-off to the next lowest order and repeat the procedure, where unigram probabilities serve as the base case in the recursion. By construction, interpolation mixes input from every n-gram that can be sliced from hw, so `up` recurses on the order through the unigrams to compute $p(w_i|h) \forall w_i \in \mathcal{V}$ using Eq. (3). Naturally, while the probability mass is concentrated on expected transitions, `INTERP-up` models are able to reach any transition in the vocabulary of transitions at any point in time. This is not the case for `BO-up` models.

`down` corresponds to the unnatural distribution. For example, suppose we are given the same conditional estimates θ. For `BO` models, `down` will normalize the estimates, sort the tokens according to their probabilities and then reverse the probabilities over the tokens to produce the following distribution: $\{p(w_c|h) : 0.55, p(w_d|h) : 0.22, p(w_a|h) : 0.11, p(w_b|h) : 0.11\}$. In cases where two tokens have the same probability, `down` will randomly arrange the tokens, so $\{p(w_c|h) : 0.55, p(w_d|h) : 0.22\}$ and $\{p(w_c|h) : 0.22, p(w_d|h) : 0.55\}$ are equally likely. According to the training log, tokens w_c and w_d are unnatural given the context, but `down` seeks the unnatural transitions when it generates samples. For `INTERP` models, `down` will build a categorical distribution over $\mathcal{V}$ using Eq. (3). Therefore, as with `up`, we sort the tokens in $\mathcal{V}$ according to their probabilities and then reverse the probabilities over the sample space. `INTERP down` models marshal uncharacteristic transitions in $\mathcal{V}$ given the context into the natural space, replacing frequently observed transitions.

`strange` linearly interpolates `up` and `down`, which likely has the effect of increasing the entropy of $p(w|h)$. When the entropy of the natural distribution over tokens is low and the mixture coefficient $\lambda \approx 0.5$, `strange` distributions can take an interesting form where the probability mass is simultaneously concentrated on very (un)natural events. This strange case describes a scenario where the generative model produces a stream of natural tokens, yet at any point in time is "equally" likely to choose a corner case in the execution.

3) Generating Event Sequences: The generation engine estimates two n-gram language models—one `BO` and one `INTERP`— using the vocabulary extracted in the *Mine* phase. The default order and smoother for both models are three and modified Kneser-Ney [14], respectively. Tokens in the vocabulary that are not in the event-sequence repository are factored into the models' unigram estimates. Clients (i.e., instances of the *Chimp* component in Fig. 1) can communicate with the *Sequence Generator* by sending JSON requests and reading JSON responses. A request includes: model, flavor, history, and length. The model field can take one of two values: `BO` or `INTERP`. The flavor field can take one of three values: `up`, `down` or `strange`. The history field contains the initial prefix as an array of strings. If the client does not have a prefix, then the history field is the empty string. When the client does not have a history (or the length of the history is suboptimal), then the *Sequence Generator* will essentially bootstrap the history to length $n-1$, where n is the order of the model(s). When the client gives a history that is longer than n, the *Sequence Generator* will apply the Markov assumption, so the history will be sliced according to the order before querying the language model. The length field governs the length of the sequence to query from the language model (i.e., number of events in the test case). This enables us to use the *Sequence Generator* in serial mode (e.g., requesting a sequence with 100 events) or in interactive mode (i.e., only one event is requested at a time).

D. `Validate`: Filtering Actionable Scenarios

The streams of events generated by the *Sequence Generator* are expressed at GUI level, which decouples the sequence from device-specific locations. Thus, the *Chimp* component (Fig. 1-d) understands the events and `Validates` them dynamically on a device. The validation can be performed in a serial (as in Tonella et al. [49]) or with MONKEYLAB interactive mode.

In the serial mode, the *Chimp* requests a sequence that is validated iteratively following the procedure in Alg. 1. For each event e in the event sequence S, the *Chimp* searches for the component (in the event tuple) in the `Hierarchy View` (i.e., GUI tree) of the current GUI displayed in the target device. If the component is not in the GUI or the current activity is not the same as in the tuple e (line 5), then the *Chimp* discards the event (line 7); otherwise the *Chimp* queries the `View Server` (line 8) in the device to identify the location of the component. Afterward, it generates an `input` command by using the location $<x, y>$ of the component and the action in the tuple e (line 9). Then the

