Mobile Test Script Generation from Natural Language Descriptions

Chun Li[1], Yifan Xiong[1], Zhong Li[1], Wenhua Yang[2], and Minxue Pan[1,*]
[1]Nanjing University, Nanjing, China
[2]Nanjing University of Aeronautics and Astronautics, Nanjing, China
{chunli, 211250043, mg1733033}@smail.nju.edu.cn, ywh@nuaa.edu.cn, mxp@nju.edu.cn
*corresponding author

Abstract—Mobile applications are increasingly integral to our daily lives. Currently, the correctness of GUI functions of mobile application is mainly ensured by executing manually written test scripts. However, manually writing these test scripts is not only time-consuming but also costly. Moreover, test scripts are highly vulnerable to application modifications and prone to corruption. In this paper, we propose a novel approach for writing test scripts that enables testers to directly express test intents in natural language within the script. Additionally, we present a new test script generation tool that transforms these test intents into their corresponding test events. Our proposed tool, named GenDroid, employs pre-trained models in conjunction with random forest to facilitate the conversion of test intents into the respective test scripts. To further alleviate the workload of testers and enable them to focus on composing critical test intents, we leverage the application's UI transfer graph to facilitate the automated generation of other test events, such as jump actions, throughout the generation process. Our results indicate an intent coverage of 88.1%, a notable 20.68% improvement compared to the similar-purpose tool, seq2act.

Keywords–Android testing; Test script generation; Natural language processing

1. Introduction

Currently, the predominant approach for GUI testing in mobile applications involves manual creation of test scripts followed by their execution through automated testing frameworks such as Appium[1] or Robotium[2]. Nevertheless, manual script writing proves to be a time-consuming, laborious, and expensive process [1]. The inclusion of interface widgets in test scripts, primarily through absolute positions or widget attributes, renders the scripts highly susceptible to widget positioning changes and prone to corruption. Moreover, hard-coding absolute positions or widget attributes circumvents the ability of the compiler or interpreter to verify the accuracy of this data. In certain cases, the failure of executing a test event can also result in subsequent test events not being executed.

There have been initial endeavors to enhance the process of test script creation. Nevertheless, testers still encounter the need to hard code control properties within the test script [2, 3, 4, 5]. Simultaneously, these methods often necessitate the specification of formal rules, which can potentially impose an additional burden on testers.

To address the limitations imposed by manually written test scripts, many researchers have proposed approaches [6, 7, 8, 9, 10, 11] to test applications by automating the exploration of applications. Despite the significant advancements in exploration approaches, which enable high code coverage and facilitate the identification of application defects, these methods often lack explicit guidance on test intent. Consequently, it becomes challenging to effectively test specific features of the application [8, 12].

To tackle the challenges mentioned above, we propose a new approach to writing test scripts to replace manual operation such as widget positioning. In our approach, testers only need to write natural language descriptions of the corresponding test events, which we refer to as **test intent**. There are two notable advantages to adopting this approach. Firstly, utilizing natural language offers significant flexibility, aligning with human usage patterns and avoiding additional burdens on testers. Secondly, by eliminating the need for testers to hard code control properties or locations within the test script, the generated scripts become more resilient to widget changes or application evolution, thereby enhancing their overall robustness. To further reduce the burden on the tester, we relax the test intent requirement as much as possible; informally, it only needs to contain a verb and a noun phrase like "modify account setting". We will offer a comprehensive definition and elaborate description of test intent in Section 2.

In order to facilitate the transformation of testers' test intent into their corresponding test scripts, we propose a novel method for test script generation called **GenDroid**. This approach leverages a pre-trained language model and random forest to achieve its objective. GenDroid employs a matching mechanism to associate widgets with their respective test intent and execute the relevant actions. To accomplish this, we extract both semantic information and positional information from the provided test intent and widget properties. Specifically, we capture the semantic content of the test intent and widgets, and compute the semantic similarity between them utilizing SentenceBERT [13]. Subsequently, we employ random forest to integrate the semantic similarity scores with the positional information, thereby effectively matching the widgets.

As test scripts typically consist of multiple test intents, it is common for there to be numerous transitions or jumps between two consecutive test intents. In order to further alleviate the burden on testers, GenDroid incorporates an automatic

[1]https://appium.io
[2]https://github.com/RobotiumTech/robotium

2693-9177/23/$31.00 ©2023 IEEE
DOI 10.1109/QRS60937.2023.00042

generation mechanism for test events that facilitate omitted test events generation between these adjacent test intents. To be more precise, prior to translating the test intent into test scripts, we utilize Droidbot [14], a lightweight mobile application automation exploration tool, to generate a UI transfer graph for the target application. Before each match based on the test intent, we employ the UI transfer graph to identify the widget with the highest confidence and transition the application to the corresponding target state. We consider the events generated during the transition process as omitted test events. By doing so, we ensure that the application is in the appropriate state before executing the matching process for the test intent. In addition to facilitating the connection between adjacent test intents, this mechanism also effectively reduces the impact of matching failures in previous test intents on the subsequent matching of test intents.

To assess the effectiveness of our proposed approach, we conducted experiments utilizing the PixelHelp dataset [15]. This dataset comprises 187 tasks, each consisting of multiple test intents. Our tool was evaluated on this dataset, resulting in an impressive intent coverage of 88.1%. This represents a notable improvement of 20.68% compared to the previous work, seq2act [15]. Furthermore, to evaluate the applicability of our tool in real-world scenarios, we selected 11 open-source applications from F-Droid for additional evaluation. When combined with the PixelHelp dataset, our approach achieved an intent coverage of 84.4% across a total of 636 test intents. In this paper, we make the following main contributions:

- We propose a new approach to writing mobile application test scripts. Testers can write test intents directly, eliminating the time and other costs required to write test scripts manually.
- We propose a script generation method based on a pre-trained model with a random forest, which generate test scripts by matching the corresponding widgets according to the test intent.
- We have conducted large-scale experiments, and the results show that our approach is more generalizable and outperforms existing approaches in test intent coverage. Both the tools and data will be publicly available to facilitate future research.

The remainder of the paper is structured as follows. Section 2 gives formal definition of test intent. Section 3 provides an overview of our framework. Section 4 presents the widget matching module. Section 5 presents the state transfer module. Section 6 presents the evaluation results. Section 7 surveys related work and Section 8 makes a conclusion.

2. Definition of Test Intent

In order to overcome the drawbacks associated with manual test script creation, we present a novel approach that involves directly writing test intents. This section focuses on providing a formal definition of test intent. Initially, we identify the fundamental components of a test intent by analyzing the crucial steps of test events. Subsequently, we draw insights from the AndroidHowTo dataset [15] to discern three distinct characteristics observed when describing test events in natural language. Finally, we will give a formal definition of test intent.

