AVGUST: A Tool for Generating Usage-Based Tests from Videos of App Executions

Saghar Talebipour[1*], Hyojae Park[1†], Kesina Baral[‡], Leon Yee[§],
Safwat Ali Khan[‡], Kevin Moran[‡], Yuriy Brun[¶], Nenad Medvidovic[*], Yixue Zhao[¶]
[*]*University of Southern California* (USA), [†]*Sharon High School* (USA), [‡]*George Mason University* (USA)
[§]*Valley Christian High School* (USA) [¶]*University of Massachusetts Amherst* (USA)
talebipo@usc.edu, hyoj.p20@gmail.com, kbaral4@gmu.edu, leon.yee000@gmail.com,
skhan89@gmu.edu, kpmoran@gmu.edu, brun@cs.umass.edu, neno@usc.edu, yixuezhao@cs.umass.edu

***Abstract*—Creating UI tests for mobile applications is a difficult and time-consuming task. As such, there has been a considerable amount of work carried out to automate the generation of mobile tests—largely focused upon the goals of maximizing code coverage or finding crashes. However, comparatively fewer automated techniques have been proposed to generate a highly sought after type of test: *usage-based tests*. These tests exercise targeted app functionalities for activities such as regression testing. In this paper, we present the AVGUST tool for automating the construction of usage-based tests for mobile apps. AVGUST learns usage patterns from videos of app executions collected by beta testers or crowd-workers, translates these into an app-agnostic state-machine encoding, and then uses this encoding to generate new test cases for an unseen target app. We evaluated AVGUST on 374 videos of use cases from 18 popular apps and found that it can successfully exercise the desired usage in 69% of the tests. AVGUST is an open-source tool available at https://github.com/felicitia/UsageTesting-Repo/tree/demo. A video illustrating the capabilities of AVGUST can be found at: https://youtu.be/LPICxVd0YAg.**

***Index Terms*—Mobile Application, UI Understanding, Mobile Testing, Test Generation, AI/ML**

I. INTRODUCTION

UI testing, which tests the quality of an app by triggering a sequence of GUI interactions, is a critical tool for successfully developing high quality software. There exists a large body of research on automatically generating UI tests that mostly focus on quantitative metrics such as code coverage [1]–[3]. However, studies show that *usage-based* UI tests are highly preferred by developers in practice [4]. A usage-based test is defined as a UI test that exercises a specific functionality of an app, such as "sign in" or "add item to the shopping cart" [5], [6]. These tests mimic realistic end-user behaviors in relation to the specific usages, but developers have to write them manually [4], [7]. Unfortunately, manually creating usage-based tests is time-consuming and error-prone, and hiring real users to interact with the app can be expensive. Therefore, automating the process of generating usage-based tests is highly sought after.

Recently, the emerging topic of UI test reuse has offered a promising solution to generating usage-based tests by transferring tests from one app (the source app) to another app (the target app) [5], [8]–[15]. However, this method poses three limitations: (1) UI test reuse requires *pre-existing tests* for the source app that can be hard to obtain in practice; (2) UI test reuse relies on the similarities between the source app and the target app, and requires them to be in the same domain (e.g., transferring between two shopping apps). (3) Lastly, UI test reuse often depends on program analyses that require the source code of the app, which may not always be accessible.

To overcome these limitations, we designed and implemented AVGUST, a tool for app-video-based generation of usage tests. The **software engineering challenge addressed** by AVGUST is the automated generation of usage-based tests for mobile apps, and hence the **envisioned users** are mobile app developers and testers. AVGUST's **usage methodology** is described in Section II-D. AVGUST is able to semi-automatically synthesize tests by first learning an app-agnostic state machine representation of feature-based usages from app videos, and then uses this representation to synthesize tests for a new, unseen target application using various deep learning and computer vision techniques. As such, AVGUST does not require pre-existing tests, is app-agnostic, and has the ability to learn from diverse apps across different domains. Furthermore, AVGUST does not require access to an app's source code (which might not always be available), instrumentation, or program analysis. Instead, AVGUST only relies on *visual information* extracted from app videos, which are common artifacts, and can be easily obtained by crowd workers with no testing expertise. Moreover, AVGUST's design is not tied to a specific mobile platform such as Android (upon which it is evaluated), but is applicable to any device or platform in principle.