Algorithm 1: Validating Event Sequences: Serial Mode

Input: S
Output: AS
1 **begin**
2 $i = 1, ATC = \emptyset$;
3 $startAppInDevice()$;
4 **foreach** $e \in S$ **do**
5 $feasible = queryVS(e.component, e.action)$;
6 **if** *!feasible* **then**
7 $continue_with_next_event$;
8 $< x, y >= searchVS(e.component)$;
9 $cmd_i = getInputCmd(e.action, < x, y >)$;
10 $addEvent(AS, < e, cmd_i >)$;
11 $executeInputCmd(cmd_i)$;
12 $i = i + 1$;

Algorithm 2: Validating Sequences: Interactive Mode

Input: k
Output: AS
1 **begin**
2 $ATC = \emptyset, history = \emptyset$;
3 $startAppInDevice()$;
4 **for** $i \in 1 : k$ **do**
5 $e = queryEvent(history)$
$feasible = queryVS(e.component, e.action)$;
6 **if** $!feasible$ **then**
7 $e = getEvent(randComponent())$;
8 $< x, y >= searchVS(e.component)$;
9 $cmd_i = getInputCmd(e.action, < x, y >)$;
10 $addEvent(AS, < e, cmd_i >)$;
11 $history = e$;
12 $executeInputCmd(cmd_i)$;
13 $i = i + 1$;

event e and its corresponding input command are added to the actionable scenario (line 10). Finally, the command cmd_i is executed on the GUI to update the app state. This serial model is similar to the one proposed in [49], however we are proposing three different flavors for the interpolated *n*-grams and the back-off model (Sec. III-C1 and Sec. III-C2) and we automatically validate the sequences on a target device.

In the interactive mode (Alg. 2), the *Chimp* requests single events until the target length k is exhausted. Our motivation for this mode is the possibility of having infeasible events in a sequence (i.e., in serial model) that can influence further events. We are augmenting the vocabulary used to train the models, with individual tokens extracted statically from the source code. So, it is possible that unseen—but also infeasible—events appear in a sequence. For example, in the sequence $e_1 e_2 ... t_1 e_k e_{k+1} ... t_2 t_3 e_l e_{l+1}$, tokens t_1, t_2, and t_3 are infeasible, which leads to the following issues: (i) the inclusion of events e_k, e_{k+1}, e_l, e_{l+1} into the sequence is influenced by the infeasible tokens t_1, t_2, and t_3; and (ii) once the first infeasible token is read by the *Chimp*, the further events in the sequence can also be infeasible in the GUI. This is why, in the serial mode, we opted to skip infeasible events and continue reading until the sequence is exhausted, which drives to sequences with less events than the target k.

Therefore, as a second option, we opted for requesting single events and executing only feasible events. This is how the interactive mode operates. The *Chimp* in interactive mode asks for a single token (similarly to code suggestion problem [26]) until a target number of feasible events (lines 4 and 5 in Alg. 2) is executed on the device. To avoid infeasible events and loops because of chains of infeasible events (i.e., the returned token is always infeasible), we execute a random feasible event—queried from the current GUI state— (line 7) when the component and action in event e are not feasible (line 6). This guarantees that the *Chimp* always executes a feasible event, and that event is the history for the next one (line 11). The interactive mode relies on the language model extracted from the event logs and the source code, and takes advantage of the current GUI state to reduce the rate of infeasible-events produced by the model. In summary, our interactive mode combines static and dynamic analyses as well as GUI ripping.

IV. Empirical Study Design

Our main hypothesis is that models derived with MonkeyLab are able to produce event sequences for natural scenarios and corner cases unseen in the scenarios executed by stakeholders. Therefore, event sequences generated with MonkeyLab should not only include events from scenarios used to derive the models (i.e., scenarios collected from users) but also events not observed in those scenarios. To test our hypothesis, we performed a case study with five free Android apps and unlocked/rooted Nexus 7 [22] Axus tablets each having a 1.5GHz Qualcomm Snapdragon S4 Pro CPU and equipped with Android 4.4.2 (kernel version 3.4.0-gac9222c).

