Context-aware Bug Reproduction for Mobile Apps

Yuchao Huang[1,2,3], Junjie Wang[1,2,3,*], Zhe Liu[1,2,3], Song Wang[4], Chunyang Chen[5], Mingyang Li[1,2,3], Qing Wang[1,2,3,*]
[1]Science and Technology on Integrated Information System Laboratory,
Institute of Software Chinese Academy of Sciences, Beijing, China;
[2]State Key Laboratory of Intelligent Game, Beijing, China;
[3]University of Chinese Academy of Sciences, Beijing, China; [*]Corresponding author;
[4]Electrical Engineering and Computer Science, York University, Canada;
[5]Monash University, Melbourne, Australia;
{yuchao2019,junjie,mingyang2017,wq}@iscas.ac.cn, liuzhe181@mails.ucas.edu.cn
wangsong@yorku.ca,Chunyang.chen@monash.edu

Abstract—**Bug reports are vital for software maintenance that allow the developers being informed of the problems encountered in the software. Before bug fixing, developers need to reproduce the bugs which is an extremely time-consuming and tedious task, and it is highly expected to automate this process. However, it is challenging to do so considering the imprecise or incomplete natural language described in reproducing steps, and the missing or ambiguous single source of information in GUI components. In this paper, we propose a context-aware bug reproduction approach ScopeDroid which automatically reproduces crashes from textual bug reports for mobile apps. It first constructs a state transition graph (STG) and extracts the contextual information of components. We then design a multi-modal neural matching network to derive the fuzzy matching matrix between all candidate GUI events and reproducing steps. With the STG and matching information, it plans the exploration path for reproducing the bug, and enriches the initial STG iteratively. We evaluate the approach on 102 bug reports from 69 popular Android apps, and it successfully reproduces 63.7% of the crashes, outperforming the state-of-the-art baselines by 32.6% and 38.3%. We also evaluate the usefulness and robustness of ScopeDroid with promising results. Furthermore, to train the neural matching network, we develop a heuristic-based automated training data generation method, which can potentially motivate and facilitate other activities as user interface operations.**

I. INTRODUCTION

Mobile applications (apps) are becoming extremely popular – as of the second quarter of 2022 there are over 3.5 million apps in Google Play's app store[1]. As developers add more features and capabilities to their apps to make them more competitive, the corresponding increase in app complexity has made testing and maintenance activities more challenging. The competitive app marketplace has also made these activities quite important for an app's success. As shown in a survey, 88% of app users would abandon an app if they repeatedly encounter a functionality issue [1]. This motivates developers to rapidly identify and resolve issues, or risk losing users otherwise.

To track and expedite the process of resolving app issues, many modern software projects use bug-tracking systems (e.g., GitHub Issue Tracker [2], Bugzilla [3], Google Code Issue Tracker [4]). These systems allow testers and users to report bugs they have identified in an app. These bug reports are becoming a non-neglectable source of information for improving app quality and user satisfaction. Once developers receive a bug report, one of the first steps to debugging the issue is to reproduce the reported issue as shown in Figure 1. However, this step involves many human efforts, including natural language comprehension, app usage acquisition, and human app interaction. It is expected to relieve developers from this tedious task with an automated approach.

Previous approaches [5]–[8] apply natural language processing (NLP) techniques to match the reproducing steps with app's GUI events (i.e., operations on GUI components, e.g., clicking 'login' button of an app), and employ random or simple guided exploration strategies with the matched information for bug reproduction. Despite their promising performance, there exist the following four major drawbacks which hinder them from achieving higher effectiveness and efficiency.

First, the natural language description of a bug report is inherently imprecise and incomplete [9], therefore the parse and extraction of reproduce-related elements from the reproducing steps can be far from accurate. For example, since the reproducing step *click send text* contains two verbs, the operation object *send text* can hardly be extracted by existing approaches. **Second**, previous approaches mainly utilize the name-related information (e.g., displayed text, content description) of the GUI component to match the reproducing steps, yet our motivational study (in Section II) shows that 75% components would miss the display text, and 81% components miss the content description, which hinders the accurate matching and follow-up reproduction. **Third**, the successful reproduction of a bug report by existing approaches often relies on the accurate and complete reproducing step description. However, 56% of bug reports start from 2-6 steps after app launch, and 15% of reports miss some steps in the middle when describing the bug, according to our observation. **Fourth**, there are similar components on different pages of the app, e.g., text input for username on the login/register page. These further complicate the bug reproducing since existing

[1]https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

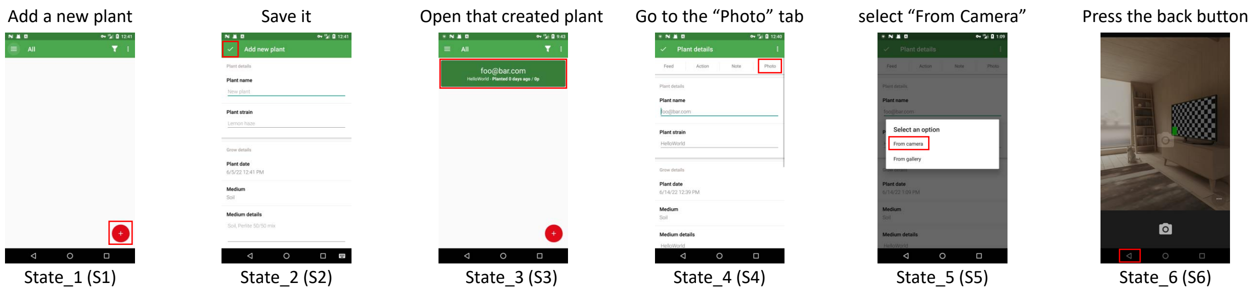


Fig. 1: Examples of reproducing steps and their corresponding GUI events

approaches adopt a greedy-based algorithm when matching the GUI events and the reproducing steps, and the similar matching in the current state is not always the optimal one from the global perspective.

The above-mentioned characteristics of the mobile apps and bug reports motivate us to utilize the context information to make the reproduction more accurate. We utilize the information from two viewpoints. The first is the context information about a specific component (local context), e.g., the nearby components, which facilitates the accurate matching between GUI event and the reproducing steps (tackle the first and second drawback). The second is the context information about a GUI event (global context), e.g., the nearby state transition information of the app, which facilitates the accurate construction of the GUI event sequence for bug reproducing, especially when facing with the missing reproducing steps or similar components in different pages (tackle the third and fourth drawback).

This paper proposes a context-aware bug reproduction approach ScopeDroid which automatically reproduces crash from a text description of mobile bug reports[2]. First, we construct the initial State Transition Graph (STG) with the automated app exploration tool, and extract the textual/icon information of each component and the contextual information from its nearby components. Second, we design a multi-modal neural matching network, which takes the prior extracted information, for predicting the probability of each GUI event being mapped with a certain reproducing step. It derives a fuzzy matching matrix between all candidate GUI events and reproducing steps, which alleviates the influence of the blurred natural language descriptions and the missing name-related information. Third, with the initial STG and fuzzy matching matrix, we plan the path for reproducing the bugs, which considers the global perspective provided by STG and alleviate the influence of missing steps in the report and similar components in the app. Meanwhile, considering the auto-constructed STG can be incomplete, we enrich the initial STG guided by the planned exploration, and iteratively conduct the reproducing step matching, path planning and STG enrichment until a crash occurs.

[2]Note that, we focus on crash reports as previous studies [5] [7] since the reports involving app crashes are of particular concern to developers because it directly impacts an app's usability; and we will use bug report and crash report interactively in the following paper.

To train the neural matching network for step matching, we develop a heuristic-based method for automatically generating the reproducing steps corresponding to certain GUI events following linguistic patterns, which serves as the training data for the automatic model training. The matching between the natural language described operational steps and GUI events is widely utilized in software engineering and human-computer interaction fields, such as user interface operation [10], task shortcuts generation for virtual assistants [11]. Our proposed method for automated training data generation can potentially motivate and facilitate these activities.

To evaluate the effectiveness of our approach, we run ScopeDroid on 102 bug reports collected from 69 popular Android apps, involving three datasets. ScopeDroid successfully reproduces 65 (63.7%) of the crashes, which outperforms the state-of-the-art baselines by 32.6% and 38.3%. The step matching neural network of ScopeDroid can successfully conduct the match at the first recommendation in 53% - 61% cases, and this value is 74% - 85% when considering the first five recommendations, outperforming the state-of-the-art baselines by a large margin. To evaluate the usefulness, we conduct a user study and the results show that ScopeDroid can reproduce 30% more bugs that cannot be reproduced by at least one developer, and is highly preferred by developers in comparison to a manual process. Besides, ScopeDroid is highly robust in terms of different described bug reports.