To demonstrate AVGUST, we collected 374 video recordings of common app usages from 18 apps in order to learn 125 intermediate-representation Models (IR Models). These IR Models can be used directly by developers to generate tests for the usages targeted in this paper, or can be extended to new usages through video collection and processing. We **empirically evaluated** AVGUST from two perspectives: the quality of the generated tests, and the accuracy of its underlying screen and widget classification techniques in generating the IR Models. Using 374 video recordings, AVGUST was tasked with generating 51 tests across 18 types of app usages for 18 target apps. We found that 69% of the tests created by AVGUST successfully exercised the desired usage. Furthermore, AVGUST's

[1]First & second authors contributed equally to the paper.

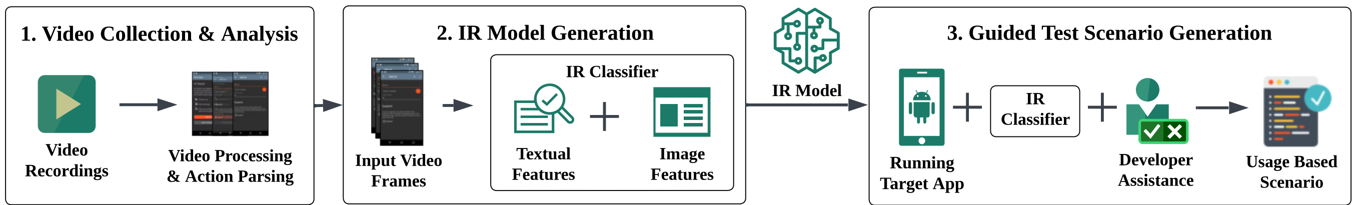

Fig. 1. An illustration and overview of AVGUST's three-phase workflow.

underlying screen and widget classification techniques achieved high accuracy across the board and consistently outperformed the state-of-the-art. All of AVGUST's artifacts including its source code and experimental data is publicly available [16].

II. THE AVGUST TEST GENERATION TOOL

AVGUST is a developer-in-the-loop tool that uses AI techniques to generate usage-based tests by learning from app videos. As shown in Figure 1, AVGUST consists of three main phases: (1) *Video Collection & Analysis*, (2) *IR Model Generation*, and (3) *Guided Test Scenario Generation*. More comprehensive technical details related to each of the phases are described in our research paper [6].

First, in the *Video Collection & Analysis* phase AVGUST processes videos using computer vision techniques to identify non-redundant video frames where touch actions occur. Next, in the *IR Model Generation* phase, screens with touch actions are classified into categories (e.g., "search screen") and exercised widgets are classified into canonical types (e.g., "menu button"). Then an app agnostic state machine for the input video usage is encoded using the categories. Finally, in the *Guided Test Scenario Generation* phase the encoded state-machine is used to guide a developer through the test generation process by suggesting actions and paths through the app that accomplish the target usage. We next describe each of the phases in detail.

A. Video Collection & Analysis

Given the collected videos of app usages, AVGUST automatically identifies key frames with user interactions and filters out the frames that contain sensitive information, such as user passwords. To identify the key frames with user actions, AVGUST extends V2S [17], a technique that leverages neural object detection and image classification to identify the user actions in a video, and translates these actions into a replayable scenario. Our extended V2S module stores the necessary information of the key steps in each video, including a sequence of *event frames* (key frames containing unique user actions), the respective action type (i.e., click, long tap, and swipe), and the action location on the screen (e.g., the touch coordinates).

To remove the sensitive frames, AVGUST identifies the frames that are associated with the *typing* action, since they may expose user's private information such as a password. To filter these frames, we trained a binary image classifier using a simple convolutional neural net (ConvNet) architecture that classifies whether a video frame contains the keyboard. In the end, AVGUST automatically eliminates the frame if it contains the keyboard and the location of the user action is within the area of the keyboard.