We measured statement coverage of sequences generated with (i) MonkeyLab, (ii) a random-based approach (i.e., `Android GUI monkey`), (iii) GUI ripping using a DFS heuristic, and (iv) manual execution. We also compared the GUI events in the sequences to identify events that were executed only by one approach when compared to another (e.g., GUI events in sequences from MonkeyLab but not in the users' traces). In summary, the study aimed at investigating the following research questions (RQs):

RQ$_1$: *Which language model-based strategy is more suitable for generating effective (un)natural scenarios?*
RQ$_2$: *Do scenarios generated with* MonkeyLab *achieve higher coverage as compared to* `Android UI monkey`*?*
RQ$_3$: *How do scenarios generated with* MonkeyLab *compare to a DFS-based GUI ripper in terms of coverage?*
RQ$_4$: *Do* MonkeyLab *scenarios achieve higher coverage than manual executions used to train the models?*

A. Data Collection

Tab. I lists the Android apps used in the study, lines of code (excluding third party libraries), number of Activities,

TABLE I
ANDROID APPS USED IN OUR STUDY. THE STATS INCLUDE THE NUMBER OF ACTIVITIES, METHODS, AND GUI COMPONENTS. LAST TWO COLUMNS LIST THE NUMBER OF RAW EVENTS (#RE) IN THE EVENT LOGS (I.E., LINES IN THE FILES COLLECTED WITH THE GETEVENT COMMAND) AND GUI LEVEL EVENTS MINED FROM THE RAW LOGS (#GE)

App	Ver.	LOC	#Act.	#M.	#Comp.	#RE	#GE
Car Report	2.9.1	7K+	6	764	142	23.4K+	1.5K+
GnuCash	1.5.3	10K+	6	1,027	275	14.7K+	895
Mileage	3.1.1	10K+	51	1,139	99	9.8K+	783
My Expenses	2.4.0	24K+	17	1778	693	20.3K+	854
Tasks	1.0.12	10K+	4	561	200	70.6K+	1.7K+

methods, and GUI components. We selected the applications looking for a diverse set in terms of events supported by the apps (`click`, `long click`, `swipe`), number of GUI components, number of Activities, and application domain. However, the main selection criteria was the app's size. We decided to use only medium-to-large applications that exhibit non-trivial use-cases and a large set of feasible events.

To answer the research questions, we simulated a system-level testing process where we asked graduate students to get familiarized with the apps for five minutes (exploratory testing), and then to test the app executing multiple scenarios for 15 minutes (functional testing). Five Ph.D. students at the College of William and Mary executed the `Record` phase of our MONKEYLAB framework. Afterward, we executed the `Mine`→`Generate`→`Validate` phases in MONKEYLAB to generate actionable sequences for each app. During the *Validate* phase, we collected coverage measurements for each of the generated event sequences by using a tailored version of Emma [3] for Android. Tab. I lists the total number of low-level and GUI-level events collected from the participants.

To represent a random approach, we generated sequences of touch events with `Android GUI monkey`, simulating test cases composed of random events. Regarding the GUI ripping approach, we implemented our own version of DFS-based exploration. In terms of GUI-model extraction/inference, it considers cases that were not captured in other tools [11], [39], such as pop-up windows in menus/internal windows, on-screen keyboard, and containers. In this case, we do not have a set of actionable sequences because the app is explored trying to visit as many clickable GUI components as posible. However, we measured the accumulated coverage for the whole systematic exploration. This measurement helped us to establish a baseline to validate whether our event sequence generation approach outperforms competitive approaches. More details of the DFS implementation are in our online appendix.

B. Design Space

For training the language models, we used the traces collected from the participants for the functional testing scenario. Given two LMs (*i.e.*, `INTERP` and `BO`) and three flavors (*i.e.*, `up`, `down`, `strange`), we generated 100 scenarios for each combination <*Model, Flavor, APP*> with *3*-gram LMs, then we generated actionable scenarios using the serial mode. For the interactive mode, we generated 33 scenarios for each of the three INTERP flavors. Each scenario (in serial and interactice modes) was composed of 100 events. We also executed `Android UI monkey` on each app 100 times with 100 touchable events and inter-arrival delay of 500ms. Finally, we executed our DFS-based GUI ripper on each app.