Based on the documentation of Appium, a widely recognized cross-platform GUI testing tool, the code within the test script can be succinctly summarized into two primary steps. The initial step involves positioning the widget through the utilization of either the absolute position or the properties of the widget. Subsequently, actions are performed on the identified widget. To illustrate this process, consider the following example:

```
close_dialog_button= \
  find_element_by_id("android:id/button1")
close_dialog_button.click()
```

The first line define the variable “close_dialog_button” as a widget located by the attribute “resource-id=android:id/button1”. Then second line perform “click” on it. To replace the current test script, the test intent should naturally include the description of the widget and the corresponding action. In this example, the test intent is to “click the close dialog button”. Hence, the essential components of a test intent should consist of the description of the widget and the corresponding action.

To obtain standardized patterns for describing events in natural language, while avoiding excessive constraints on testers, we drew inspiration from the AndroidHowTo dataset [15]. This dataset, which consists of a collection of 9,893 distinct English How-To instructions obtained through web data crawling, specifically focuses on operating Android devices. We preserved the three fundamental characteristics observed in natural language event descriptions, namely position words, verb types, and singular event occurrences, and incorporated them into our test intent.

a) Position Words: Natural language descriptions of events frequently incorporate position words, such as “at the top left.” While our initial aim in introducing the test intent was to mitigate the extensive burden of widget localization on testers, we observe that the substantial prevalence of locators within the dataset indicates that employing natural language to express the approximate widget location is an effective and convenient means to narrow down the widget search.

b) Verb Types: We found that besides the intent like “click close dialog button” related to operating devices, intent like “select verification method” or “change keys respond” which is related to application logic, also appear in the dataset. The main difference between them is types of verb. Verbs such as `click`, `tap`, etc. are related to operating devices, while `create`, `change` are related to function and logic. To take this into account, we restrict the natural language to the test task only and not to the specific words used.

c) Singular Event: We observed that nearly all natural language descriptions revolve around a single event. This characteristic simplifies the specification of the test intent and alleviates the burden on testers by avoiding the need for excessively long sentences. Additionally, this singular focus facilitates the translation of the test intent into the corresponding test code.

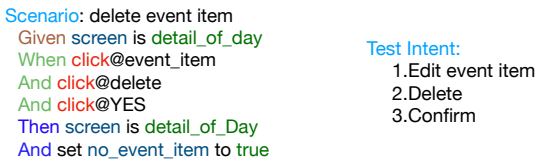

Figure 1. The difference between cucumber and test intent on deleting event items

Now, we can give a formal definition of test intent based on what we have learned about the dataset. A *test intent* is a natural language description that expresses and only expresses a single test event. The test intent must contain a description of the widget with the corresponding action. Location related information is optional.

Compared to the other format used to express test events, the most similar format is gherkin [5]. Gherkin is a formal language used in the Behavior-Driven Development tool cucumber to describe test scenarios. It is also used to generate test scripts in appflow [2]. The Figure 1 shows the difference between gherkin and test intent on the same task. Gherkin requires pre and post-conditions; acceptable actions are restricted and must have widget properties. Gherkin contains more detailed information, but it also imposes a writing burden on testers. In contrast, our proposed test intent is more concise and easy to use. In the following section, we demonstrate how to convert the test intent into the corresponding test script.

3. Approach Overview

The overview of GenDroid is shown in Figure 2. GenDroid consists of two modules. The first module is **Widget Matching**, which is used to match the test intent with specific controls. The second module is the **Path Planning**, which is used to supplement test events that may be missing between two consecutive test intents.

Widget Matching involves several steps to match the test intent with the corresponding widget. Firstly, we extract semantic information and optional positional information from both the test intent and the widgets in layout. Secondly, we employ SentenceBERT, a pre-trained language model, to compute the semantic similarity between the test intent and the widgets. Finally, we combine the semantic similarity with the location information using Random Forest. The widget with the highest confidence assigned by Random Forest will be selected.

To generate potentially omitted test events, we employ Path Planning before matching each test intent. Firstly, we obtain the current state s_i of the application and determine the target state s_{i+1} by matching the test intent with the widgets present in the entire UI transfer graph. Secondly, once s_{i+1} is determined, our path planning will generate paths between s_i and s_{i+1} and verify them. We will select the highest confidence path P from them. Finally, we migrate the application to state s_{i+1} by following path P. The events present in the generated path P are considered as omitted events that need to be included in the test execution.

Next, we will introduce the mechanism of widget matching, including the method of feature extraction, the process of computing semantic similarity, and the construction of random forests (Section 4). Then, we will show the path planning module's operation mechanism and the path confidence calculation method (Section 5).

4. Widget Matching

The goal of Widget Matching is to compute the confidence value between the test intent and the widget, enabling the ranking of widgets on the interface and selection of the widget with the highest confidence as the target widget. The workflow of widget matching initiates with the extraction of the semantic and positional features from both the test intent and the widget. Subsequently, these semantic features are employed to calculate the semantic similarity between the test intent and the widget. Finally, the positional features, along with the semantic similarity, are fed into a random forest model. The resulting output of this model serves as the confidence measure between the test intent and the widget. In the subsequent sections, we provide a detailed elaboration of each individual step.

4.1. Feature Extraction

We initiate the process by extracting the semantic and positional features from both the test intent and the widget. In the case of the test intent, which is expressed in natural language, we employ part-of-speech tagging (POS Tagging) [16] to extract its semantic and positional features. POS Tagging facilitates the identification of prepositions in instances where positional information is present within the test intent. Consequently, we can utilize POS Tagging to isolate the positional features within the test intent. By conducting a thorough investigation of the AndroidHowTo dataset, we have incorporated two positioning methods, namely absolute and relative, which are distinguished based on specific keywords. Additionally, for the remaining portion of the test intent, we utilize POS Tagging to analyze the verbs and their corresponding noun phrases. For example, the phrase "create new contact" can be segmented into the action "create" and the widget information "new contact." The rationale why we distinguish verbs from corresponding noun phrases will be explained later.

Regarding the widget, according to official documentation [3] [4] [5], there are three properties, namely *text*, *content-desc*, and *resource-id*, which serve as indicators of the widget's functionality. Therefore, we employ these properties as the basis for extracting semantic information from the widget. These properties written in natural language can be analyzed using POS Tagging to extract their verbs and corresponding noun phrases as semantic features. For example, "Delete" will be divided only into action "Delete", "newTask" should be divided into action "new" and object "task". The positional feature of widget (absolute position) could be extract from *bound* attribute.