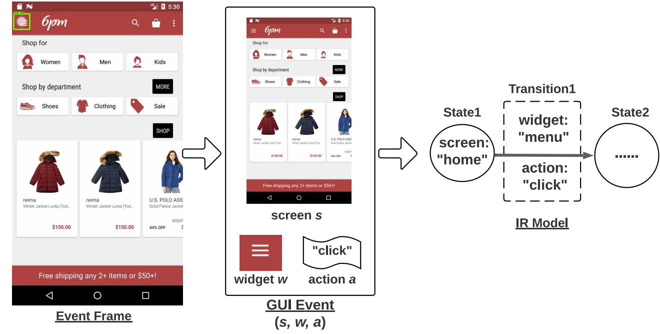

Fig. 2. An example of converting a 6pm's Event Frame into an IR Model.

By filtering out unnecessary and sensitive frames, AVGUST effectively translates videos into a compact dataset that retains sufficient information to learn each usage scenario, allowing the generation of various IR Models, as described next.

B. IR Model Generation

Using the cleaned event frames from the first phase, AVGUST generates IR Models (e.g., an example of IR Model can be seen from Figure 3) iteratively, by converting each *event frame* into a *GUI event*, and finally into one step in the IR Model as shown in Figure 2. As an illustration, we now describe how AVGUST translates one event frame into its respective step in the IR Model. The entire IR Model is constructed by repeating the same process.

First, an event frame is converted into a GUI event triple, which consists of the app *screen* that shows a snapshot of the app's execution state, the GUI *widget* that the user interacts with, and the *action* performed by the user (e.g., click, swipe). The *screen* and *action* are easily extracted by reusing the outputs from the first phase, while extracting the *widget* poses technical challenges as explained below.

To extract the widget from the event frame, AVGUST needs to first detect the bounding boxes of the GUI widgets in the screen, and then pinpoints which widget the user interacts with. To do so, we extended UIED [18], a state-of-the-art tool capable of detecting the bounding boxes of the widgets by using textual and visual information from the screen. However, UIED detects textual widgets and visual widgets separately, which can cause the widgets to lose important contextual information. For instance, the visual widget checkbox will be detected without its textual label, making it difficult to understand the

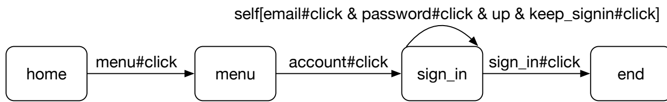

Fig. 3. An example IR Model containing the sign-in usage of the app 6pm.

semantics of such widgets. We thus extended UIED to group the visual widget and its corresponding textual information (if any) together based on their spatial information. With the detected bounding boxes, AVGUST identifies the widget that the user interacts with by calculating the proximity between each widget and the coordinates of the user's touch (obtained in the first phase as described earlier). The closest widget is chosen and automatically cropped out from the screen.

Next, each GUI event triple (*screen*, *widget*, *action*) is translated into its corresponding step in the IR Model, which is an app-agnostic representation of a particular usage in the form of a finite state machine (FSM). Figure 3 shows an example IR Model containing a sign-in usage converted from the app 6pm. Each state in the IR Model represents a particular app screen and is captured as an app-independent canonical screen, while each transition represents a user interaction with a canonical widget and its corresponding action. A self-transition (e.g., shown in the "sign_in" state in Figure 3) means that the app stays on the same screen during certain user interactions.

Specifically, the *action* in the GUI event triple stays the same in the IR Model, while the *screen* and *widget* are classified into their canonical categories using a Screen Classifier and a Widget Classifier that we developed. The canonical categories are pre-defined by refining prior work [5], resulting in 37 canonical screens and 74 canonical widgets that can be found in our repository [16]. Note that AVGUST's classifiers solely rely on pixel-based visual information extracted from app videos without the app's source code or UI hierarchy information. This poses a unique technical challenge that has not been addressed before. We now explain how AVGUST addresses this challenge using its novel Screen and Widget Classifiers, and a more in-depth discussion can be found in our full paper [6].

