Mobile Bug Report Reproduction via Global Search on the App UI Model

ZHAOXU ZHANG, University of Southern California, USA
FAZLE MOHAMMED TAWSIF, University of Southern California, USA
KOMEI RYU, University of Southern California, USA
TINGTING YU, University of Connecticut, USA
WILLIAM G. J. HALFOND, University of Southern California, USA

Bug report reproduction is an important, but time-consuming task carried out during mobile app maintenance. To accelerate this task, current research has proposed automated reproduction techniques that rely on a guided dynamic exploration of the app to match bug report steps with UI events in a mobile app. However, these techniques struggle to find the correct match when the bug reports have missing or inaccurately described steps. To address these limitations, we propose a new bug report reproduction technique that uses an app's UI model to perform a global search across all possible matches between steps and UI actions and identify the most likely match while accounting for the possibility of missing or inaccurate steps. To do this, our approach redefines the bug report reproduction process as a Markov model and finds the best paths through the model using a dynamic programming based technique. We conducted an empirical evaluation on 72 real-world bug reports. Our approach achieved a 94% reproduction rate on the total bug reports and a 93% reproduction rate on bug reports with missing steps, significantly outperforming the state-of-the-art approaches. Our approach was also more effective in finding the matches from the steps to UI events than the state-of-the-art approaches.

CCS Concepts: • **Software and its engineering → Software maintenance tools**.

Additional Key Words and Phrases: Bug Report Reproduction

ACM Reference Format:
Zhaoxu Zhang, Fazle Mohammed Tawsif, Komei Ryu, Tingting Yu, and William G. J. Halfond. 2024. Mobile Bug Report Reproduction via Global Search on the App UI Model. *Proc. ACM Softw. Eng.* 1, FSE, Article 117 (July 2024), 21 pages. https://doi.org/10.1145/3660824

1 INTRODUCTION

Mobile applications have become extremely popular, with currently over 5 million apps in the Google Play and Apple stores. To remain competitive in this thriving market and ensure a high-quality user experience, app developers must swiftly address any issues reported by the users of their apps. When users experience a problem, they typically submit a bug report to describe the app failure and provide steps to reproduce the issue. After receiving such bug reports, developers need to first reproduce the failure following the provided steps [15] so that they can debug their app and fix the issue. However, the reproduction steps provided in these bug reports are often unclear, incomplete, or inaccurate [16, 31]. This can make it challenging for developers to reproduce

Authors' addresses: Zhaoxu Zhang, University of Southern California, Los Angeles, USA, zhaoxuzh@usc.edu; Fazle Mohammed Tawsif, University of Southern California, Los Angeles, USA, tawsif@usc.edu; Komei Ryu, University of Southern California, Los Angeles, USA, eryu@usc.edu; Tingting Yu, University of Connecticut, Storrs, USA, tingting.yu@uconn.edu; William G. J. Halfond, University of Southern California, Los Angeles, USA, halfond@usc.edu.

ACM 2994-970X/2024/7-ART117
https://doi.org/10.1145/3660824

the bug report manually. Even in cases where the provided steps are well-written, the process can still be time-consuming since modern apps generally have complex user interfaces (UIs) and support many similar sequences of UI events on different screens, each of which may or may not lead to the reported failure. As a result, developers may need to exhaustively explore the app to identify the correct events for reproduction. These challenges can make bug report reproduction a time-consuming and error-prone task for developers, potentially causing a significant delay in debugging the app.

To accelerate the reproduction process, the software engineering community has proposed automated techniques for reproducing bug reports of mobile applications. Given the reproduction steps in a bug report, these techniques dynamically explore the app and attempt to sequentially match each step against a UI event in the app. However, the low quality of bug reports can cause problems for automated reproduction. If a step is described inaccurately or vaguely, an automated approach may incorrectly match it with the wrong UI event. This can cause unintended UIs to appear and make it impossible to match subsequent steps. If the bug report has missing steps, a similar problem may occur. In this situation, the approach prematurely tries to match a step against the incorrect UI event, unaware that additional events are required to properly identify the event that matches the step accurately. These situations can lead to automated approaches needing more time to search for the correct reproducing events or cause them to fail completely if they are unable to account for the shortcomings of the bug reports.

Existing state-of-the-art approaches for bug report reproduction, ReCDroid [52], Yakusu [24] and ReproBot [50], attempt to address these challenges. However, these approaches have limitations that limit their overall effectiveness. To find a sequence of UI events that match with the given steps, ReCDroid and Yakusu employ a greedy search based strategy to navigate through an app's UIs. Inaccurate or missing steps can cause this strategy to incorrectly prioritize the exploration of paths with initially high similarity between steps and UI events, but that ultimately do not lead to an overall correct match. These approaches also incorporate heuristics in the search when they are unable to find a correct matching path, but in the worst case, their search may reduce to an exhaustive exploration of the app, which is very time-consuming. ReproBot improves this search process by utilizing a reinforcement learning (RL) based algorithm to guide the search. This allows the search to consider sequences of events that exhibit less similarity at the beginning but eventually lead to overall better matches. Nonetheless, the RL based approach cannot guarantee that it will find the best matching sequence since it is, by definition, non-deterministic. As we show in Section 4, the sub-optimal search strategies employed by these techniques can reduce their overall effectiveness in reproducing bug reports.