The contributions of this paper are as follows:

- The design and development of a novel approach to automatically reproduce bugs for mobile apps directly from the textual description of bug reports.
- Experimental evaluation showing that ScopeDroid is effective and useful at bug reproduction, outperforming the state-of-the-art baselines and human reproduction.
- Method for matching natural language described reproducing steps with the GUI event without manually labeled training data, which can potentially motivate and facilitate other tasks as user interface operation [10], and task shortcuts generation for virtual assistants [11].

- Public released source code of ScopeDroid and the dataset of our experiments help other researchers replicate and extend this study [3].

II. Motivation

To reveal the challenges of bug reproduction and motivate the design of our approach, three authors try to reproduce the real-world crashes with the help of existing approaches (i.e., ReCDroid [5] and MaCa [7]). In detail, we choose GitHub as the data source following existing studies [5], [12], as it contains a large number of publicly available valid issue reports. We crawl the issue reports from Android projects and focus on issue reports that are created from Jan. 2015 to May. 2022, resulting in 96,451 issue reports. Note that, not all issue reports are related to app crash, we use keywords, e.g., crash, exception, to search for reports involving app crashes following existing studies [5], [12]. As a result, there is a total number of 6,959 crash reports, and we randomly sample 100 crash bug reports for human reproduction. During the process, the authors reveal the following three findings related to the characteristics of the Android apps and the reproducing steps, which influences the successful reproduction of crashes with existing approaches.

1) ***Name-related information of a GUI component can be missing or misleading****:* Previous studies typically utilize the name (e.g., displayed text, content description, identifier) of a GUI component to match the textual description of reproducing steps, in order to reproduce the crash. For related apps of the collected bug reports, we randomly choose 100 GUI pages with 399 clickable components, and investigate their name-related information. Three authors carry out the process separately, and discussion is conducted until a final consensus is reached. Results indicate that in 75% (299/399) components, the displayed text is missing, and in 81% (325/399) components, the content description is missing. Furthermore, the identifier is missing in 32% (131/399) components, and 21% (58/268) of the remaining identifiers are misleading, e.g., meaningless names like *button1* and *edit2*. This is mainly due to the developers not following the naming conventions, and could seriously influence program comprehension and bug reproducing.

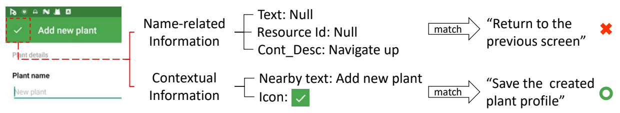


Fig. 2: Illustrative example

On the contrary, we observe that the contextual information of a GUI component can facilitate the recognition of its true intent. We list an example from GrowTracker, an app with 124 stars in Github[4], in Figure 2. For the green icon with the check mark, the name-related information is either missing (e.g., displayed text, resource id) or incorrect (e.g., content description), because of which the matching would go wrong. Nevertheless, the contextual information, i.e., displayed text of its nearby components (*add new plant*) and icon image, can help the correct matching with the reproducing steps.

This motivates us to include contextual information, such as the nearby components and the icon image, for the matching with the reproducing steps.

2) ***Reproducing steps can have missing steps****:* The success reproduction of a bug report by previous studies often relies on the accurate and complete reproducing steps. The quality of the bug report often depends on the experience of the testers, and some crowd testers in an open environment often could not precisely describe all the reproducing steps. Three authors inspect the reproducing steps of the 100 bug reports, and conduct discussion until a final consensus is reached. Results reveal that only 44% reports are described from the launch of the app, while the remaining 56% reports typically start from 2-6 steps after the app launch. Worse more, 15% reports involve missing middle steps when describing the bug.

Since previous studies adopt the greedy-based algorithm when matching the GUI event and reproducing the bug, these missing steps can easily lead the algorithm to get in the wrong direction and significantly reduce the success of the reproduction. In addition, when faced with the missing steps, the optimal matching between GUI event and reproducing steps in the current state is not always the optimal matching from the global perspective.

This motivates us to derive the global matching information between each reproducing step and all GUI events, and employ this information for the exploration path planning from the global perspective.

3) ***There are similar components on different pages of the app****:* Another characteristics of the mobile apps are that different GUI pages have the same or similar components, which makes it challenging for accurate matching with reproducing steps. This is mainly caused by the following three practices: (1) components that play the same role in different GUI pages, e.g., the text input for username on the login/register/reset password page; (2) commonly-used components, e.g., the "ok" or "cancel" on each confirmation box; (3) system keys, the "back" button in the Android system. We name these components as ambiguous components in this scenario. Three authors inspect the collected bug reports and their related apps to find the ambiguous components, and discuss until reaching a final consensus. The results reveal that 12% of bug reports contain at least one step related with the ambiguous components. Together with the situation that reproducing steps can have the missing steps, the existence of ambiguous components makes it even more challenging to precisely match the GUI event with the reproducing step.

This further motivates us to conduct reproduction from a global perspective. Existing approaches adopt the greedy-based algorithm when matching the GUI event and reproducing step, which mainly utilizes the information about current step and cannot combine the previous and subsequent steps

[3]https://github.com/wuchiuwong/ContextAwareReproduction
[4]https://github.com/7LPdWcaW/GrowTracker-Android

for a better decision. We utilize the context information about the nearby state transition information of the app for better constructing the GUI event sequence in reproducing the crash.

In summary, the characteristics of the mobile apps and bug reports motivate us to utilize the context information to make the reproduction more accurate. We utilize the information from two viewpoints. The first is the context information about a specific component (local context), e.g., the nearby components, which facilitates the accurate matching between the GUI event and the reproducing steps (tackle the first and second drawback). The second is the context information about a GUI event (global context), e.g., the nearby state transition information of the app, which facilitates the accurate construction of the GUI event sequence for bug reproducing, especially when facing with the missing reproducing steps or similar components in different pages (tackle the third and fourth drawback).

III. APPROACH

This paper proposes a context-aware automated bug reproduction approach ScopeDroid to reproduce bugs from the natural language described bug reports of mobile apps. Figure 3 shows the pipeline of ScopeDroid, which includes three modules.

STG construction and information extraction (Section III-A). It employs an automated app exploration tool (i.e., Droidbot) to generate the initial state transition graph (STG), and obtain the contextual information of each GUI component (i.e., nearby textual related attributes and its icon image).

Fuzzy reproducing step matching (Section III-B). It designs a multi-modal neural matching network for predicting the probability of each GUI event being mapped to a certain reproducing step. We name this phase as “fuzzy matching” because it does not explicitly map a GUI event to a given reproducing step, rather it derives a matrix considering all the GUI events in an app and each reproducing step, which serves as the input for the path planning.

Path planning and STG enrichment (Section III-C). With the initial STG and the fuzzy matching matrix, it plans the path (i.e., the sequence of GUI events) for reproducing the bugs, and uses it for guiding the app exploration; meanwhile it enriches the initial STG during exploration, and iteratively conducts the path planning and STG enrichment until crash occurs.

A. STG Construction and Information Extraction

*1) **STG Construction**:* The state transition graph (STG) of mobile apps is widely used to illustrate the transitions across different states triggered by typical operations such as *press login button* [13], [14]. In detail, in one STG, a node is an app state, while an edge is a transition between two connected states, which contains an operation and the component on which the operation is conducted. We adopt Droidbot [15], a widely-used automatic dynamic app exploration tool for STG construction.

In detail, for an app, we run Droidbot with a breadth-first search strategy, and obtain all the interactive components (i.e., clickable is true) and their related states. Since the dynamic exploration tool like DroidBot treats any two states with slight differences (e.g. text changes in input boxes) as different states, the generated raw STG is redundant. We then merge the duplicate states to produce a more concise STG. Specifically, following previous works [15]–[17], we obtain the identifier (i.e., *xpath*) for each component in an GUI page (i.e., state), and compute the jaccard distance [18] of the components' *xpath* for two GUI pages. If the distance exceeds a predefined threshold (following existing studies [15], [19], [20], the empirically set as 0.8 in this study, we treat them as duplicate states and merge them into one.

*2) **Information Extraction**:* We extract the textual attributes and icon image of each GUI component for the reproducing step matching.

Textual Information of GUI Components. Following existing studies [21], [22] we extract 10 types of textual information for each GUI component. Among them, 6 types are the basic attributes of the GUI components, i.e., type, displayed text, content description, hint for the input information, resource id used for identification, and absolute location (one of the nine grid locations within the UI page). The remaining 4 types are the component's contextual information derived from its nearby components, i.e., the displayed text of its sibling components, its parent component, its children components, and its neighborhood components.