[3]https://developer.android.com/guide/topics/resources/providing-resources
[4]https://developer.android.com/guide/topics/ui/accessibility/principles
[5]https://developer.android.com/reference/androidx/test/uiautomator/UiSelector

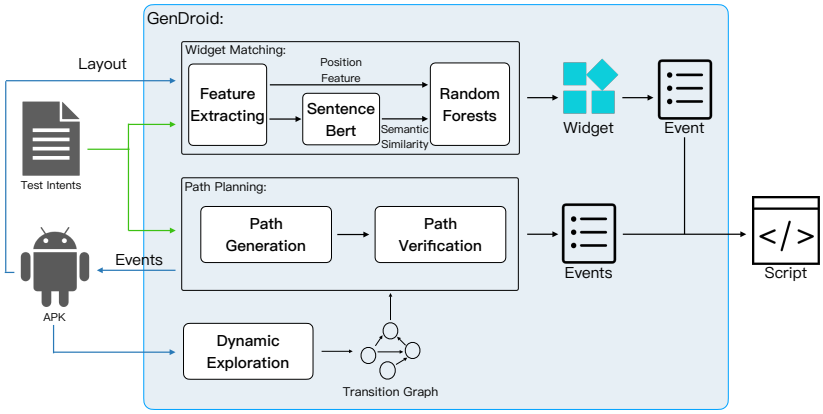

Figure 2. Overview of GenDroid

The reason why we distinguish actions from corresponding noun phrases when extracting semantic features is because they are often unrelated semantic parts. Different verbs can be paired with different noun phrases, and vice versa. Therefore, if no differentiation is made, it may result in only sub-optimal results in subsequent semantic similarity calculations. For example, if the test intent is "create new contact" and there are two widget which properties is "create group" and "new contact". If we do not distinguish between verbs and nouns, then the test intent and the first widget both contain "create", which may result in high semantic similarity between them.

4.2. Semantic Similarity Calculation

Algorithm 1: Semantic Similarity Calculation

Input: TestIntent, Widget
Output: SemanticSimialrity

1 Attribute ← `GetAttribute(`*Widget*`)` ;
2 WidgetVerb, WidgetNoun ← `FeatureExtraction(`*Attribute*`)` ;
3 IntentVerb, IntentNoun ← `FeatureExtraction(`*TestIntent*`)` ;
4 VerbSimilarity ← `Similarity(`*WidgetVerb, IntentVerb*`)` ;
5 NounSimilarity ← `Similarity(`*WidgetNoun, IntentNoun*`)` ;
6 WholeSimilarity ← `Similarity(`*Attribute, Intent*`)` ;
7 SemanticSimilarity ← `Max(`*VerbSimilarity, NounSimilarity, WholeSimilarity*`)` ;
8 **return** *SemanticSimilarity*

After extracting the semantic features of the test intent and widget, the next step is to calculate the semantic similarity. In this work, we utilize the pre-trained model SentenceBERT [13] for the purpose of conducting the semantic similarity calculation. The decision to use the pre-trained model is two-fold: (1) Based on our analysis of the AndroidHowTo dataset, we have observed that nearly all test intents share the same semantic space as that of general-purpose usage. (2) The attributes of the widget, expressed in natural language, also share the same semantic space as that of general-purpose language. Therefore, employing a language model pre-trained on large-scale general-purpose corpora can lead to improved effectiveness and reduced costs in the calculation of semantic similarity.

Another consideration is the choice between a sentence-level model and a word-level model. The utilization of a sentence-level model is preferred due to the nature of the test intent, which is a complete sentence, and the fact that widget attributes often consist of multiple words. Employing a word-level model does not adequately capture the overall meaning of the sentence [17]. Algorithm 1 provides an overview of the semantic similarity calculation process.

First, we obtain the first non-empty semantic attribute in the order of *text*, *content-desc*, *resource-id* (Line 1). Then, we extract the semantic features from the test intent and attributes (Line 2-3). Next, we calculate the semantic similarity of verbs and noun phrases in the test intent and attributes (Line 4-5). We also considered the direct semantic similarity between test intent and attribute (Line 6). Finally, we aggregate these three semantic similarities and select the most significant similarity between the test intent and the widget (Line 7).

We do not choose to use the three attributes at once because they have different purposes. Although both *text* and *content-desc* contain the semantic information of the widget, *text* is displayed to the user, so we believe that *text* is more reflective of the semantic information of the widget than *content-desc*. *resource-id* is used by developers to reference different controls in their code, sometimes only containing some category information. So, when we calculate the semantic similarity, we try *text*, *content-desc* and *resource-id* in order of priority.

4.3. Feature Fusion

Next, we discuss how to fuse positional feature with semantic similarity. To achieve this, we curated a subset from PixelHelp [15] and utilized it to create a new dataset. Subsequently, we trained a Random Forest [18] model on this dataset to effectively combine feature information from diverse sources.

The resulting output of the Random Forest serves as the confidence measure between the test intent and the control.

The reason why we use Random forest is that the positional feature in the test intent is expressed by natural language, and usually can only represent a certain area on the screen, while the positional feature in the widget accurately reflects the position on the screen. Therefore, the positional information in test intent and widget cannot be normalized. On the other hand, positional feature is discrete, while semantic similarity is continuous. According to the characteristics of our data, it is reasonable to use Random forest.

To begin, we construct a dataset that corresponds to our requirements. Initially, we select a subset from the PixelHelp dataset. For each test intent within this subset, we replicate it on a virtual machine to acquire the corresponding widget. Subsequently, we extract the features from both the test intent and the widget, employing the methodology outlined in Section 4-A. We calculate the semantic similarity between them utilizing the algorithm described in Section 4-B. Ultimately, we generate a data entry for each test intent and widget pair, encompassing their semantic similarity, positional information within the test intent, and positional information within the widget.

In order to prevent class imbalance problems [19], we add counterexamples by data augmentation. Specifically, for positive examples lacking positional information, we want the model to capture the differences caused by the difference in semantic similarity, so we generate counterexamples with arbitrary locations and lower semantic similarity. For positive examples with positional information, we want the model to capture the differences due to location information when the semantic similarity is very close. So, we generate counterexamples with close semantic similarity but different location information. Finally, we generate 144 class-balanced samples, each of which is a vector containing the positioning method, the positioning information, the position of the widget, the screen size or the position of the reference widget, and the semantic similarity of the test intent to that widget.

Finally, we trained a dichotomous random forests model, removed the layer of output categories, and directly used its output probabilities as the confidence between test intent and widget. After we get the widget with the highest confidence, we will synthesize the test events based on the type of widget. Currently, our tool only supports the two most commonly used operations, `click` and `set_text`. The remaining operations will be supported in the future.

5. Path Planning

To further reduce the burden on testers and increase the flexibility of test intents, we allow intervals between successive test intents, i.e., we always assume that the current application state does not contain the widget corresponding to the test intent. Formally, let the current state of the application be s_i, and the target state that contains the widget corresponding to the next test intent is s_j. We assume that $i \neq j$.

We use DroidBot [14] to generate a UI transfer graph G for the application under test. Each node in G represents a state s, and each edge (s_i, s_j) represents a transfer between two states via event. Here, we use Droidbot's definition of a state, which considers the *class*, *resource-id*, *text*, *enable*, *checked*, and *selected* attributes for each widget on the activity.

Static analysis techniques [20, 21] can also be used to generate UI transfer graph. However, it will produce many paths that are unreachable or duplicative, which requires extensive path validation for practical use. Although more time-consuming, the dynamic analysis technique can generate more accurate UI transfer graphs. Therefore, we believe such a time cost of dynamic analysis is reasonable for achieving better UI transfer graph. Furthermore, considering that generation process is offline, the time cost of dynamic analysis is acceptable in practice.