AVGUST's Screen Classifier leverages both visual information, encoded using a pre-trained autoencoder model we developed, and textual information retrieved from the screen using the Tesseract OCR engine and encoded using the BERT language model [19]. Specifically, our autoencoder is trained on 33,000 abstracted UI screenshots from the RICO dataset [20]. In this scenario we define an *abstracted image* to be a screenshot wherein textual components are drawn as yellow boxes and non-textual components are drawn as blue boxes on a black canvas that matches the size of the UI screenshot. To classify an incoming screen, we run it through this abstraction process, and then through the encoder of our autoencoder network to extract a feature vector.

The Screen Classifier employs linear layers to combine the autoencoder and BERT embeddings and classify the screens. The architecture consists of three blocks, each containing a linear layer, BatchNorm, a ReLU activation function, and a dropout layer. These blocks are followed by a fully connected output layer that applies softmax function to predict the probability distribution of different screen classes. To train AVGUST's Screen Classifier, we used the partitions of data collected for our evaluation (see Section III), where individual classifiers for each app were trained on data sourced from other apps, demonstrating AVGUST's capability to classify screens for *unseen* apps.

AVGUST's Widget Classifier uses five types of information extracted from the widget images as the features: (1) The visual information of the widget is encoded using a pre-trained ResNet model [21]; (2) The textual information is detected using Tesseract and encoded using the BERT model; (3) The canonical screen that the widget belongs to is mapped to an `id` and transformed into a continuous vector via an embedding layer; (4) The type of the widget (e.g., `EditText`) is classified using ReDraw [22], and is translated into an embedding; (5) The widget's location on the screen is obtained by dividing the screen into 9 zones and then transformed to an embedding.

Similarly, AVGUST's Widget Classifier adapts the same architecture as the Screen Classifier, and is trained on our dataset (see Section III). In the end, pre-trained Screen Classifiers and Widget Classifiers are produced for each app under test (AUT) using the remaining data from other apps.

Finally, AVGUST uses its screen and widget classifiers to recommend top-k canonical categories for developers to refine. This process converts each GUI event triple discussed earlier into its corresponding step in the IR Model. As a result, the entire IR Model is generated after iterating through each GUI event triple. Note that the IR labels refined by developers and the generated IR Models can be reused by future work.

C. Guided Test Scenario Generation

In order to generate a test scenario for a particular usage, AVGUST uses the IR Models generated in the previous phase, and follows a developer-in-the-loop process that reuses the Screen Classifier and Widget Classifier discussed earlier. Internally, IR Models of the same usage are represented as a single merged model where multiple scenarios for a given usage populate the same state machine. In this way, AVGUST is able to learn "all possible transitions" for a usage from a variety of executions, and generate multiple tests for that usage.

Specifically, this test scenario generation phase is iterative: each step of the test is generated based on the AUT's current state, until the end condition is met. The process begins by launching the AUT and running AVGUST's Screen Classifier to retrieve the most likely canonical categories of the AUT's starting screen. AVGUST then presents the developer with these top-k classification results obtained by its Screen Classifier, and the canonical category selected by the developer is used to match the AUT's current screen to the canonical screen states in the IR Model. Next, AVGUST recommends the top-k app widgets for developers to interact with by using its Widget Classifier to map the canonical widgets in the IR Model to the widgets on the AUT's current screen. After a widget is chosen by the developer, AVGUST checks whether the test should terminate based on whether the chosen widget will lead

to the end state in the IR Model. If not, the chosen widget is triggered, the AUT's next state becomes its current state, and this process repeats until the end state in the unified IR Model for a given usage is reached.

D. AVGUST*'s Implementation & Usage Methodology*

AVGUST is implemented in Python with 10,700 SLOC, of which the Screen and Widget Classifiers are stand-alone modules totaling 2,800 SLOC, and include the autoencoder model we developed. AVGUST employs the `pytransitions` library [23] to manipulate its state-machine IR Models, and uses the Appium testing framework [24] for its test generation.