After extracting the 10 types of textual information, we combine them into a sentence with the format *[key] value*, where the key is the field name of the corresponding information. Taking the confirm button (i.e., green check mark) in Figure 2 as an example, its combined textual information is *[type] imagebutton [displayed text] none [content description] navigate up [hint] none [resource id] none [absolute location] top left corner [neighbor text] plant details [sibling text] add new plant [child text] none [parent text] none*.

Icon Image of GUI Components. When running Droidbot, we can obtain the boundary of each component in the screenshot, with which we then capture the icon image of the component from the screenshot.

B. Fuzzy Reproducing Step Matching

To efficiently map the reproducing steps to the GUI events, we design a multi-modal neural matching network which leverages both the textual information and the icon image. It outputs the matching score of a reproducing step mapped to a GUI event. We also develop a heuristic-based automated training data generation method for facilitating model training.

The reasons why we design such a neural network are threefolds. First, it can mitigate the drawbacks of existing approaches in syntactic parsing the reproducing steps for matching with GUI events, since the parsing is relied on the human-created linguistic rules and is error-prone. Second, it can better capture the semantics of the reproducing steps and GUI components to derive more accurate matching, e.g., “add”

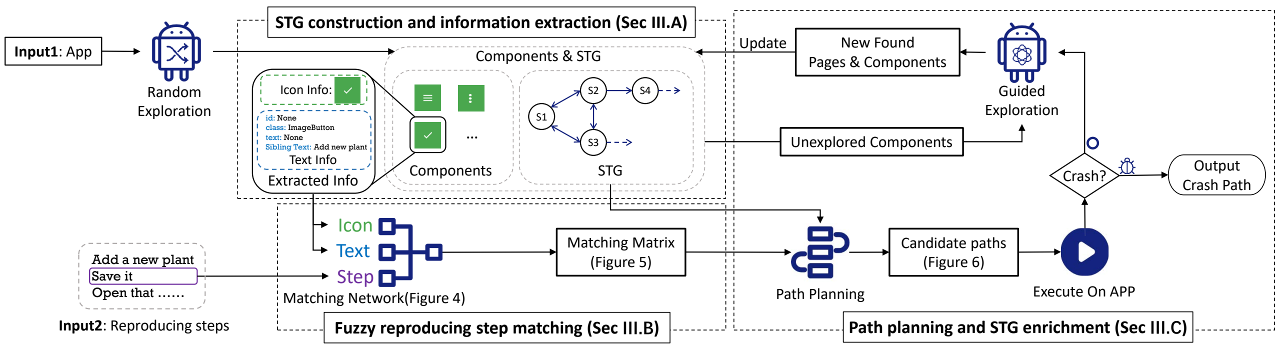

STG construction and information extraction (Sec III.A)
Input1: App
Random Exploration
Components & STG
Icon Info:
id: None
class: ImageButton
text: None
Sibling Text: Add new plant
Text Info
Extracted Info
Components
STG
S1
S2
S3
S4
Update
New Found Pages & Components
Guided Exploration
Unexplored Components
Crash?
Output Crash Path
Icon
Text
Step
Matching Network(Figure 4)
Matching Matrix (Figure 5)
Path Planning
Candidate paths (Figure 6)
Execute On APP
Add a new plant
Save it
Open that ......
Input2: Reproducing steps
Fuzzy reproducing step matching (Sec III.B)
Path planning and STG enrichment (Sec III.C)

Fig. 3: Overview of ScopeDroid

TABLE I: Heuristic-based training data generation

	Button, TextView, ImageView, ImageButton	**CheckBox**	**RadioButton**	**Switch**	**EditText**	**Layout**
Operation	click, launch, choose, press, select, tap, open, click on, clicking, tapping, pressing, go to, going, enter, choose	click, select, tap, click on, turn on, turn off	swipe, switch, turn, enable, disable	click, select, tap, click on	type, write, enter, input, put	click, open, tap, press, click on, choose, select
Component	displayed text, content description, hint, resource id, sibling text, child text, neighbor text	sibling text, neighbor text			resource id, sibling text, parent text, neighbor text	child text, neighbor text
Location	Random choose one, i.e., at/on the top / bottom (left / right (corner)), at/on the left / right / center					
Article	Random add the article (i.e., the, a/an) between operation and operated component, e.g., click *the* login button					
Quotes	Random add quotation mark for the operated component, e.g., click “login”.					
Input	Random add the input for EditText, i.e., type the username, e.g., *foo*, or type *foo* in the username					

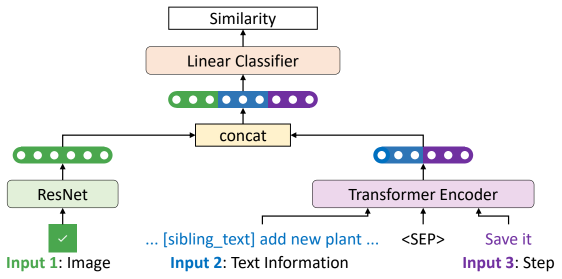
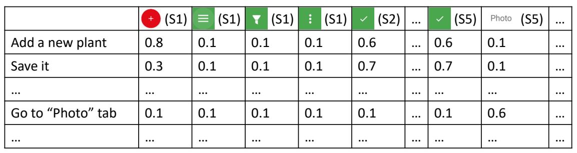


Fig. 4: Architecture of Multi-modal Neural Matching Network



	🟥 (S1)	🟩 (S1)	🟩 (S1)	🟩 (S1)	✓ (S2)	...	✓ (S5)	Photo (S5)	...
Add a new plant	0.8	0.1	0.1	0.1	0.6	...	0.6	0.1	...
Save it	0.3	0.1	0.1	0.1	0.7	...	0.7	0.1	...
...	...	...	...	...	...	...	...	...	...
Go to “Photo” tab	0.1	0.1	0.1	0.1	0.1	...	0.1	0.6	...
...	...	...	...	...	...	...	...	...	...

Fig. 5: Example of similarity matrix between reproducing steps and GUI components

and “save” in the example of Figure 2. Third, the image-like components (e.g. ImageButton) can be modeled and its semantic information can be taken into account which further contributes to the better matching.

*1) **Multi-modal Neural Matching Network**:* As shown in Figure 4, the input of the multi-modal neural matching network is the reproducing step $step$ and the event's related component C (with textual information C_{text} and icon image C_{image} of the component), the network would output the matching score $Sim(step, C)$ of the reproducing step corresponding to the GUI event.

We utilize the Cross-Encoder structure [23], which has been to be effective in information retrieval, question answering, duplicate question detection, etc [24]–[26]. The reproducing step $step$ and the textual information of the component C_{text} are concatenated with symbol $<SEP>$, then input into the pre-trained transformer encoder $TextEncoder$ to generate the hidden state of the text.

$$hidden_{text} = TextEncoder(C_{text}, step) \tag{1}$$

Meanwhile, the image of the component C_{image} is input into the ResNet encoder $ImageEncoder$ for obtaining the hidden state of the image.

$$hidden_{image} = ImageEncoder(C_{image}) \tag{2}$$

Finally the textual hidden state $hidden_{text}$ and the image hidden state $hidden_{image}$ are input into a fully connected layer W for obtaining the match score $Sim(step, C)$.

$$Sim(step, C) = W([hidden_{text}, hidden_{image}]) \tag{3}$$

*2) **Heuristic-based Training Data Generation**:* For training an effective multi-modal neural matching network, we need a large amount of training data with reproducing step and corresponding GUI events, e.g., reproducing step *press the back button* and corresponding event *click back button* in the app as shown in the last step of Figure 1. However, there is no such type of open dataset so far, and collecting the related data from scratch is time- and effort-consuming. Meanwhile, through examining the bug reports in open source projects, we observe that there are certain linguistic patterns when writing the reproducing steps in bug reports. This motivates us in developing a heuristic-based automated training data generation method for collecting the satisfied training data.

The primary idea is that for each interactive component in a GUI page, we heuristically generate the reproducing step which can operate on it for transitioning to the next state; meanwhile, the generated reproducing step pair with other irrelevant components serve as the negative data instances. Take the GUI page of Figure 1 as an example, for the *back* component in the left corner, the generated reproducing step can be *Click back* or *Press back in the left corner*.

To derive the heuristic rules, the first two authors examine 200 open source bug reports (93 apps involved) randomly sampled from the data collected in Section II, summarizing the linguistic patterns for writing the reproducing steps. Note that none of these apps are utilized in our experimental evaluations.