Given that the current application is in state s_i, our initial objective is to identify the target state s_j on graph G where the widget with the highest confidence, based on the current test intent, is located. Subsequently, we generate all possible simple paths from the current state s_i to the target state s_j. These paths are then verified and sorted based on their confidence values. The path with the highest confidence is selected as the optimal transfer path. The detailed planning process is illustrated in Algorithm 2.

Algorithm 2: Path Planning

Input: TestIntent, UI Transition Graph G, CurrentState cs
Output: Path

```
1  Widgets ← GetAllWidgets(G) ;
2  Widgets ← SortByConfidence(Widgets, TestIntent);
3  foreach w in Widgets do
4      States ← GetStatesByWidget(w) ;
5      Paths ← ∅ ;
6      foreach ds in States do
7          ps ← FindAllSimplePath(cs, ds) ;
8          Paths.Extend(ps) ;
9      end
10     Candidates ← ∅ ;
11     foreach p in Paths do
12         result ← Validate(p) ;
13         if R then
14             Candidates.Append(p);
15         end
16         Reset(cs) ;
17     end
18     Candidates ← Sort(Candidates, TestIntent) ;
19     if Candidates ≠ ∅ then
20         return Candidates[0]
21     end
22 end
23 return None
```

Firstly, we extract all widgets from graph G and calculate their respective confidences in relation to the current test intent.

Subsequently, we sort the widgets based on these confidence values (Lines 1-2). For each widget, we locate all candidate states that contain the widget (Line 4) and compute all simple paths from the current state to each candidate state (Lines 6-9). Once all paths have been obtained, the verification process begins (Lines 11-17). Path validation entails executing the events along the path. If the path can be executed successfully, it passes the validation; otherwise, it does not (Lines 11-17). After each path is verified, we reset the system to the current state (Line 16). Specifically, we uninstall and reinstall the application, restoring the application to its initial state and then transitioning it to the current state. Ultimately, the path with the highest confidence is selected (Lines 18-20).

We prefer to choose paths that are shorter and have higher semantic similarity to the test intent for two reasons: (a) considering the locality principle, the widgets corresponding to adjacent test intents are more likely to be close to each other on the graph, so we prefer to choose shorter paths; (b) if the paths pass the validation, we want the widget corresponding to the test events on the paths associated with the test intent as much as possible. To consider both simultaneously, we borrowed from CraftDroid [22], with the following formula.

$$\mathcal{C}(path) = \frac{avg(\mathcal{C}(widgets, TestIntent))}{1 + log(len(path))}$$

Let $\mathcal{C}$ represent the confidence. The confidence of a path is calculated as the average of the confidence of the widgets involved in the events along the path, divided by the log of the path length. The use of log ensures that our formula exhibits greater sensitivity to shorter paths, while diminishing the emphasis on path length when all paths are long. Instead, the focus is placed on the semantic similarity between each widget and the test intent.

6. Evaluation

In this Section, we will look at two main areas: 1. the ability of the widget matching module to find the correct widget, and 2. the ability of the entire tool to translate test intent into a test script. We focus mainly on answering the following research questions:

RQ1: Widget Matching Module. Can our widget matching module find the correct widget based on the test intent?

RQ2: Test Intent Coverage. How does GenDroid compare to previous tool in terms of test intent coverage?

RQ3: Impact Factor. What are the most important reasons affecting the capability of GenDroid?

6.1. Experimental Setup

a) Dataset: We chose the PixelHelp dataset mentioned in seq2act [15], which has 187 tasks containing multiple test intents. Since the paper proposing the dataset does not mention what version and device of the virtual machine the dataset was built on, we manually re-executed these 187 tasks on our experimental virtual machine. We ended up with 127 successfully running tasks and 387 test intents. The rest of the tasks were missing UI, or the application could not be launched due to version issues. For the tasks that could not run successfully, we discarded them because they could not convert successfully from test intent to test script.

In addition to the PixelHelp dataset, we collected eleven applications from F-Droid to verify our approach's applicability. Our collection strategy was as follows: firstly, since GenDroid cannot support complex interactions, we excluded games, images, and multimedia-related applications. Secondly, we wanted the applications not to have frequent interface changes because of network-related features, which would make the applications unstable, so we excluded applications that relied too much on network. Among the remaining applications, we randomly selected 11 applications as our benchmark. We convened eleven graduate students with Android development experience to write a total of 249 test intentions.

b) Widget Matching: We separate the widget matching module from the tool to test our ability to match widgets. We first set the application to the target state where contains the target widget corresponding the test intent. Next, we input the layout information of the application into the module and check if the widget output by the module are correct. For the random forests of widget matching, we extracted a series of test intents from the discarded tasks in PixelHelp. We collected and labeled the data as mentioned above. We used a random hyperparameter search with ten-fold cross-validation to determine the hyperparameters of the random forests.

c) Path Planning: We will only consider the first five widgets during path planning to avoid the app searching for a long time without stopping. Each widget considers the first five paths due to the same reason above.

d) Test Intent Coverage: To make the comparison, we conducted comparative experiments with seq2act on the 387 successful test intents. Since seq2act considers the startup application as a test intent, we included the startup application when conducting the comparison experiments. For a fair comparison, we fed the original data to seq2act, collected its output widgets and actions on each test intent, and played them back on a virtual machine to get its final performance. At the same time, we collect each widget and action output by GenDroid and play it back on the virtual machine. Finally, we compare the test intent coverage of both tools. In addition, GenDroid is experimented with on our newly collected dataset to verify the generalizability of it.

e) UI Transfer Graph: For UI transfer graph generation, we used Droidbot to construct the UI transfer graph by generating 100 events for all tested applications according to the breadth-first strategy. We set a five-second wait between every two test events to bring the app back to a stable state.

f) Experiment Environment: All experiments were run on macOS 11.5.2 with 2.6 Ghz 6 cores CPU and 32GB RAM. The version of Android VM we used is Android 8.0 With Google API, API level 26.

6.2. RQ1: Widget Matching Module

In this section, we will evaluate the random forests and widget matching accuracy in the widget matching module.

TABLE I
EVALUATION OF RANDOM FORESTS

Accuracy	Precision	Recall	F1
89.6%	84.6%	91.6%	88%

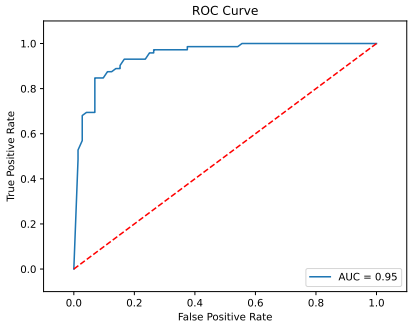

Figure 3. Roc curve and AUC

a) Random Forests: We gathered a dataset comprising 144 samples with balanced class distribution and allocated 80% of them as the training set. To determine the optimal hyperparameters for the random forests classifier, we utilized the random search algorithm implemented in SKlearn [23]. The random search consisted of 1000 iterations, with each iteration employing a 10-fold cross-validation. The entire process was completed in under twenty minutes on our experimental machine. Once the hyperparameters were determined, the performance of our model became fixed. Notably, our model achieved an accuracy of 89% on the test set, which demonstrates its capability to effectively integrate location information with semantic information.