TABLE II
ACCUMULATED STATEMENT COVERAGE OF THE LMS

App	INTERP up	INTERP down	INTERP str.	BO up	BO down	BO str.	*I-LM*
Car Report	12%	14%	13%	11%	11%	11%	**30%**
GnuCash	7%	6%	7%	7%	7%	7%	**22%**
Mileage	**36%**	10%	25%	32%	24%	24%	26%
My Expenses	10%	10%	10%	10%	10%	10%	**26%**
Tasks	40%	34%	35%	22%	22%	34%	**42%**

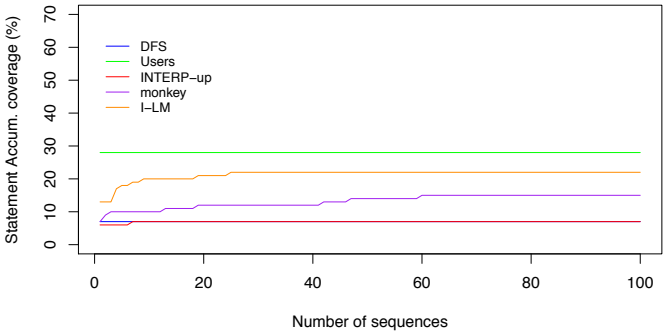

Fig. 2. Accumulated Coverage for GnuCash

V. RESULTS AND DISCUSSION

Fig. 2 shows cumulative coverage of MONKEYLAB and Android `UI Monkey` for the event sequences, in addition to coverage achieved by logs collected from human participants and coverage of the DFS-based exploration for the *GnuCash app*. The figures for the other apps are in our online appendix. In the case of MONKEYLAB, the figure depicts the accumulated coverage of the best LM in serial mode (in red color), and the combination of the 99 scenarios of `INTER-up`, `INTER-down`, and `INTER-strange` (33 scenarios for each strategy) in interactive mode (*I-LM*). In the following subsections, we present the results for the research questions defined in Sec. IV.

A. RQ_1: Language Models and Flavors

The accumulated coverage values for the language models in serial mode are listed in Tab. II. In general, the `INTERP-up` flavor is the one with the highest accumulated coverage for the serial mode. Apps with splash-screens and modal dialogs in clean launches (i.e., the app is launched right after being installed) such as *GnuCash*, *Car Report*, and *My Expenses* posed a challenge for the language models (see Sec. V-E for more details). The interactive mode (*I-LM*), outperforms the language models in serial model in 4 out of the 5 apps. The *I-LM* mode is able to deal with the problem of splash-screens and modal dialogs in clean launches to reduce the notoriously high rate of infeasible events.

The accumulated coverage provides a high-level measure when comparing two different models, however, it does not reveal the entire story. It is possible that different strategies are generating different events and sequences that drive to

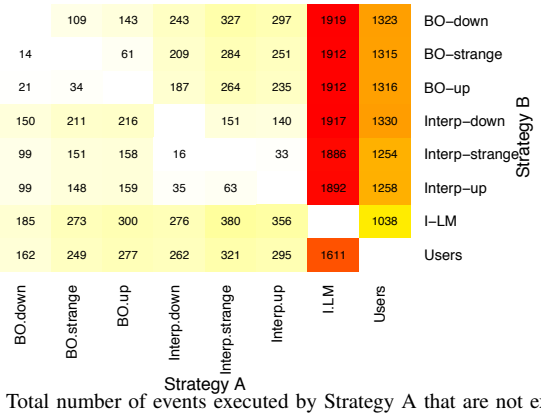

Fig. 3. Total number of events executed by Strategy A that are not executed by Strategy B. The key color goes from white (zero) to red (highest value)

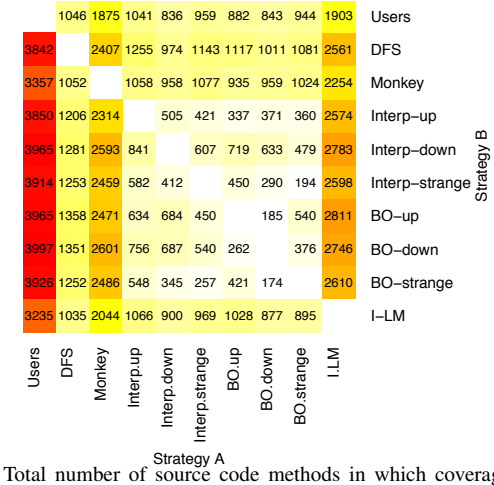

Fig. 4. Total number of source code methods in which coverage is higher when comparing coverage of Strategy A versus Strategy B

different executions of the apps. In fact, our goal behind proposing six different flavors of LMs and the interactive mode is based on the possibility that each flavor can explore different regions in the GUI event space. Therefore, we investigated the mutually exclusive events executed by each strategy in the five apps. We also measured the number of times the method coverage for each method in an app was higher in one strategy when doing pairwise comparisons (e.g., `INTER-up` versus `INTER-down`). The results are presented with heat-maps in Fig. 3 and Fig. 4. Both figures corroborate the fact that each LM strategy is able to generate different sets of events. For example, when comparing `INTERP-strange` to `BO-down` (Fig. 3), the former strategy was able to generate 327 GUI events that were not generated by the latter. In addition, the *I-LM* mode is the strategy with the highest difference of executed events when compared to the other strategies.