As as shown in Table I, the first two fields, i.e., *operation* and *component* are required, while the others are optional and would be chosen randomly. For *operation*, we summarize a set of verbs for each component category to make the description more vivid, as shown in Table I, and would randomly choose one verb to act as the operation. For *component*, we summarize the data source from which the component name is generated (e.g, the *displayed text* of a button is *login*, the reproducing step can be *click login*). To cope with the empty data source, we list all the candidates and would choose the first non-empty one. We also notice that the data sources are not applicable for all components (e.g., the *text* for CheckBox is *checked/unchecked*), this is why we design a specific list per component category. For *Button* and related component categories, we also randomly add the term *button* after the component name to make it more vivid.

The data generation is based on the Rico [27] dataset which contains more than 66K unique screenshots from 9.3K Android apps, as well as their accompanied JSON file (i.e., the detailed run-time view hierarchy of the screenshot). For each iterative component, we randomly generate three different reproducing steps following the above-mentioned heuristic rules, to serve as the positive data instances in the training data. For the negative data instances, we follow the hard negative mining strategy [28] to enhance the discriminability of the model. In detail, we retrieve four other components similar to the chosen one and pair them with the three generated steps, i.e., twelve negative instances. We use BM25 algorithm [29], [30] to find the similar components by comparing the textual information of each component (details are in Section III-A2).

*3) **Reproducing Step Matching with Neural Matching Network**:* With the auto-generated training data, we train the multi-modal neural matching network. We build our model based on PyTorch [31] and Sentence Transformers [32]. The text processing module is loaded with DistilBert [33], a 12-layer transformer-based pre-training model. The image processing module is loaded with the icon classification ResNet trained by Mehralian et al. [21]. We use AdamW as the optimizer, BCEWithLogitsLoss as the loss function, and train the model with the batch size set to 20.

For each reproducing step and each GUI event of the app, we pair them and input all pairs into the trained model to obtain a similarity matrix like the one shown in Figure 5. This fuzzy matching matrix will be used in the candidate path scoring in the subsequent section.

C. Path Planning and STG Enrichment

With the fuzzy matching matrix, we design a path planning method to derive the GUI event sequence for bug reproduce. The ideal situation for path planning would be to design a dynamic programming algorithm to find the optimal event sequence for reproducing the bug. Yet in real practice, since the initial STG can hardly be completely restricted by current app exploration tools [15]–[17], hence we can only design an algorithm by taking the utmost use of current information. Meanwhile when executing the planned path, we also apply the guided exploration with the reproducing steps for enriching the initial STG and iteratively conduct the patch planning.

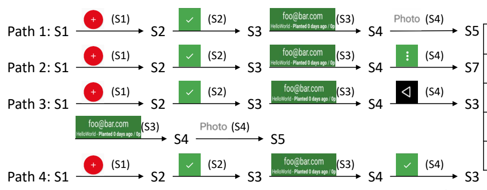
*1) **Path Planning based on Fuzzy Matching Matrix**:* The path planning algorithm is designed as a weighted algorithm of three metrics, i.e., matching accumulation, length penalty, and goal count, for all candidate GUI event sequences. This is designed from the perspective of goal achievement and redundancy reduction when considering the incomplete build STG.

Matching accumulation S_{MA} measures the sum of the matching score (obtained in the fuzzy matching matrix) for the involved GUI events and the reproducing steps. Since the *matching accumulation* would assign a higher score to the longer exploration sequence, **Length penalty** S_{LP} punishes the case of a meaningless long exploration loop. Note that, there are usually looped reproducing steps involving repeated exploration, thus the exploration loop is acceptable, yet we should avoid the meaningless loop to optimize the exploration. **Goals scored** S_{GS} measures how many reproducing steps are explored. It reflects the distance from achieving the final reproduction.

We take the four generated candidate paths in Figure 6 as examples to better illustrate how these three metrics are calculated. Path 1 is the most optimal of these four paths, achieving more steps than the others and without redundant operations. Path 2 involves one less GUI event that matches the reproducing step compared to Path 1, so its *matching accumulation* is lower than Path 1. Path 3 has a redundant operation after the first arrival at State_4 and suffers a higher *length penalty*. Path 4 contains four GUI events matching the reproducing steps like path 1, but the path only completes the first three steps in the reproducing steps, so its *goals scored* is lower than Path 1.

Finally, we take the weighted sum of these three metrics as the path score:

$$Score_{path} = S_{MA} + \alpha S_{LP} + \beta S_{GS} \tag{4}$$



Path 1: S1 (S1) S2 (S2) S3 foo@bar.com (S3) S4 Photo (S4) S5
Path 2: S1 (S1) S2 (S2) S3 foo@bar.com (S3) S4 (S4) S7
Path 3: S1 (S1) S2 (S2) S3 foo@bar.com (S3) S4 (S4) S3
foo@bar.com (S3) S4 Photo (S4) S5
Path 4: S1 (S1) S2 (S2) S3 foo@bar.com (S3) S4 (S4) S3
HelloWorld - Planted 0 days ago / 0p

Name	S_{MA}	S_{Lp}	S_{GS}
Path 1	0.8 (●) + 0.7 (✓) +0.1 (foo@bar.com HelloWorld - Planted 0 days ago / 0p) + 0.6 (Photo) = 2.2	4	4 (Finish Step 1, 2, 3, 4)
Path 2	0.8 (●) + 0.7 (✓) +0.1 (foo@bar.com HelloWorld - Planted 0 days ago / 0p) + 0.1 (⋮) = 1.7	4	3 (Finish Step 1, 2, 3)
Path 3	0.8 (●) + 0.7 (✓) +0.1 (foo@bar.com HelloWorld - Planted 0 days ago / 0p) + 0.1 (◁) + 0.1 (foo@bar.com HelloWorld - Planted 0 days ago / 0p) + 0.6 (Photo) = 2.4	6	4 (Finish Step 1, 2, 3, 4)
Path 4	0.8 (●) + 0.7 (✓) +0.1 (foo@bar.com HelloWorld - Planted 0 days ago / 0p) + 0.7 (✓) = 2.3	4	3 (Finish Step 1, 2, 3)

Fig. 6: Example of scoring candidate paths

When choosing the parameter values, we consider two main balances: the balance between avoiding path redundancy and encouraging long path exploration (controlled by S_{LP}), and the balance between encouraging exploration of paths that achieve goals and avoiding wasting time on paths that are wrongly achieved (controlled by S_{GS}). To determine the parameter values, we tune our approach on 17 other "crash" reports which are excluded from the experimental data. These "crash" reports are obtained during our data collection process (details in the Section IV-A). They have clear reproducing steps, but we fail to trigger crashes after manual reproduction due to different hardware configurations or other engineering reasons. We experiment with α from -2 to -0.5 with an increment of 0.5 in between, and β from 1 to 5 with an increment of 1 in between. We run the approach with each group of parameters (α and β), and record the reproduction success rate; then we choose the parameter value when the highest success rate being achieved. After parameter tuning, α is set as -1 and β is set as 3 in our experiment. Based on this, we choose the path having the highest score for execution or guided exploration. We execute the planned path on the simulator automatically through Appium [34].

2) ***STG Enrichment**:* Since existing automated app exploration tools can hardly explore all app states, and the built STG is incomplete, i.e., some pages are not explored, or some components in an explored page are not interacted.

Therefore, during the exploration process, we would enrich the STG to include more states, update the path planning and facilitate the bug reproduction. Generally speaking, guided by the current planned path, we would include the unexplored components on newly touched states, and the subsequent state of the component which matches a reproducing step. After the enhancement, we calculate the matching scores of the newly discovered components with each reproducing step and update the fuzzy matching matrix. Then we re-plan the execution sequence and conduct the exploration.

IV. EXPERIMENT DESIGN

To evaluate ScopeDroid, we consider four research questions:

RQ1: How effective and efficient is ScopeDroid at reproducing crashes in bug reports?

RQ2: How accurate is the step matching?

RQ3: Does ScopeDroid benefit developers compared to manual reproduction?

RQ4: Is ScopeDroid robust in terms of different described bug reports?