In addition to the accuracy, we also calculated our model's Precision, Recall, and F1 values, as shown in the Table I. Precision and Recall reflect that our model is more successful in distinguishing between positive and negative samples. In contrast, the F1 value reflects that our model is more stable.

In addition, we examined the ROC curve of our model by assessing the AUC (Area Under the Curve), as depicted in Figure 3. This analysis allowed us to evaluate the performance of our random forests classifier across various threshold values. Notably, our model achieved an impressive AUC area of 0.95 on the entire dataset, which signifies its reliability in accurately distinguishing between positive and negative examples.

We also conducted a analysis of the model's output. It was observed that for correctly predicted samples, irrespective of the positioning method, the model assigned higher confidence to widgets exhibiting greater semantic similarity, particularly when their positions were similar. When the test intent contained positional information, the model demonstrated a focus on such information, assigning higher confidence to widgets that aligned more closely with the provided location. Among the samples with prediction errors, there were positive examples where the semantic similarity was too low to be distinguished, even for less experienced developers. For instance, the semantic similarity between "Navigate Up" and "Menu" was less than 0.3, which posed challenges for our random forests classifier. Fortunately, even if certain test intents were not adequately covered during the process, it did not lead to the failure of all subsequent test intents. This was attributed to our path planning module, which ensured that even if an incorrect widget was currently selected, it did not impact the success rate of the next test intent. The path leading to the state of the widget corresponding to the subsequent test intent might include the actual corresponding test event from the preceding test intent.

TABLE II
COMPARISON WITH SEQ2ACT ON THE PIXELHELP DATASET

App	Tasks	Intents	Coverage(%)	
			seq2act	GenDroid
Chrome	28	138	84.1 (116/138)	87.7 (121/138)
Clock	11	41	82.9 (34/41)	97.6 (40/41)
Photo	16	58	87.7 (50/57)	87.7 (50/57)
Gmail	23	91	68.1 (62/91)	87.9 (80/91)
Setting	44	172	58.1 (100/172)	85.5 (149/172)
Contact	2	6	100 (6/6)	100 (6/6)
Google	3	9	77.8 (7/9)	77.8 (7/9)
Total	127	514	73.0 (375/514)	88.1 (453/514)

b) Widget Matching Accuracy: The accuracy of the widget matching module was evaluated independently. To isolate the impact of other modules, we manually configured the application to the correct state, provided the test intent as input to the widget matching module, and observed whether the module correctly identified the corresponding widget. Out of a total of 636 test intents, we successfully matched 590 corresponding widgets, resulting in a match rate of 93.7%.

In conclusion, using random forests effectively fuses semantic similarity with positional features and gives confidence in the widgets. The widget matching module achieved an accuracy of 93.7% on the dataset, and the test intent transformation using the widget matching module is reliable.

6.3. RQ2: Test Intent Coverage

The criterion for testing intent coverage is that its corresponding widget is triggered in the correct state. Table II shows the results of our experiments with seq2act on the PixelHelp dataset. Since seq2act treats launching the application as a test intent, we have included it for a fair comparison.

GenDroid achieves a test intent coverage of 88.1%, surpassing seq2act by 20.68%. In quantitative terms, GenDroid covers 78 additional test intents compared to seq2act. More specifically, GenDroid exceeded seq2act's test coverage on 4 apps, while 3 apps had the same test coverage as seq2act.

We conducted an investigation to understand the significant disparity of nearly 30% observed in the **Setting** application when comparing it against seq2act. Upon analyzing the experimental results, we discovered that seq2act solely selects widgets that are currently present in the interface. However,

there is a portion of the dataset where the test intent is discontinuous. Relying solely on the events generated by seq2act makes it impossible to navigate to the correct state, resulting in subsequent test intents remaining uncovered. In contrast, our tool successfully addressed this issue by leveraging our path planning module, which attempts to perform state transfers before transitioning to each test intent. Similar challenges were also encountered with the **Gmail** application, where certain test intents were not accurately covered. Nevertheless, thanks to our path planning module, subsequent test intents were still successfully covered.

We also examined the reasons behind the successful outcomes observed in the **Chrome** and **Clock** applications. Upon analysis, we discovered that the main reason is the failure of seq2act to correctly match the widget in the appropriate state. Given that seq2act employs a deep neural network such as Transformer, it becomes challenging to pinpoint the exact reasons for these inaccuracies. However, considering that widget properties are typically defined by developers to ensure readability, they should ideally exist within the same semantic space as a general-purpose domain. Furthermore, it is important to note that seq2act has been trained solely on Android-related datasets, specifically Rico [24] and AndroidHowTo [15]. This limited training data may not capture the semantics of general-purpose scenarios accurately. Consequently, adopting a pre-trained model that already aligns with the desired semantic space is a reasonable approach, which is supported by the experimental results.

In addition to the PixelHelp dataset, we extended our evaluation to include eleven other open-source software, measuring the test intent coverage for each. The results of the experiments are presented in Table III. The official code for seq2act exclusively supports the PixelHelp dataset. Despite our diligent efforts to adapt our dataset to meet the requirements of seq2act, the official code remains nonfunctional. Consequently, we solely considered the outcomes obtained using our tool for these open-source software evaluations. Notably, launching the application was not included as a test intent, as it was not relevant for comparative experiments. we also provide the results of an ablation experiment where we removed the path planning module. These additional results are included in the table for comparison.

The experimental results demonstrate a significant 20.05% performance improvement achieved by incorporating the path planning module. This improvement can be attributed, in part, to the path planning module's ability to ensure that the failure of a preceding test intent does not impact the coverage of subsequent test intents. Figure 4 illustrates an example scenario where the path planning module is utilized. In this case, Screen S1 transitions to Screen S2 using path planning, but mistakenly selects the widget *Help* within the red box, which is not the correct widget for the initial intent. Then, Screen S2 then transitions to Screen S3 by clicking *Help*, and subsequently, Screen S3 transitions to Screen S4 under the guidance of the path planning module. Finally, the correct widget *Add activity* within the red box is selected, fulfilling the second intent.