Summary for RQ$_1$: The LMs strategies are able to generate diverse and orthogonal sets of GUI events, and the interactive mode is the most effective.

B. *RQ$_2$:* MONKEYLAB *vs. Android* `UI monkey`

The coverage of `UI monkey` surprisingly outperformed the LMs in serial model for *Gnu Cash*, *Car Report*, and *MyExpenses*. LMs provided a better coverage only in *Tasks* and showed similar coverage in *Mileage*. The random nature of the events generated by the `UI monkey` allows this strategy to execute more GUI components without deeply exploring execution paths that are related to the use cases. For instance, `UI Monkey` is able to click on a diverse set of GUI components; however, it does so without context awareness of the execution history. The LMs are able to explore execution paths that lead to the activation of Activities and features that can not be reached easily by the `UI Monkey`. However, the *I-LM* mode outperforms the accumulated coverage of `UI monkey` in all the apps except for *Mileage*. When looking into the results in Fig. 4, it is clear how `UI monkey` is able to achieve higher coverage in source code methods more times than the serial LM-strategies. However, the *I-LM* is able to achieve higher coverage in more source code methods when compared to `Android UI monkey`.

Summary for RQ$_2$: `Android UI monkey` is able to achieve higher coverage than MONKEYLAB in serial model. However, our interactive mode outperforms `UI monkey`. In terms of scenarios, all the MONKEYLAB strategies are able to generate execution paths that are not covered by `UI monkey`.

C. *RQ$_3$:* MONKEYLAB *vs. DFS*

The LMs in a serial mode provided similar statement coverage as compared to DFS in `GnuCash`, `Tasks`, and `MyExpenses`, better coverage in `Mileage`, and lower in `Car Report`. DFS follows a systematic strategy for executing click events on the GUIs. This systematic exploration is able to exercise deeper executions paths as compared to `UI monkey`, but DFS is not able to recognize data dependencies. We found that the LMs can actually model some of these dependencies (e.g., some fields are required to create a budget or task). In other cases, the LMs can get stuck in specific paths that are very common (frequent) in the observed traces. However, the *I-LM* mode outperformed the accumulated coverage of DFS in the 5 apps. Concerning, the source code methods with higher coverage (Fig. 4), there is a notorious difference between DFS and the LMs in serial mode. However, the *I-LM* provides higher coverage on more methods when compared to DFS (*I-LM*−DFS = 2,561 and DFS−*I-LM* = 1,035 methods).

Summary for RQ$_3$: Although the LMs in serial mode are not able to achieve better coverage than DFS, *I-LM* is able to achieve higher coverage as compared to DFS.

D. *RQ$_4$:* MONKEYLAB *vs. Manual execution*

The coverage achieved by the users during the `Record` phase outperforms `UI Monkey`, DFS, and the LMs in serial and interactive modes, as originally expected. However, the LMs were able to execute events that were not observed in the event traces collected by the users (see Fig. 3). In general, the LMs were able to generate actionable scenarios for natural executions, i.e., there are sequences generated with the LMs including GUI events from the collected traces. However, the LMs were also able to generate sequences with unseen events, which means that the actionable scenarios cover natural cases but also corner-cases not considered by the users. And the benefit is noticeable when using the interactive mode, *which*

was able to generate 1,611 GUI events not considered by the participants in our study.

While we can not claim that the LMs are better than the other approaches in terms of coverage, after looking into the details of the events executed by each method, it becomes clear that the combinations of manual testing and automated approaches can significantly improve the coverage. However, MONKEYLAB is able to learn a model and then generate actionable scenarios that can be executed on a target device. In addition, MONKEYLAB generates not only natural scenarios but also corner cases that are not generated by any of the competitive approaches.

Summary for RQ$_4$: Although the overall coverage achieved by MONKEYLAB is not as high as compared to manual execution, MONKEYLAB is able to generate actionable scenarios including not only natural GUI events, but also events that are not considered during manual testing. Therefore, the scenarios generated by MONKEYLAB can help increase the coverage achieved by manual testing without extra effort that is imposed by collecting test-scripts.