A. *Experimental Dataset*

We employ the crash reports from three sources for evaluation, i.e., ReCDroid's dataset [5], AndroR2 dataset [12], and GitHub. For the crash reports used by ReCDroid [5], we employ all the 33 reports provided in their replicate package for the experiment. For the reports contained in AndRoR2 dataset [12], we use all its 22 crash reports. Other reports, e.g., display issue reports, are out of the scope of this study. For 6,959 crash reports crawled and filtered from GitHub as described in Section II, we randomly sampled 1500 of them and conduct manual checking to get the final dataset. This manual filtering is performed independently by three graduate students with 2-4 years of industrial software development experiment, and each report is manually reproduced by two of them. We exclude those that could not be reproduced (e.g., lack of apks, failed-to-compile apps, environment issues) or required special conditions (e.g., account, hardware, data). This results in 47 crash reports from 28 apps, and we refer to this dataset as ScopeDroid's dataset. This process also yields 17 "crash" reports, which have clear reproducing steps, but we fail to trigger crashes due to different hardware configurations or other engineering reasons. These "crash" reports are only used for parameter tuning as shown in Section III-C and excluded from the experimental data. Note that, we manually extract the reproducing steps from the crash reports, which is a commonly-used practice in existing studies [5], [7]. Due to space limitations, the details of the data (e.g., steps per report, pages/components per app) can be viewed on our website[3].

B. *Baselines*

To demonstrate the advantages of our proposed ScopeDroid, we compare with 2 state-of-the-art approaches.

ReCDroid [5]: It is the state-of-the-art approach for reproducing crashes from natural language described reproducing steps. It leverages dependency parsing and predefined grammar patterns to extract GUI event representations from the reproducing steps. Then it conducts the exploration guided by the matching status between the extracted GUI event representations and the displayed text of GUI components. Note that the same authors proposed an improved tool called ReCDroid+ [35], which treats the raw bug reports as input and focuses on the automatic extraction of the reproducing steps from the raw bug reports. Except for the initial extraction of reproducing steps, RecDroid+ is the same as ReCDriod. ScopeDroid takes the reproducing steps rather than the raw bug reports as input, so we do not select ReCDriod+ but only ReCDroid as the baseline. We compare it for effectiveness and efficiency of bug reproduction (RQ1) and for the accuracy of step matching (RQ2).

MaCa [7]: It is proposed to automatically identify and classify the action words in bug reports in order to boost the performance of bug reproduction tools like ReCDroid. It first identifies action words based on natural language processing, then extracts its related information (i.e., enclosing segment, associated UI target); this information is fed into a machine learner to predict the category of the action word (e.g., click, type). We treat this approach as the baseline for step matching (RQ2). In addition, we integrate it with ReCDroid as described in their paper (denoted as 'ReCDroid+MaCa') which serves as the baseline for RQ1.

C. *Experimental Setup and Evaluation Metrics*

For RQ1, we verify the effectiveness and efficiency of ScopeDroid in two aspects: (1) the percentage of reports that can be successfully reproduced in given time (denoted as *success rate*), i.e., two hours following previous practice [5]. (2) The time required for successful reproduction (denoted as *reproducing time*).

For RQ2, we validate whether our matching module can accurately match the GUI event of the app with the reproducing steps in the report. For the 67 crash reports that can be successfully reproduced by at least one tool, we first obtain all UI pages related to the reports with automatic exploration (DroidBot) and manual supplementation. We then run the matching module of ScopeDroid and the baselines to calculate the similarity between each reproducing step and GUI event on all found pages. Ideally, the target GUI event should get a higher score and be ranked at the top. Therefore, we evaluate the matching modules with the following two evaluation metrics. **Hit@k** checks whether the recommended top-k GUI events contain the correct one which matches the given reproducing step. We set k as 1, 3, 5, and 10 to obtain a relative thorough view. **MRR (Mean Reciprocal Rank)** is the mean of Reciprocal Rank (RR) values obtained for all reproducing steps. RR of a single reproducing step is multiplicative inverse of the rank of corresponding component.

For RQ3, we evaluate the usefulness of ScopeDroid. We invite 21 graduate students to participate in this experiment. All of them have experience in mobile application testing, 12 are Android developers with at least 5 years of development experience, and 10 work in the crowdtesting platform. For the 67 crash reports that can be reproduced by at least one tool, we divide them into 7 groups with each group having 10 reports. Each participant is assigned to three groups of reports, i.e., ensuring that each test report is manually reproduced by 3 participants. Participants need to reproduce these reports manually on the emulator with the corresponding app already installed, and we record the time cost. If participants are unable to reproduce within 30 minutes following existing study [5], the report is marked as not reproduced. After participants manually reproduce their reports, we show them video of ScopeDroid in reproducing these crashes and ask their opinions about it with 5-Likert scale [36], i.e., very useful, useful, neutral, not useful, and useless.

TABLE II: Performance of reproduction success rate

# Dataset	ReCDroid	ReCDroid+Maca	ScopeDroid
ReCDroid's Dataset (33 Reports)	**32 (97%)**	**32 (97%)**	30 (91%)
AndroR2 Dataset (22 Reports)	6 (27%)	6 (27%)	**11 (50%)**
ScopeDroid's Dataset (47 Reports)	9 (19%)	11 (23%)	**24 (51%)**
# Total (102 Reports)	47 (46%)	49 (48%)	**65 (63%)**

For RQ 4, we verify the robustness of the ScopeDroid. Since reproducing steps are typed manually, there may be inaccuracies in describing the GUI events, such as missing words, using synonyms, and improper word order, which may affect the success of the automatic reproduction. To simulate the above problem, we create variants of reproducing steps by using EDA [37], a common-used data augmentation tool that generates semantically similar but textually different samples. In detail, we set two degrees of mutation, i.e., randomly changing 10%/20% (a large mutation rate can easily lead to the semantic change) of the words in reproducing steps by performing four commonly-used operations: synonym replacement, random insertion, random swap and random deletion. We conduct this experiment on the 30 successfully reproduced reports in ReCDroid's dataset, which already exerts diversified step length and reproduction time. We randomly generate three variants for each degree of variation (a total of $2 \times 3 \times 30 = 180$), run ScopeDroid on these variants and record reproduction results.

V. Results and Analysis

A. *RQ 1: Effectiveness and Efficiency of ScopeDroid*

Table II shows the success rate of reproducing crash reports from three datasets. Overall, ScopeDroid can reproduce 63.7% of them (65 out of 102), outperforming the baselines by a large margin, i.e., 32.6% (63.7 vs. 48.0) higher than ReCDroid+MaCa, and 38.3% (63.7 vs. 46.0) higher than ReCDroid. This indicates that ScopeDroid is more effective in reproducing crash reports. ScopeDroid successfully reproduces 30 reports on the ReCDroid's Dataset, and the other three failed cases are due to the incompatibility with our environment or lack of a valid account. While on the AndroR2 dataset and the ScopeDroid's dataset, ScopeDroid can achieve a success rate of about 50%. In contrast, ReCDroid or ReCDroid+MaCa can only successfully reproduce a small portion of reports with simple reproducing steps, with the success rate of 19% to 27%.

Table III shows the reproducing time of ScopeDroid and two baselines on each report. Note that MaCa takes less than a second to analyze the bug report, which is negligible and we treat the reproducing time of ReCDroid+MaCa the same as ReCDroid.

For the 45 crash reports that the baselines can reproduce, the average reproducing time of ReCDroid is 440 seconds, while ScopeDroid's average time is 83 seconds which is faster than ReCDroid. The above comparison is conducted when not considering the pre-exploration time of DroidBot. This implies

TABLE III: Detailed reproduction results on three datasets. **RD**, **RD+M**, **SD** in the table header refers to ReCDroid, ReCDroid+MaCa and ScopeDroid respectively. **p** refers to the number of participants who successfully reproduced the report, and **pt** refers to the average time it takes for the participants to reproduce the report, × means that the tool fails to reproduce a report within the time limit. Note that in order to save space, reports that cannot be reproduced by either approach are omitted in this table, and can be find in our website.