AVGUST consists of two command line utilities, one for processing videos into the IR models, and another for generating a usage-based test of a target app. For the first utility, a developer simply provides a video labeled with a given usage name, and AVGUST processes the video into the appropriate IR Model, asking for developer feedback when necessary. For the second utility, which is illustrated in our demo video, the developer launches an emulator, and then executes AVGUST from the command line, and follows the prompts to generate the desired test for a specified usage.

III. EVALUATION

We evaluated AVGUST according to two criteria: (i) the *effectiveness* of its generated tests, and (ii) the *accuracy* of its Screen and Widget Classifiers. To do so, 374 video recordings of 18 common usages across 18 subject apps were collected by 31 computer science students in a Master's level course at George Mason University (GMU). This data collection study was reviewed by the GMU Institutional Review Board (IRB #1666261-1). A complete list of the apps and usage scenarios can be found in our full paper [6].

To evaluate AVGUST's *effectiveness*, we randomly selected 3 apps under each of the 18 usage scenarios as the AUTs to generate the tests for. We used the remaining apps' videos as the training data to generate the IR Models to guide the test generation. Four of the authors served as the developers interacting with the tool during this process. In the end, 51 tests are generated by AVGUST across 18 app usages. As a result, 35 of the 51 tests (68.6%) carried out the usage, meaning that AVGUST successfully generated a correct test to exercise the desired usage. We also inspected the 16 remaining tests, and measured how similar those tests are compared to the corresponding user interactions indicated in the collected app videos (exercised by humans to serve as the ground truth). On average, 79% of the states and 47% of the transitions in the generated tests are the same as the steps captured in the app videos. This indicates that AVGUST is successful in generating correct tests a large portion of the time, and can generate tests similar to the steps exercised by humans in the remaining cases, directly saving developers time and effort.

The *accuracy* of AVGUST's Screen and Widget Classifiers are evaluated in two contexts: (1) when used as stand-alone image classifiers, and (2) when used in AVGUST's test generation phase. In the first context, we annotated 2,478 screens and 2,434 widgets across 18 apps in our dataset to serve as the ground-truth labels. We tasked AVGUST's screen and widget classifiers and the state-of-the-art tool Screen2Vec (S2V) [25] in classifying screens and widgets. The results show that our classifiers constantly outperforms S2V, with an average Top-5 accuracy of 81% in the Screen Classification, and 76% in the Widget Classification. In the second context, we recorded Avgust's Top-1 and Top-5 recommendations at each step during the Test Generation phase, and evaluated the accuracy of those recommendations. We also compared the original vision-only classifiers of AVGUST with an extended version that uses runtime information, such as the dynamic UI hierarchy information of the app screen for this context. Interestingly, our results show that adding runtime features decreases accuracy, perhaps because these features are too different from the ones used to train AVGUST's vision-only models. Detailed evaluation results and discussion can be found in our paper [6].

IV. MOST CLOSELY RELATED EXISTING TOOLS

Liu et.al. introduced the NaviDroid tool [26], which provides testers with hint moves for the purpose of reaching unexplored regions of a given application. Hu et.al. introduced the AppFlow technique [27], which uses machine learning to help developers write tests that can be transferred between apps. Finally, researchers have developed a number of techniques for test case transfer [5], [8]–[15], which utilize a variety of AI and program analysis techniques.

AVGUST has three major advances over the prior work described above that highlight its novelty: (i) AVGUST operates purely on videos as input, which allows for easy crowd-sourcing of usage-based tests, unlike AppFlow which requires developers to manually record tests; (ii) AVGUST is geared toward aiding developers in generating usage-based tests, which studies have illustrated developers prefer over optimizing for code coverage [4], which is NaviDroid's goal; (iii) AVGUST does not require there to be similarities between apps for test generation, as much of the past work on test transfer requires.