E. Limitations

Automated GUI-based testing approaches can benefit from the following information in this study, since some of these issues can pose similar problems for various testing *strategies*.

Encapsulated components. Encapsulated components (e.g., `KeyboardView`, `AutoCompleteTextView`, `CalendarView`, and `DatePicker`) cannot be analyzed during systematic exploration, because the sub-components (e.g., each key of the keyboard) are not available for ripping at execution time. Thus, the sub-components are not recognized by the `View Server` nor the `Hierarchy Viewer`. This type of component requires a predefined model of the feasible events with the locations and areas of these sub-components.

Training corpus size in serial mode. Another lesson gleaned from this study concerns an effective training set. Specifically, we diagnosed two critical issues that contributed to the high rate of infeasible events produced by the interpolated n-grams and back-off models. The principal issue was relatively small amount of training data for each app as compared to the size of the respective vocabularies. The need for substantial training data was observed in most of the sequences for the apps (especially *Keepscore*) for which the LM-based strategies were not able to produce a response. For example, an app with a vocabulary size of 250 events would require 250×249 and 250×250×249 parameters to reliably estimate bigram and trigram models, respectively. Data sparsity is a key concern in statistical language modeling, and a number of techniques have been developed to manage the problem of deriving useful statistical estimates from relatively small corpora [14]. Naturally, one approach to controlling the number of parameters is to reduce the order of the model [41]; however, reducing the order of our models presented another issue. In our experiments, reducing the order degenerated the effectiveness of the `INTERP-up` models at generating feasible test cases. This degeneration can be attributed to allocating too much weight to the unigrams in the mixture model (Eq. 3). In other words, after unfolding the recurrence in Eq. 3, unigrams will generally have more influence on the predictions of interpolated n-gram models as compared to interpolated models whose order is greater than n. For example, unigrams will have more influence on the predictions of interpolated trigrams than interpolated 4-grams, because the unigrams are multiplied by $\beta(h)$ (Eq. 3), a number less than one, an additional time in interpolated 4-grams. Our expectation was for `INTERP-up` models to yield favorable coverage results in line with previous empirical studies [49], yet the low-orders disabled the models in our experiments. However, we did not expect `INTERP-down` nor `INTERP-strange` models to yield high rates of feasible events. Considering the nature of the interpolation smoothing technique, models at *all* orders are factored into the estimator. So, regardless of the order of the interpolated model and the history of events, every event will have a chance to be selected at each point in time. Moreover, for `INTERP-down` and `INTERP-strange`, the probability distributions are transformed such that *all* of the unlikely events (given the context) in the vocabulary are generally assigned more probability mass. Thus, we expected `INTERP-down` and `INTERP-strange` models to yield high rates of infeasible events. The sparse training set had a similar degenerative effect on the back-off models. With relatively few data points to inform high-order component models, the back-off model will successively probe low-order models until reaching unigrams—the base case. Probing low-order models is not conducive to generating feasible events. The purpose of smoothing techniques like interpolation and back-off is to use substantial contexts when possible before defaulting to lower-order models (e.g., unigrams and bigrams), but this is predicated on a training set that can support the estimation of higher-order models.

VI. CONCLUSION AND FUTURE WORK

We present a novel framework for automatic generation of actionable scenarios for Android apps. The framework uses two language models and three proposed flavors. For the evaluation we used several medium-large Android apps. On one hand, the results suggest that the `BO` model is more suitable for generating unnatural scenarios with small and large corpora. On the other hand, `INTERP` is not able to generate scenarios when the available corpora are small. If small corpora are available, `INTERP` is more suitable for natural scenarios. Finally, the interactive mode in MONKEYLAB outperformed the serial mode in terms of coverage.

We also provide a set of learned lessons for GUI-based testing of Android apps using actionable scenarios. The lessons recommend designing models for encapsulated components and increasing training corpus size of LMs. The lessons can be used by researchers to implement automated approaches for GUI-based testing that are more attractive and useful to mobile developers and testers. Actionable scenarios can also be combined with automatic generation of testing oracles [35] for generating GUI-based test cases for mobile apps.

REFERENCES

[1] Apktool. https://code.google.com/p/android-apktool/.

[2] dex2jar. https://code.google.com/p/dex2jar/.