										ReCDroid's Dataset													
id	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**			
1	NewsBlur-1053	210	210	53	3	90	12	FastAdapter-394	29	29	25	3	118	23	ownCloud-487	74	74	16	3	17			
2	Markor-194	2185	2185	135	2	149	13	LibreNews-22	186	186	58	2	51	24	OBDReader-22	1581	1581	523	3	190			
3	Birthdroid-13	147	147	37	3	120	14	LibreNews-23	49	49	68	3	205	25	Dagger-46	18	18	12	3	12			
4	Car Report-43	487	487	113	2	164	15	LibreNews-27	112	112	61	3	90	26	ODK-2086	99	99	83	3	122			
5	Sudoku-173	917	917	181	3	137	16	SMSsync-464	1167	1167	75	1	312	27	K-9Mail-3255	185	185	50	3	155			
6	ACV-22	500	500	×	3	88	17	Transistor-63	43	43	23	3	120	28	K-9Mail-2612	103	103	38	3	96			
7	AnyMemo-18	69	69	38	3	32	18	Zom-271	88	88	21	2	123	29	K-9Mail-2019	55	55	19	2	46			
8	AnyMemo-440	1531	1531	134	3	251	19	Pix-Art-125	891	891	35	3	192	30	Anki-4586	90	90	423	2	46			
9	Notepad-23	348	348	73	3	34	20	Pix-Art-127	191	191	33	3	64	31	TagMo-12	27	27	24	2	54			
10	Olam-2	58	58	13	3	142	21	ScreenCam-25	760	760	20	3	45	32	FlashCards-13	20	20	64	3	135			
11	Olam-1	27	27	14	3	35	22	Ventriloid-1	69	69	×	3	110										
										AndroR2 Dataset													
id	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**			
33	HABPanel-25	×	×	35	3	55	37	OpenMap-1030	314	314	430	3	90	41	Hex-9	12	12	155	3	56			
34	Noad Player-1	11	11	10	2	40	38	andOTP-500	×	×	536	3	141	42	Firefox-3932	3310	3310	98	2	120			
35	Weather-61	×	×	20	3	71	39	K-9Mail-3255	185	185	50	3	46	43	Aegis-3932	×	×	1044	2	249			
36	Berkeley-82	32	32	14	3	54	40	K-9Mail-3971	×	×	117	3	53										
										ScopeDroid's Dataset													
id	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**	**id**	**#Bug Reports**	**RD (s)**	**RD+M (s)**	**SD (s)**	**p**	**pt (s)**			
44	NewPipe-7825	×	105	60	1	80	52	Anki-3370	×	×	165	2	157	60	nRF Mesh-495	1342	1342	96	3	43			
45	SDBViewer-10	×	×	73	3	85	53	Anki-2765	37	37	81	3	176	61	SDBViewer-7	×	×	13	3	20			
46	Anki-9914	×	×	150	2	55	54	Anki-2564	×	×	268	1	105	62	FakeStandby-30	×	26	17	2	71			
47	Anki-10584	57	57	14	3	100	55	Anki-2681	×	×	455	3	196	63	pedometer-101	82	82	60	3	39			
48	Alarmio-47	×	×	35	3	51	56	WhereUGo-368	70	70	55	2	124	64	Revolution-183	2521	2521	135	3	41			
49	plusTimer-19	×	×	172	2	93	57	FoodTracker-55	72	72	37	3	61	65	Anki-3224	×	×	125	3	48			
50	GrowTracker-89	×	×	196	1	537	58	GrowTracker-87	×	×	259	3	218	66	getodk-219	13	13	42	3	152			
51	Shuttle-456	×	×	213	2	336	59	Markor-1565	×	×	346	3	141	67	Anitrend-110	11	11	10	3	13			

that our proposed approach has an advantage in efficiency when the STG of an app being freely available, e.g., being extracted in advance.

We admit that the pre-exploration by Droidbot is time-consuming, and ScopeDroid may be slower than ReCDroid when considering the time cost of pre-exploration. But note that the pre-exploration is a one-time job for an app and can be conducted in advance. Meanwhile, this time can be averaged into bug reports of an app and becomes even negligible. Taken in this sense, ScopeDroid has an efficiency advantage over baselines in the scenario of an app having multiple crash reports. Take the app Pix-Art which has multiple reports as an example (e.g., *Pix-Art-125,127* in ReCDroid's dataset), we only need to run DroidBot once for reproducing these two reports and its running time can be shared, which makes our approach faster ($600/2s + 83s = 383s < 440s$).

The reason why ScopeDroid performs better than the baselines is mainly due to the following three reasons. First, our approach develops a multi-modal neural matching network to match the reproducing steps with the GUI event, which can achieve a higher matching accuracy than the baselines which mainly utilize name-related information and natural language parsing. Second, ScopeDroid can conduct the exploration from a relatively global perspective, while the baselines use a greed-based exploration strategy which may waste time on repeated exploration over certain pages and fail to reach the status related to crash. Third, our approach supports more types of GUI events, e.g., scrolling up and down the page, which are not included by baselines. We also check the reasons for failure cases and summarize them in Section VI-A.

B. RQ 2: Matching Accuracy

Table IV shows the performance of the step matching module for each approach. Our step matching module can accurately match the reproducing step with GUI event at the first recommended result (i.e., Hit@1) on 61%, 55%, and 53% cases respectively on the three datasets. This indicates more than half of the cases, the reproducing step can be correctly matched to the GUI event, which facilitates path planning and crash reproduction. When considering the first five recommended results (i.e., Hit@5), the performance is 85%, 83%, and 74%. This indicates ScopeDroid can relatively accurately match the reproducing steps with GUI events, which constructs a solid foundation for crash reproduction.

The step matching of ScopeDroid outperforms the baselines by a large margin, i.e., 17%-26% higher than ReCDroid in Hit@1, and 33%-36% higher than MaCa in Hit@1. The reasons for the advantages of our approach are two folds. First, we develop a multi-modal matching network which can take advantage of the context information of the GUI components. By comparison, the baselines only utilize the name-related information, which would fail in certain types of components like *CheckBox*, *ImageButton*. Second, our developed network does not require linguistic patterns for step matching, which could mitigate the issues brought by the incorrect grammar parsing.

C. RQ 3: Usefulness of ScopeDroid

The last two columns of Table III show the results of participants' manual reproduction of bug reports. For the 67 reports that at least one tool can reproduce successfully, there are 30% (20/67) reports that failed to be reproduced by at least one developer, indicating that ScopeDroid can reproduce crashes that human testers cannot reproduce, and ScopeDroid reproduces bugs faster than testers on 42 reports. Although ScopeDroid does not perform as well as human testers on some complex reports, it is still useful in many cases, where ScopeDroid can automatically finish the tedious

TABLE IV: Performance of the matching module for each approach

# Match Module	ReCDroid's Dataset					AndroR2 Dataset					ScopeDroid's Dataset				
	Hit@1	**Hit@3**	**Hit@5**	**Hit@10**	**MRR**	**Hit@1**	**Hit@3**	**Hit@5**	**Hit@10**	**MRR**	**Hit@1**	**Hit@3**	**Hit@5**	**Hit@10**	**MRR**
ReCDroid	0.35	0.44	0.52	0.54	0.411	0.38	0.44	0.55	0.55	0.441	0.29	0.36	0.36	0.36	0.319
Maca	0.27	0.31	0.35	0.42	0.307	0.22	0.38	0.44	0.55	0.33	0.17	0.21	0.23	0.36	0.213
ScopeDroid	**0.61**	**0.77**	**0.85**	**0.91**	**0.709**	**0.55**	**0.83**	**0.83**	**0.94**	**0.682**	**0.53**	**0.7**	**0.74**	**0.78**	**0.62**

TABLE V: Performance of robustness evaluation of ScopeDroid. The **Ori** column shows the reproducing time of ScopeDroid on the original report. The columns **10%** and **20%** under the **Degree of mutation** show the average time of ScopeDroid on the variants, and the number in parentheses is the number of successfully reproduced variants (out of 3)

#Bug Reports	Ori	Degree of mutation		#Bug Reports	Ori	Degree of mutation	
		10%	**20%**			**10%**	**20%**
NewsBlur-1053	53	52 (3)	52 (3)	Transistor-63	23	24 (3)	22 (3)
Markor-194	135	137 (3)	135 (1)	Zom-271	21	21 (3)	36 (3)
Birthdroid-13	37	46 (3)	46 (3)	Pix-Art-125	35	117 (3)	115 (3)
Car Report-43	113	111 (3)	112 (3)	Pix-Art-127	33	259 (3)	241 (3)
Sudoku-173	181	216 (3)	187 (1)	ScreenCam-25	20	19 (3)	19 (1)
AnyMemo-18	38	43 (3)	39 (3)	ownCloud-487	16	54 (3)	57 (3)
AnyMemo-440	134	130 (2)	188 (1)	OBDReader-22	523	523 (2)	Na (0)
Notepad-23	73	70 (3)	70 (3)	Dagger-46	12	12 (3)	12 (3)
Olam-2	13	15 (2)	18 (2)	ODK-2086	83	94 (3)	146 (3)
Olam-1	14	16 (3)	14 (3)	K-9Mail-3255	50	48 (3)	200 (3)
FastAdapter-394	25	85 (3)	24 (3)	K-9Mail-2612	38	42 (3)	42 (3)
LibreNews-22	58	225 (3)	Na (0)	K-9Mail-2019	19	19 (3)	21 (3)
LibreNews-23	68	72 (2)	Na (0)	Anki-4586	423	422 (3)	423 (2)
LibreNews-27	61	62 (2)	72 (2)	TagMo-12	24	25 (3)	29 (3)
SMSsync-464	75	80 (3)	80 (2)	FlashCards-13	64	66 (3)	78 (3)

and time-wasting process of exploring the app and finding the components involved, saving the developer effort and time.