V. CONTRIBUTIONS AND FUTURE WORK

We have presented AVGUST, a developer-in-the-loop tool that generates usage-based tests by learning from videos of app executions. AVGUST leverages novel computer vision techniques (e.g., image classification, object detection) to extract app-agnostic IR Models from videos, and uses them to recommend actions for developers during the test generation. AVGUST also provides ready-to-use models trained on our dataset for developers to use directly [16]. In the future, we aim to improve the accuracy of the image classification to further automate AVGUST's recommendation system, and build an infrastructure to facilitate community efforts in this area.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants no. CCF-1763423, CCF-1955853, CCF-2210243, CCF-2106871, and CCF-2030859 (to the Computing Research Association for the CIFellows Project). Additionally, we would like to thank Arthur Wu for his help on data collection.

References

[1] K. Li and M. Wu, *Effective GUI testing automation: Developing an automated GUI testing tool*. John Wiley & Sons, 2006.

[2] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: automated remote ui testing on android," in *MUM'12*, pp. 1–4.

[3] 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 *FSE'17*, pp. 245–256.

[4] M. Linares-Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *ICSME'17*. IEEE, pp. 613–622.

[5] Y. Zhao, J. Chen, A. Sejfia, M. Schmitt Laser, J. Zhang, F. Sarro, M. Harman, and N. Medvidovic, "Fruiter: a framework for evaluating ui test reuse," in *ESEC/FSE'20*, pp. 1190–1201.

[6] Y. Zhao, S. Talebipour, K. Baral, H. Park, L. Yee, S. A. Khan, Y. Brun, N. Medvidovic, and K. Moran, "Avgust: Automating usage-based test generation from videos of app executions," in *ESEC/FSE'22*, Singapore.

[7] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet?(e)," in *ASE'15*. IEEE, pp. 429–440.

[8] F. Behrang and A. Orso, "Test migration for efficient large-scale assessment of mobile app coding assignments," in *ISSTA'18*, pp. 164–175.

[9] ——, "Test migration between mobile apps with similar functionality," in *ASE'19*. IEEE, pp. 54–65.

[10] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *ESEC/FSE'18*, pp. 269–282.

[11] J.-W. Lin, R. Jabbarvand, and S. Malek, "Test transfer across mobile apps through semantic mapping," in *ASE'19*. IEEE, pp. 42–53.

[12] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of gui events for test reuse: are we there yet?" in *ISSTA'21*, pp. 177–190.

[13] L. Mariani, M. Pezzè, V. Terragni, and D. Zuddas, "An evolutionary approach to adapt tests across mobile apps," in *AST'21*. IEEE, pp. 70–79.

[14] X. Qin, H. Zhong, and X. Wang, "Testmig: Migrating gui test cases from ios to android," in *ISSTA'19*, pp. 284–295.

[15] S. Talebipour, Y. Zhao, L. Dojcilović, C. Li, and N. Medvidović, "Ui test migration across mobile platforms," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 756–767.

[16] "Avgust's public repository," 2022. [Online]. Available: https://github.com/felicitia/UsageTesting-Repo/tree/demo

[17] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *ICSE'20*, pp. 309–321.

[18] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "Uied: a hybrid tool for gui element detection," in *ESEC/FSE'20*, pp. 1655–1659.

[19] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*.

[20] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A mobile app dataset for building data-driven design applications," in *UIST'17*, pp. 845–854.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *CVPR'16*, pp. 770–778.

[22] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine learning-based prototyping of graphical user interfaces for mobile apps," *TSE'18*, vol. 46, no. 2, pp. 196–221.

[23] pytransitions, "transitions," 2021. [Online]. Available: https://github.com/pytransitions/transitions

[24] "Appium: Mobile App Automation Made Awesome." 2021. [Online]. Available: http://appium.io

[25] T. J.-J. Li, L. Popowski, T. Mitchell, and B. A. Myers, "Screen2vec: Semantic embedding of gui screens and gui components," in *CHI'21*, pp. 1–15.

[26] Z. Liu, C. Chen, J. Wang, Y. Huang, J. Hu, and Q. Wang, "Guided bug crush: Assist manual gui testing of android apps via hint moves," in *CHI Conference on Human Factors in Computing Systems*, 2022, pp. 1–14.

[27] G. Hu, L. Zhu, and J. Yang, "Appflow: using machine learning to synthesize robust, reusable ui tests," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 269–282.