[3] Emma: a free java code coverage tool. http://emma.sourceforge.net/.

[4] Robotium. https://code.google.com/p/robotium/.

[5] Sending keyboard input via adb to your android device. http://blog.rungeek.com/post/42456936947/sending-keyboard-input-via-adb-to-your-android.

[6] Srcml. http://www.sdml.info/projects/srcml/.

[7] P. Aho, M. Suarez, T. Kanstren, and A. Memon. Murphy tools: Utilizing extracted gui models for industrial software testing. In *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, pages 343–348, 2014.

[8] D. Amalfitano, A. Fasolino, P. Tramontana, S. De Carmine, and A. Memon. Using gui ripping for automated testing of android applications. In 258-261, editor, *International Conference on Automated Software Engineering (ASE'12)*, 2012.

[9] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. Memon. Mobiguitar - a tool for automated model-based testing of mobile apps. *IEEE Software*, page to appear, 2014.

[10] S. Anand, M. Naik, M. J. Harrold, and H. Yang. Automated concolic testing of smartphone apps. In *FSE'12*, 2012.

[11] T. Azim and I. Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 641–660, 2013.

[12] G. Bavota, M. Linares-Vásquez, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. The impact of api change- and fault-proneness on the user ratings of android apps. *IEEE Transactions on Software Engineering (TSE)*, 2015.

[13] C. C. K. K. C. Yeh, H. Lu and S. Huang. Craxdroid: Automatic android system testing by selective symbolic execution. In *IEEE Eighth International Conference on Software Security and Reliability (SERE-C)*, pages 140–148, 2014.

[14] S. F. Chen and J. Goodman. An empirical study of smoothing techniques for language modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, ACL '96, pages 310–318, Stroudsburg, PA, USA, 1996. Association for Computational Linguistics.

[15] W. Choi, G. Necula, and K. Sen. Guided gui testing of android apps with minimal restart and approximate learning. In *International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA'13)*, pages 623–640, 2013.

[16] K. Church and W. Gale. A comparison of the enhanced good-turing and deleted estimation methods for estimating probabilities of english bigrams. *Computer Speech and Language*, 5:19–54, 1991.

[17] V. Dallmeier, N. Knopp, C. Mallon, G. Fraser, S. Hack, and A. Zeller. Automatically generating test cases for specification mining. *Software Engineering, IEEE Transactions on*, 38(2):243–257, 2012.

[18] A. Dias-Neto, R. Subramanyan, M. Vieira, and G. Travassos. A survey on model-based testing approaches: a systematic review. In *International Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, pages 31–36, 2007.

[19] S. Elbaum, G. Rothermel, S. Karre, and M. Fisher II. Leveraging user-session data to support web application testing. *IEEE Trans. Softw. Eng.*, 31(3):187–202, Mar. 2005.

[20] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein. Reran: Timing- and touch-sensitive record and replay for android. In *International Conference on Software Engineering (ICSE'13)*, pages 72–81, 2013.

[21] I. Good. The population frequencies of species and the estimation of population parameters. *Biometrika*, 40(3/4):237–264, 1953.

[22] Google. Nexus 7. http://www.google.com/nexus/7/.

[23] Google. Ui/application exerciser monkey. http://developer.android.com/tools/help/monkey.html.

[24] M. Grechanik, Q. Xie, and C. Fu. Experimental assessment of manual versus tool-based maintenance of gui-directed test scripts. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 9–18, Sept 2009.

[25] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and W. Strouila. Understanding Android fragmentation with topic analysis of vendor-specific bugs. In *19th Working Conference on Reverse Engineering*, pages 83–92, 2012.

[26] A. Hindle, E. Barr, Z. Su, M. Gabel, and P. Devanbu. On the naturalness of software. In *International Conference on Software Engineering (ICSE'12)*, pages 837–847, 2012.

[27] B.-J. P. Hsu. *Language Modeling for Limited-data Domains*. PhD thesis, Cambridge, MA, USA, 2009. AAI0822209.

[28] G. Hu, X. Yuan, Y. Tang, and J. Yang. Efficiently, effectively detecting mobile app bugs with appdoctor. In *Ninth European Conference on Computer Systems (EuroSys'14)*, page Article No.18, 2014.

[29] F. Jelinek and R. L. Mercer. Interpolated estimation of Markov source parameters from sparse data. In *In Proceedings of the Workshop on Pattern Recognition in Practice*, pages 381–397, Amsterdam, The Netherlands: North-Holland, May 1980.