TABLE III
TEST INTENT COVERAGE ON ALL APPLICATIONS WITH ABLATION EXPERIMENTS

App	Tasks	Intents	w/o Path Planning	GenDroid
Chrome	28	110	82.7% (91/110)	84.5% (93/110)
Clock	11	30	96.7% (29/30)	96.7% (29/30)
Photo	16	41	82.9% (34/41)	82.9% (34/41)
Gmail	23	68	55.9% (38/68)	83.8% (57/68)
Setting	44	128	53.9% (69/128)	82.0% (105/128)
Contact	2	4	100% (4/4)	100% (4/4)
Google	3	6	66.67% (4/6)	66.67% (4/6)
Cleartodo	7	16	62.5% (10/16)	68.8% (11/16)
Currency	9	25	60% (15/25)	96% (24/25)
MyExpenses	9	34	70.6% (24/34)	82.4% (28/34)
Notify	6	24	70.8% (17/24)	91.7% (22/24)
MaterialFiles	6	15	80% (12/15)	100% (15/15)
ShopWithMom	4	10	40% (4/10)	60% (6/10)
SimpleCalendarPro	8	25	64% (16/25)	84% (21/25)
TodoList	10	31	87.1% (27/31)	93.5% (29/31)
SimpleBitcoinWallet	8	24	91.7% (22/24)	91.7% (22/24)
Wikipedia	4	20	70.0% (14/20)	45.0% (9/20)
Timetracker	8	25	72.0% (18/25)	96% (24/25)
Total	206	636	70.3% (447/636)	84.4% (537/636)

TABLE IV
TIME TO CONVERT TEST INTENT TO TEST SCRIPT

GenDroid	Average(Second)	Median(Second)
Origin	262	106.5
w/o Path Planning	26.5	25

In addition to test intent coverage, the efficiency of converting test intent into test scripts is also crucial. One of the motivations of our work is to reduce the time overhead and cost of testers. Therefore, we further counted the time spent on test script generation by GenDroid.

We also investigated the efficiency impact of path planning on GenDroid. During the statistical analysis, we found that some scripts took significantly more time than others, causing the mean to not accurately reflect the changes before and after. Therefore, our primary focus was on changes in the median. The results in Table IV show that the median time increased from 25 seconds to 106 seconds after combining path planning. Since this generation process is offline and one-time, considering the 20.13% coverage improvement brought about by path planning, these time costs are acceptable. The increased time overhead primarily arises from the path validation process during path planning. During the verification of each path, there are pauses inserted between executed events to ensure application stability. Following the verification of each path, the application is uninstalled and reinstalled to restore it to its initial state, which also incurs a time overhead. To maximize efficiency, we can cache paths that have been verified as infeasible and compare new paths against them to reduce computation and accelerate the process. This part of the work will be integrated into our tool in the future.

6.4. RQ3: Impact Factor

The test intent coverage is a key metric for evaluating the effectiveness of our tool, with its success heavily reliant on

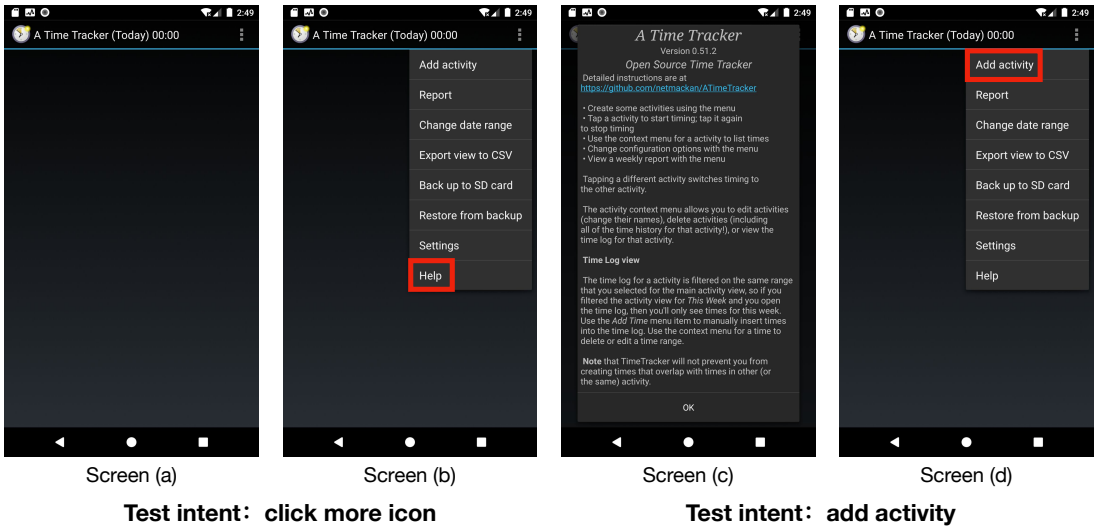

Figure 4. A task of **Timetracker** in the dataset where the first test intent "click more icon" is not covered correctly but does not affect the success of subsequent test intents. The path planning module fixes this error automatically

the widget matching module's ability to accurately match the correct widget. Upon analyzing the test intents that were incorrectly matched, we identified two main reasons for these inaccuracies. Firstly, incomplete or missing properties of the widget contributed to the incorrect matches. Secondly, incomplete UI transfer graphs also played a role in these errors.

Through our analysis of the experimental data, we identified a need for better maintenance of widget properties in the open-source applications. Specifically, we observed instances where different widgets possessed identical properties, leading to challenges in distinguishing between them based on similarity during the path planning process. Figure 5 illustrates an example from the **Cleartodo** software, where the widget located in the lower right corner of the first figure and the widget in the upper right corner of the second figure share the same properties, despite serving distinct functions. The first widget is responsible for adding new todo items, while the second widget is used for adding new groups. This scenario presents a significant challenge for our path planning module. To tackle the second issue, achieving a complete UI transfer graph for an application is nearly impossible due to the limitations of dynamic or static exploration techniques in the mobile application domain. We experimented with static exploration techniques such as TrimDroid [20] and Gator [21] to generate UI transfer graphs. However, since these techniques rely on static analysis, the generated state transfer graphs are constrained by the accuracy of static analysis and may not capture all widgets and activities relationships. We have designed our tool with various interfaces to facilitate the replacement of the UI transfer graph generation tool, allowing for the exploration of alternative techniques.

In scenarios where the UI transfer graph is incomplete, the target state associated with the test intent does not appear, rendering our path planning module unable to identify the correct path. This issue arises in the official Google app. For

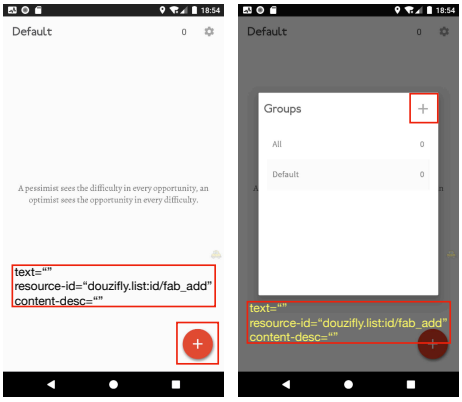

Figure 5. The Add button appears with the same properties in different positions for different functions

instance, in the **Setting** application, there are intervals between certain test intents. However, the widget corresponding to the test intent is not present in the UI transfer graph, leading to the failure of covering the test intent successfully. As depicted in Figure 6, the left and right states correspond to consecutive test intents. After completing the first test intent, the application does not transition to the right state but remains in an intermediate state. At this juncture, the path planning module must transfer the application to the correct state; however, the right state is absent from the transfer graph. Consequently, it becomes impossible to cover the second test intent.