After manually reproducing the bug reports, we invite all participants to rate ScopeDroid. ScopeDroid achieves a mean score of 4.14 (between *very useful* and *useful*). Most participants comment that *ScopeDroid is accurate in matching the GUI event with reproducing step*. One participant complain that *it takes a lot of time to find the component corresponding to the first step because some previous steps are omitted*, and ScopeDroid can potentially save time by helping her locate the component in the first step. Another crowdsourcing developer is interested in ScopeDroid. She thinks *ScopeDroid could be used in the future to complete test report information by adding screenshots or GIFs for some text-only steps, thus allowing other crowdsourcing participants to quickly understand the bugs found to reduce duplicate submissions*.

D. RQ 4: Robustness of ScopeDroid

On the sampled dataset, our method achieves success rates of 94.4% (85 out of 90) and 76.7% (69 out of 90) at the mutation levels of 10% and 20% respectively. And the slowdowns caused by the mutation with respect to the original bug reports are only 1.26 times and 1.37 times, respectively. This indicates that our approach has good robustness and can resist the disruption of missing words or word changes to some extent. We check the variants that fail to reproduce, and the results reveal that these failure cases are mainly because of the significant changes in the meaning of the reproducing step. For example, when *enable automatically refresh* in LibreNews-23[5] is changed to *enable mechanically brush up* with the EDA technique in Section IV-C, even the human can hardly understand what its actual meaning.

[5]https://github.com/milesmcc/LibreNews-Android/issues/23

VI. DISCUSSION

A. Limitations

There are three main limitations of ScopeDroid which hinder it from reproducing all bug reports, which also indicates the challenges in bug reproduction of mobile apps and calls for further research.

First, ScopeDroid assumes that a reproducing step refers to a single interaction with the GUI component, yet this assumption is not always true in real-world practice. Reporters may abbreviate commonly-known processes into one step, e.g., *create a contact* corresponds to three GUI events *go to the contact create page, enter contact information, click save*. Besides, ScopeDroid cannot correctly understand the steps that require inference, e.g., *click the big button*, although these steps are intuitive to human testers. This common-sense knowledge needs to be further learned and incorporated into the approach for facilitating reproduction.

Second, ScopeDroid cannot reproduce the bugs which involve specific prior knowledge, e.g., entering a valid username and password to log in, which requires human intervention and cannot be fully automated.

Third, ScopeDroid cannot cover all the actions at the execution time. It already supports the GUI events like tapping, long pressing and typing, rotating the screen, scrolling up and down, restarting the app, etc. There are other operations not covered due to the limitations of emulator or engineering issues, e.g., using hardware that the emulator does not have (e.g., Bluetooth, NFC) and fast continuous clicking.

B. Generality across Platforms

The whole technical idea of ScopeDroid is platform-independent and can be trivially applicable to other platforms (e.g., IOS and web). Nevertheless, we utilize two platform-dependent tools respectively for STG contribution (Droidbot [15]) and execution on app (Appium [34]), and need to be replaced with their alternative tools when extending to other platforms.

In detail, the *STG construction* module (Section III-A) employs Droidbot [15], which is a testing tool for app exploration and designed specifically for Android apps. For IOS or web apps, one can choose Monkey [38] as an alternative. In addition, the *path planning and STG enrichment* module (Section III-C) utilize Appium [34], which is an automation framework for executing operations and obtaining screen layout on the emulator, and designed for Android and IOS apps. For web apps, one can turn to Selenium [39].

Furthermore, in the *fuzzy reproducing step matching* module (section III-B), we train a matching network for predicting the probability of each Android GUI event being mapped to a step, which is theoretically platform-independent. Yet to further achieve better results, one is encouraged to fine-tune the model with the data from the application scenarios in case of different naming conventions; for example, the class of the basic button is named "Button" on Android while "XCUIElementTypeButton" on IOS.

Like existing studies [5], [7], [35], this study focuses on Android apps because the bug reports are easier to fetch. We'll explore its feasibility on other platforms in the future.

C. Threats to Validity

The first threat relates to the implementation of the baselines. We reuse the source code provided on their website, and carefully follow their instruction for conducting experiments on the new datasets. The second threat is about the representativeness of the data used in the evaluation. We collect the bug reports from GitHub submitted from 2015 to 2022, and follow the procedures in previous studies to filter and select the dataset. The third threat relates to the confounding effects of participants. Following the existing approach [5], [35], we assume that students with Android programming experience can be substituted for testers, and their reproducing time and success rate are representative. The fourth threat is that the experimental result is only based on the 10-minute random exploration by an automated testing tool. More experiments (e.g., exploration time is 5 minutes) can be conducted in the future to further evaluate the effectiveness of our approach.

VII. Related Work

Mobile Bug Reports Analysis and Reproducing. Several works focus on using NLP techniques to extract critical information from bug reports, such as summarizing and classifying bug reports [40], [41], facilitating dynamic analysis [42], [43], augmenting bug reports for mobile apps [7], [44], [45] and generating test cases [44], [45]. For example, Yakusu [6] used a combination of program analysis and natural language processing techniques to generate executable test cases from bug reports. Maca [7] leveraged the pre-training language model to identify and classify action words in bug reports. The above works are close to our approach, but their goal is not to reproduce the reported bugs.

The most closely related work that focuses on reproducing the bug from the mobile bug report is ReCDroid [5], ReCDroid+ [35] and GIFdroid [46]. Specifically, GIFdroid [46] leveraged the visual screen record to perform reproduction. ReCDroid [5] leveraged the natural language described reproducing steps to perform reproduction. It designed a set of predefined grammar patterns to extract events and objects from textual reproducing steps and then adopted a greedy-based dynamic exploration to synthesize event sequences. The same authors later proposed an improved version, i.e., ReCDroid+ [35], which introduced heuristic rules and neural network classifiers to automatically extract the reproducing steps from raw bug reports, and reused ReCDroid for bug reproduction. Compared with the matching method in RecDroid and RecDroid+, ScopeDroid develops a novel matching method that first automatically extracts various information of GUI components, e.g., icon and contextual information, and then uses a multi-modal neural network to match reproducing steps to GUI events. Our matching method has two advantages: first, it does not need dependency parsing and has better generalization; second, it utilizes more information than the text of the GUI components, i.e., icon and contextual information. Meanwhile, ScopeDroid also has advanced path planning strategies than the two approaches.

Besides, there are several works focus on assisting in recording and replaying bugs in mobile and web apps by using running information [47]–[50], textual description [51]–[53].

This study contributes to this direction by proposing a novel approach for bug reproduction with promising results.

GUI Component Understanding. Several tools focus on the accessibility of the user interface [17], [54]–[59] and filling in missing descriptions for GUI components [21], [60], [61]. LabelDroid [60] automatically predicted the text labels of image-based buttons by learning from large-scale commercial apps in Google Play. COLRA [21] leveraged the contextual information of components to generate non-predefined text labels more accurately. In contrast to the above works, our approach focuses on matching GUI components to text descriptions rather than generating text descriptions. Besides, there are several works [22], [62], [63] focused on test migration on mobile apps and matching semantically similar GUI components between apps. This study designs a multi-modal reproducing step matching method with GUI events, which can also motivate the studies about the GUI component understanding.

VIII. Conclusion

In order to improve the effectiveness and benefit of automatic reproduction, this paper proposes a context-aware bug reproduction approach ScopeDroid which automatically reproduces crashes from text descriptions of mobile bug reports. We first construct the STG and extract the contextual information, then design a multi-modal neural matching network to match GUI events and reproducing steps. Using the initial STG and the matching information, we plan the path for reproducing the bugs, and enrich the STG iteratively. We evaluate the ScopeDroid on 102 bug reports from 69 popular Android apps, it can successfully reproduce 65 (63.7%) of the crashes, largely outperforming the baselines. The evaluation of usefulness and robustness of ScopeDroid also demonstrates promising results.

IX. Acknowledgments

This work was supported by the National Natural Science Foundation of China Grant No.62232016, No.62072442 and No.62272445and Youth Innovation Promotion Association Chinese Academy of Sciences.

References

[1] "Applause," https://www.applause.com/blog/app-abandonment-bug-testing.

[2] "Github," https://appium.io/.

[3] "Bugzilla keyword descriptions," https://bugzilla.mozilla.org/describekeywords.cgi.

[4] "Google code," https://code.google.com.

[5] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. J. Halfond, "Recdroid: automatically reproducing android application crashes from bug reports," in *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 128–139.

[6] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, F. Tip and E. Bodden, Eds. ACM, 2018, pp. 141–152.