[30] C. S. Jensen, M. R. Prasad, and A. Moller. Automated testing with targeted event sequence generation. In *International Symposium on Software Testing and Analysis (ISSTA'13)*, pages 67–77, 2013.

[31] N. Jones. Seven best practices for optimizing mobile testing efforts. Technical Report G00248240, Gartner, 2013.

[32] M. E. Joorabchi, A. Mesbah, and P. Kruchten. Real challenges in mobile apps. In *ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*, pages 15–24, 2013.

[33] S. M. Katz. Estimation of probabilities from sparse data for the language model component of a speech recognizer. In *IEEE Transactions on Acoustics, Speech and Signal Processing*, pages 400–401, 1987.

[34] H. Khalid, M. Nagappan, E. Shihab, and A. Hassan. Prioritizing the devices to test your app on: A case study of android game apps. In *22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE 2014)*, 2014.

[35] Y.-D. Lin, J. Rojas, E. Chu, and Y.-C. Lai. On the accuracy, efficiency, and reusability of automated test oracles for android devices. *IEEE Transactions on Software Engineering*, Preprint, 2014.

[36] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: A threat to the success of android apps. In *9th Joint Meeting of the European Software Engineering Conference and the 21st ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'13)*, pages 477–487, 2013.

[37] M. Linares-Vásquez, G. Bavota, M. D. Penta, R. Oliveto, and D. Poshyvanyk. How do API changes trigger Stack Overflow discussions? a study on the android SDK. In *22nd IEEE International Conference on Program Comprehension (ICPC'14)*, pages 83–94, 2014.

[38] M. Linares-Vásquez, M. White, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk. Mining actionable scenarios for android apps using language models - online appendix. http://www.cs.wm.edu/semeru/data/MSR15-MonkeyLab/.

[39] A. Machiry, R. Tahiliani, and M. Naik. Dynodroid: An input generation system for android apps. In *9th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'13)*, pages 224–234, 2013.

[40] R. Mahmood, N. Mirzaei, and S. Malek. Evodroid: Segmented evolutionary testing of android apps. In *FSE'14*, page to appear, 2014.

[41] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, Cambridge, Massachusetts, 1999.

[42] T. McDonnell, B. Ray, and M. Kim. An empirical study of api stability and adoption in the android ecosystem. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 70–79, Sept 2013.

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

[44] B. Nguyen and A. Memon. An observe-model-exercise* paradigm to test event-driven systems with undetermined input spaces. *IEEE Transactions on Software Engineering*, 99(Preprints), 2014.

[45] L. Ravindranath, S. nath, J. Padhye, and H. Balakrishnan. Automatic and scalable fault detection for mobile applications. In *12th annual international conference on Mobile systems, applications, and services (MobiSys'14)*, pages 190–203, 2014.

[46] A. Rountev and Y. Dacong. Static reference analysis for gui objects in android software. In *IEEE/ACM International Symposium on Code Generation and Optimization*, 2014.

[47] M. Strobel. Procyon. https://bitbucket.org/mstrobel/procyon.

[48] T. Takala, M. Katara, and J. Harty. Experiences of system-level model-based gui testing of an android application. In *Fourth International Conference on Software Testing, Verification and Validation (ICST'11)*, pages 377–386, 2011.

[49] P. Tonella, R. Tiella, and C. Nguyen. Interpolated n-grams for model based testing. In *International Conference on Software Engineering (ICSE'14)*, 2014.

[50] VisionMobile. Developer economics q3 2013: State of the developer nation, 2013.
[51] VisionMobile. Developer tools: The foundations of the app economy (developer economics 2013), 2013.
[52] VisionMobile. Developer economics q1 2014: State of the developer nation. Technical report, 2014.
[53] C.-Y. Wang, W.-C. Chu, H.-R. Chen, C.-Y. Hsu, and M. Y. Che. Evertutor: Automatically creating interactive guided tutorials on smartphones by user demonstration. In *SIGCHI Conference on Human Factors in Computing Systems (CHI'14)*, pages 4027–4036, 2014.
[54] W. Yang, M. Prasad, and T. Xie. A grey-box approach for automated gui-model generation of mobile applications. In *16th International Conference on Fundamental Approaches to Software Engineering (FASE'13)*, pages 250–265, 2013.