6.5. Threats to Validity

a) Internal Threats.: The main internal threat comes from the performance of the random forests. Due to the small dataset for training the random forests, there may be overfitting and insufficient data. For this threat, we conducted various

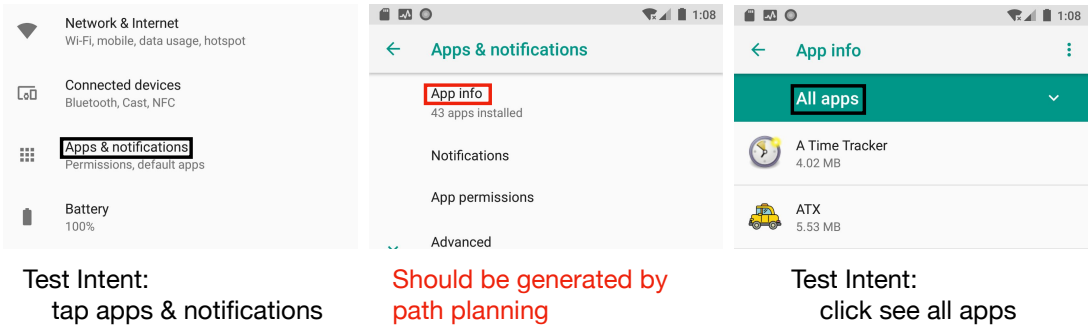

Figure 6. The screenshot on the left is the initial state of the application. The application enters the state of the middle screenshot after clicking *Apps & notifications* according to the first test intent. The path planning module should then migrate the application from the middle screenshot state to the right screenshot state according to the next test intent.

experiments to validate the ability of the random forests and experiments for the entire widget matching module to verify its effectiveness. We will continue to collect more data to optimize the model.

Also, our choice of Android version or virtual machine device may lead to unreliable results in our experiments. For this reason, we tried multiple Android versions and virtual machine devices and chose the one that could reproduce the most test intents.

b) External Threats.: The main external threat stems from the potential bias in the test intent written for open-source applications. To mitigate the possible bias, we gathered eleven graduate students with Android development experience to learn from actual data in PixelHelp and AndroidHowTo before writing test intents for open-source applications.

7. Related Work

We broadly categorize related work on mobile test script generation into (i) generation from mobile application exploration, (ii) generation from record and replay, and (iii) generation from synthesis and transfer.

7.1. Generation From Exploration

This part is mainly distributed in various Android GUI automation tests. Android GUI automation tests explore the application by adopting different exploration strategies for the application, and the events triggered during the exploration process can be used to generate test scripts. Based on the different exploration strategies, we can classify them. Some tools [6, 25] adopt random strategies to generate input for Android application. Monkey [25] is the most frequently used Android testing tools. However, the generated test cases contain a large number of noneffective or redundant events and this will be a threat to the testing effectiveness. Model-based approaches [26, 27, 28, 10, 29, 30, 9, 14] use dynamic or static strategies to build models to describe the application's behaviors. This type of work is part of the cornerstone of our UI transfer graph, where state granularity and comparison are very important. Evolutionary algorithms are applied by systematic strategies [8, 7] to generate input for Android application. They can cover hard-to-reach codes. However, the iterative process is often very time-consuming and can generate erroneous scripts during the mutation process [31]. Machine learning also has been applied in Android GUI testing. The learning approach [32, 33, 11, 34] can mitigate the drawbacks of heuristics, but requires collecting enough data for training, especially deep learning.

7.2. Generation From Record and Replay

Test record and replay works generate test scripts by recording testers' actions on Android device. The main reason [35] for developing this tool is that the Android platform can be adapted to many different devices, including different sizes and versions. The record and replay technology can automate the process to ensure that the software can run properly on different devices. The most successful record and replay tools in recent years include monkeyrunner [36], RERAN [37], and Espresso [38]. All the record and replay tools have the problem that some events cannot be recorded. In order to cope with these problems, Google developed Espresso [38]. Espresso records events through Debug mode, unlike the above three tools, which collect contextual information when the interactive function is called.

The scripts generated by the recording-playback technique still use absolute position [36, 37, 39] to position UI elements. This leads to relatively fragile test scripts [35]. Our work instead uses natural language to describe the test intent. It improves the robustness of the test scripts by abstracting them. The path planning module also makes our test scripts robust to changes in interface elements.

7.3. Generation From Synthesis and Transfer

More similar to our work are Augusto [4] and AppFlow [2], both of whom work on test generation for interface tests. They consider application-agnostic features [4, 40] or generic GUI patterns [41] for test generation. Augusto uses the formal language Alloy [3] to describe the structure of the GUI in a predefined test scenario with pre- and post-conditions to generate GUI test code. AppFlow shares the same intuition as Augusto, using predefined test scenarios containing pre-

and post- conditions and GUI structure. The difference is that AppFlow uses the formal description method Gherkin [5], a more relaxed format compared to Alloy. Augusto and Appflow use descriptions of test scenarios that are too complex relative to our test intent and will create a new burden for testers.

Another related category of work is test script migration [22, 42, 17], based on the same motivation as Augusto and Appflow, with the difference that they are migrating an existing script to another application. CraftDroid [22] and AppTestMigrator [42] use a similar approach for script migration, i.e., finding the target application widget with the highest semantic similarity to the widget operated by the source test script. For the calculation of semantic similarity, they both use word-level word embedding [43, 44] techniques to calculate the semantic similarity between two widgets by computing the cosine similarity of the embedding.

Mariani et al. first present empirical study [17] on both tools in depth and argues that sentence-level word embedding techniques [45, 46] and sentence-level similarity calculation would be better because widget attributes tend to be composed of multiple words. Also, Mariani's work points out that training word embedding on a mobile application domain dataset give better results than general purpose corpora. GenDroid draws on the strengths of previous work in that we use sentence-level semantic similarity computation techniques [13] to compute the semantic similarity between the test intent and the widget. At the same time, the properties of widgets are read and written by humans and should share the same semantic space as the general corpus in most cases. Therefore, we employ a pre-trained model and fine-tune it on a dataset of mobile applications to learn both the generic corpus and mobile application-specific semantic knowledge.

Seq2act [15], our comparison subject, does not fall into any of the above categories. seq2act can translate natural language commands for mobile user interface operations into the corresponding widgets and actions. However, it can only handle cases where the test intent is continuous. It requires training a Transformer model from scratch, which is costly. Our work uses UI transfer graphs to allow the application to migrate automatically to the correct state. The pre-trained model we use also reduces the training cost significantly.

8. Conclusion

In this paper, we propose a new approach to writing test scripts that allow testers to write test intents based on natural language, reducing testers' burden and increasing the scripts' robustness. We propose GenDroid, a new approach based on pre-trained models with random forests to convert test intents into corresponding test events. To further reduce the burden on testers, we propose a path planning module to automatically generate omitted test events. At the same time, the path planning module can also avoid the impact of the failure of the previous test intent on the subsequent test intent. Experiments show that GenDroid outperforms the state-of-the-art tool seq2act in test intent coverage.

Acknowledgment

This research was supported by the National Natural Science Foundation of China (Nos. 62232014 and 61972193).