[7] H. Liu, M. Shen, J. Jin, and Y. Jiang, "Automated classification of actions in bug reports of mobile apps," in *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds. ACM, 2020, pp. 128–140.

[8] S. Li, J. Guo, M. Fan, J. Lou, Q. Zheng, and T. Liu, "Automated bug reproduction from user reviews for android applications," in *ICSE-SEIP 2020: 42nd International Conference on Software Engineering, Software Engineering in Practice, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 51–60. [Online]. Available: https://doi.org/10.1145/3377813.3381355

[9] V. Ambriola and V. Gervasi, "Processing natural language requirements," in *1997 International Conference on Automated Software Engineering, ASE 1997, Lake Tahoe, CA, USA, November 2-5, 1997*. IEEE Computer Society, 1997, pp. 36–45.

[10] Y. Li, J. He, X. Zhou, Y. Zhang, and J. Baldridge, "Mapping natural language instructions to mobile UI action sequences," in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL 2020, Online, July 5-10, 2020*, D. Jurafsky, J. Chai, N. Schluter, and J. R. Tetreault, Eds. Association for Computational Linguistics, 2020, pp. 8198–8210. [Online]. Available: https://doi.org/10.18653/v1/2020.acl-main.729

[11] D. Arsan, A. Zaidi, A. Sagar, and R. Kumar, "App-based task shortcuts for virtual assistants," in *UIST '21: The 34th Annual ACM Symposium on User Interface Software and Technology, Virtual Event, USA, October 10-14, 2021*, J. Nichols, R. Kumar, and M. Nebeling, Eds. ACM, 2021, pp. 1089–1099. [Online]. Available: https://doi.org/10.1145/3472749.3474808

[12] T. Wendland, J. Sun, J. Mahmud, S. H. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini, "Andror2: A dataset of manually-reproduced bug reports for android apps," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 600–604.

[13] Y. Zhang, Y. Sui, and J. Xue, "Launch-mode-aware context-sensitive activity transition analysis," in *Proceedings of the 40th International Conference on Software Engineering*, 2018, pp. 598–608.

[14] S. Yang, H. Wu, H. Zhang, Y. Wang, C. Swaminathan, D. Yan, and A. Rountev, "Static window transition graphs for android," *Automated Software Engineering*, vol. 25, no. 4, pp. 833–873, 2018.

[15] Y. Li, Z. Yang, Y. Guo, and X. Chen, "Droidbot: a lightweight ui-guided test input generator for android," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 23–26.

[16] T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based gui testing of android apps," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245–256.

[17] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 596–607.

[18] P. Jaccard, "Étude comparative de la distribution florale dans une portion des alpes et des jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[19] M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153–164.

[20] R. Yandrapally, A. Stocco, and A. Mesbah, "Near-duplicate detection in web app model inference," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 186–197.

[21] F. Mehralian, N. Salehnamadi, and S. Malek, "Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in android apps," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 107–118.

[22] L. Mariani, A. Mohebbi, M. Pezzè, and V. Terragni, "Semantic matching of gui events for test reuse: are we there yet?" in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 177–190.

[23] 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*, 2018.

[24] R. Nogueira and K. Cho, "Passage re-ranking with bert," *arXiv preprint arXiv:1901.04085*, 2019.

[25] W. Yang, Y. Xie, A. Lin, X. Li, L. Tan, K. Xiong, M. Li, and J. Lin, "End-to-end open-domain question answering with bertserini," *arXiv preprint arXiv:1902.01718*, 2019.

[26] N. Poerner and H. Schütze, "Multi-view domain adapted sentence embeddings for low-resource unsupervised duplicate question detection," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, pp. 1630–1641.

[27] 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 *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, 2017, pp. 845–854.

[28] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision*. Springer, 2016, pp. 21–37.

[29] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, and Gatford, "Okapi at trec-3," *Nist Special Publication Sp*, 1995.

[30] P. Yang, H. Fang, and J. Lin, "Anserini: Enabling the use of lucene for information retrieval research," in *ACM SIGIR*, 2017.

[31] "Pytorch," https://pytorch.org/.

[32] N. Reimers and I. Gurevych, "Sentence-bert: Sentence embeddings using siamese bert-networks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: http://arxiv.org/abs/1908.10084

[33] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, "Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter," *arXiv preprint arXiv:1910.01108*, 2019.

[34] "Appium," https://www.selenium.dev/.

[35] Y. Zhao, T. Su, Y. Liu, W. Zheng, X. Wu, R. Kavuluru, W. G. Halfond, and T. Yu, "Recdroid+: Automated end-to-end crash reproduction from bug reports for android apps," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–33, 2022.

[36] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to generate pseudo-code from source code using statistical machine translation (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 574–584.

[37] J. Wei and K. Zou, "EDA: Easy data augmentation techniques for boosting performance on text classification tasks," in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 6383–6389. [Online]. Available: https://www.aclweb.org/anthology/D19-1670

[38] "Monkey," https://developer.android.com/tools/help/monkey.html.

[39] "Selenium," https://www.selenium.dev/.

[40] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 11–20.

[41] S. Rastkar, G. C. Murphy, and G. Murray, "Automatic summarization of bug reports," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[42] D. Jin, M. B. Cohen, X. Qu, and B. Robinson, “Preffinder: Getting the right preference in configurable software systems,” in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 151–162.

[43] E. Wong, L. Zhang, S. Wang, T. Liu, and L. Tan, “Dase: Document-assisted symbolic execution for improving automated software testing,” in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1. IEEE, 2015, pp. 620–631.

[44] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, “Auto-completing bug reports for android applications,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 673–686.

[45] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, “Assessing the quality of the steps to reproduce in bug reports,” in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 86–96.

[46] S. Feng and C. Chen, “Gifdroid: automated replay of visual bug reports for android apps,” in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*. IEEE, 2022, pp. 1045–1057.

[47] D. Nurmuradov and R. Bryce, “Caret-hm: recording and replaying android user sessions with heat map generation using ui state clustering,” in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 400–403.

[48] J. Yan, H. Zhou, X. Deng, P. Wang, R. Yan, J. Yan, and J. Zhang, “Efficient testing of gui applications by event sequence reduction,” *Science of Computer Programming*, vol. 201, p. 102522, 2021.

[49] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, “Reran: Timing-and touch-sensitive record and replay for android,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 72–81.

[50] A. J. Ko and B. A. Myers, “Barista: An implementation framework for enabling new tools, interaction techniques and views in code editors,” in *Proceedings of the SIGCHI conference on Human Factors in computing systems*, 2006, pp. 387–396.

[51] J. Bell, N. Sarda, and G. Kaiser, “Chronicler: Lightweight recording to reproduce field failures,” in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 362–371.

[52] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, “Reproducing field failures for programs with complex grammar-based input,” in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 163–172.

[53] M. White, M. Linares-Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk, “Generating reproducible and replayable bug reports from android application crashes,” in *2015 IEEE 23rd International Conference on Program Comprehension*. IEEE, 2015, pp. 48–59.

[54] S. P. Reiss, Y. Miao, and Q. Xin, “Seeking the user interface,” *Automated Software Engineering*, vol. 25, no. 1, pp. 157–193, 2018.

[55] C. Chen, S. Feng, Z. Xing, L. Liu, S. Zhao, and J. Wang, “Gallery dc: Design search and knowledge discovery through auto-created gui component gallery,” *CSCW*, 2019.

[56] F. Behrang, S. P. Reiss, and A. Orso, “Guifetch: supporting app design and development through gui search,” in *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*. ACM, 2018.

[57] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, “Learning design semantics for mobile apps,” in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 569–579.

[58] X. Zhang, A. S. Ross, and J. Fogarty, “Robust annotation of mobile application interfaces in methods for accessibility repair and enhancement,” in *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, 2018, pp. 609–621.

[59] B. Wang, G. Li, X. Zhou, Z. Chen, T. Grossman, and Y. Li, “Screen2words: Automatic mobile ui summarization with multimodal learning,” in *The 34th Annual ACM Symposium on User Interface Software and Technology*, 2021, pp. 498–510.

[60] J. Chen, C. Chen, Z. Xing, X. Xu, L. Zhut, G. Li, and J. Wang, “Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning,” in *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 2020, pp. 322–334.

[61] Y. Li, G. Li, L. He, J. Zheng, H. Li, and Z. Guan, “Widget captioning: generating natural language description for mobile user interface elements,” *arXiv preprint arXiv:2010.04295*, 2020.

[62] F. Behrang and A. Orso, “Test migration between mobile apps with similar functionality,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 54–65.

[63] 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.