References

[1] Jun-Wei Lin, Navid Salehnamadi, and Sam Malek. Test automation in open-source android apps: A large-scale empirical study. In *ASE*, pages 1078–1089. IEEE, 2020.

[2] Gang Hu, Linjie Zhu, and Junfeng Yang. Appflow: using machine learning to synthesize robust, reusable UI tests. In *ESEC/SIGSOFT FSE*, pages 269–282. ACM, 2018.

[3] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[4] Leonardo Mariani, Mauro Pezzè, and Daniele Zuddas. Augusto: exploiting popular functionalities for the generation of semantic GUI tests with oracles. In *ICSE*, pages 280–290. ACM, 2018.

[5] Cucumber. Gherkin syntax, 2022.

[6] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. Dynodroid: an input generation system for android apps. In *ESEC/SIGSOFT FSE*, pages 224–234. ACM, 2013.

[7] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. Evodroid: segmented evolutionary testing of android apps. In *SIGSOFT FSE*, pages 599–609. ACM, 2014.

[8] Ke Mao, Mark Harman, and Yue Jia. Sapienz: multi-objective automated testing for android applications. In *ISSTA*, pages 94–105. ACM, 2016.

[9] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. Guided, stochastic model-based GUI testing of android apps. In *ESEC/SIGSOFT FSE*, pages 245–256. ACM, 2017.

[10] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. Practical GUI testing of android applications via model abstraction and refinement. In *ICSE*, pages 269–280. IEEE / ACM, 2019.

[11] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. Reinforcement learning based curiosity-driven testing of android applications. In *ISSTA*, pages 153–164. ACM, 2020.

[12] Juha Itkonen, Mika Mäntylä, and Casper Lassenius. The role of the tester's knowledge in exploratory software testing. *IEEE Trans. Software Eng.*, 39(5):707–724, 2013.

[13] Nils Reimers and Iryna Gurevych. Sentence-bert: Sentence embeddings using siamese bert-networks. In *EMNLP/IJCNLP (1)*, pages 3980–3990. Association for Computational Linguistics, 2019.

[14] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Droidbot: a lightweight ui-guided test input generator for android. In *ICSE (Companion Volume)*, pages 23–26. IEEE Computer Society, 2017.

[15] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. Mapping natural language instructions to

mobile UI action sequences. In *ACL*, pages 8198–8210. Association for Computational Linguistics, 2020.

[16] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. spaCy: Industrial-strength Natural Language Processing in Python. 2020.

[17] Leonardo Mariani, Ali Mohebbi, Mauro Pezzè, and Valerio Terragni. Semantic matching of GUI events for test reuse: are we there yet? In *ISSTA*, pages 177–190. ACM, 2021.

[18] Tin Kam Ho. Random decision forests. In *ICDAR*, pages 278–282. IEEE Computer Society, 1995.

[19] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial: special issue on learning from imbalanced data sets. *SIGKDD Explor.*, 6(1):1–6, 2004.

[20] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. Reducing combinatorics in GUI testing of android applications. In *ICSE*, pages 559–570. ACM, 2016.

[21] Shengqian Yang, Hailong Zhang, Haowei Wu, Yan Wang, Dacong Yan, and Atanas Rountev. Static window transition graphs for android (T). In *ASE*, pages 658–668. IEEE Computer Society, 2015.

[22] Jun-Wei Lin, Reyhaneh Jabbarvand, and Sam Malek. Test transfer across mobile apps through semantic mapping. In *ASE*, pages 42–53. IEEE, 2019.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2011.

[24] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. Rico: A mobile app dataset for building data-driven design applications. In *UIST*, pages 845–854. ACM, 2017.

[25] Google. Ui/application exerciser monkey, 2022.

[26] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. Using GUI ripping for automated testing of android applications. In *ASE*, pages 258–261. ACM, 2012.

[27] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Bryan Dzung Ta, and Atif M. Memon. Mobiguitar: Automated model-based testing of mobile apps. *IEEE Softw.*, 32(5):53–59, 2015.

[28] Tanzirul Azim and Iulian Neamtiu. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*, pages 641–660. ACM, 2013.

[29] Duling Lai and Julia Rubin. Goal-driven exploration for android applications. In *ASE*, pages 115–127. IEEE, 2019.

[30] Young Min Baek and Doo-Hwan Bae. Automated model-based android GUI testing using multi-level GUI comparison criteria. In *ASE*, pages 238–249. ACM, 2016.

[31] Zhen Dong, Marcel Böhme, Lucia Cojocaru, and Abhik Roychoudhury. Time-travel testing of android apps. In *ICSE*, pages 481–492. ACM, 2020.

[32] Nataniel P. Borges Jr., Maria Gómez, and Andreas Zeller. Guiding app testing with mined interaction models. In *MOBILESoft@ICSE*, pages 133–143. ACM, 2018.

[33] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. Humanoid: A deep learning-based approach to automated black-box android app testing. In *ASE*, pages 1070–1073. IEEE, 2019.

[34] Faraz Yazdani Banafshe Daragh and Sam Malek. Deep GUI: black-box GUI input generation with deep learning. In *ASE*, pages 905–916. IEEE, 2021.

[35] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. Record and replay for android: are we there yet in industrial cases? In *ESEC/SIGSOFT FSE*, pages 854–859. ACM, 2017.

[36] Google. monkeyrunner, 2022.

[37] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd D. Millstein. RERAN: timing- and touch-sensitive record and replay for android. In *ICSE*, pages 72–81. IEEE Computer Society, 2013.

[38] Stas Negara, Naeem Esfahani, and Raymond P. L. Buse. Practical android test recording with espresso test recorder. In *ICSE (SEIP)*, pages 193–202. IEEE / ACM, 2019.

[39] Zhengrui Qin, Yutao Tang, Edmund Novak, and Qun Li. Mobiplay: a remote execution based record-and-replay tool for mobile applications. In *ICSE*, pages 571–582. ACM, 2016.

[40] Farnaz Behrang and Alessandro Orso. Automated test migration for mobile apps. In *ICSE (Companion Volume)*, pages 384–385. ACM, 2018.

[41] Markus Ermuth and Michael Pradel. Monkey see, monkey do: effective generation of GUI tests with inferred macro events. In *ISSTA*, pages 82–93. ACM, 2016.

[42] Farnaz Behrang and Alessandro Orso. Test migration between mobile apps with similar functionality. In *ASE*, pages 54–65. IEEE, 2019.

[43] Tomás Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. In *ICLR (Workshop Poster)*, 2013.

[44] Tomás Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality. In *NIPS*, pages 3111–3119, 2013.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018.

[46] Daniel Cer, Yinfei Yang, Sheng-yi Kong, Nan Hua, Nicole Limtiaco, Rhomni St. John, Noah Constant, Mario Guajardo-Cespedes, Steve Yuan, Chris Tar, Brian Strope, and Ray Kurzweil. Universal sentence encoder for english. In *EMNLP (Demonstration)*, pages 169–174. Association for Computational Linguistics, 2018.