To address these limitations, we introduce a new approach for finding the UI events that best match the steps of a bug report. At a high level, our approach utilizes a model of the app's UIs to search for the global best-matched UI events to reproduce the steps. Since the search space in the UI model for this best match is large, our approach defines a Markov model to describe the possible ways to match steps with UI events and then uses a dynamic-programming based technique to efficiently find the best matches. In addition, our approach further uses the UI model to effectively bridge the possible missing steps in the bug report and discover the complete event paths for reproduction. We implemented our approach in a prototype tool, Roam, and compared its performance against the three aforementioned state-of-the-art approaches for bug report reproduction. The results of this evaluation showed that our approach achieved 94% reproduction rate on the subject bug reports, outperforming all three existing techniques. Our approach was particularly effective, as compared to the other approaches, when trying to reproduce bug reports with missing steps. Overall, these results were very positive and indicated that our approach to finding the best UI event matches for a bug report can lead to overall better bug report reproduction.

$\mathbf{s_x}$: (Click the test medicine.) **Missing Step**
$\mathbf{s_1}$: Edit a medicine.
$\mathbf{s_y}$: (Click Stock button.) **Missing Step**
$\mathbf{s_2}$: Click Add button.

(a) Reproduction Steps

$\mathbf{s_1}$ ={**action**: CLICK, **target**: Edit, ...}
$\mathbf{s_2}$ ={**action**: CLICK, **target**: Add button, ...}

(b) Entities of Reproduction Steps

Fig. 1. Example Bug Report

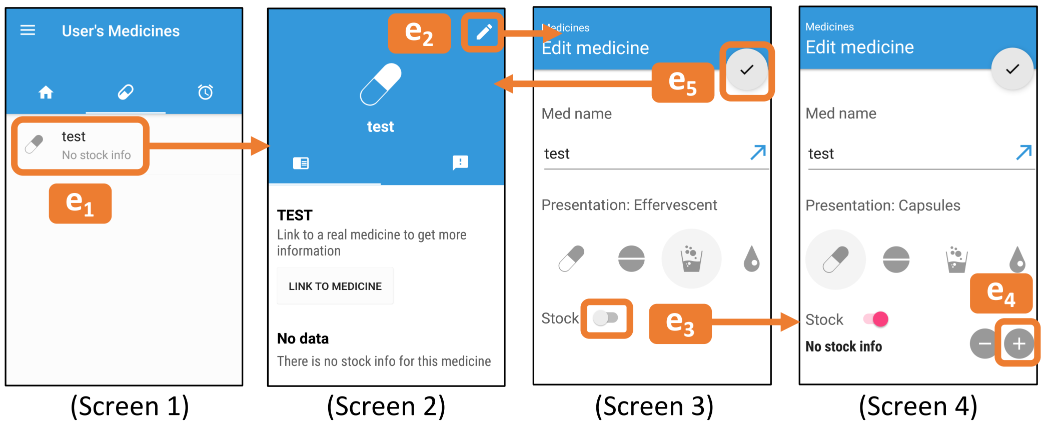

Fig. 2. The Actual UI Interactions to Replay the Steps

In summary, our paper makes the following contributions:

(1) We formulated the bug report reproduction process as a Markov model by simulating all possible matches between the steps and UI events.
(2) We designed a dynamic programming based algorithm to globally search for the most likely UI events to reproduce the steps.
(3) Based on the above two contributions, we developed a novel reproduction technique and implemented it into a research prototype.
(4) We conducted an empirical evaluation to assess the performance of our approach.
(5) Our implementation and evaluation dataset are available for future research [11].

2 MOTIVATION

In this section, we introduce an example that will be used throughout the paper to illustrate our approach and highlight the limitations of existing approaches. This example describes a bug that happened when a user attempted to increase the stock of medicine recorded in a personal medication management app, Calendula [1]. Figure 1a shows the four steps required to reproduce this bug. However, in the original bug report, only two steps were included (s_1 and s_2) while the other two were omitted (s_x and s_y). Figure 2 displays the necessary UI events for reproducing the steps in the app where each e_x denotes a click event performed on the app's UI. Specifically, to reproduce the four steps, one needs to follow the event path $e_1 \longrightarrow e_2 \longrightarrow e_3 \longrightarrow e_4$ shown in Figure 2.

[1]https://play.google.com/store/apps/details?id=es.usc.citius.servando.calendula&pli=1

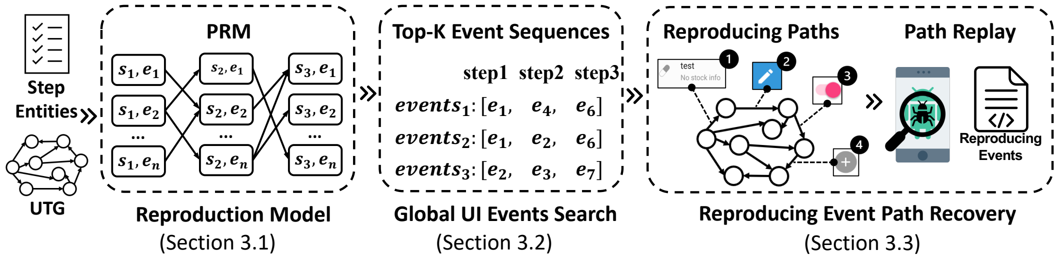

Fig. 3. Workflow of Our Approach

This example seems straightforward to reproduce. However, it may cause significant challenges to existing automated reproduction techniques.

To reproduce a given bug report, the existing approaches, ReCDroid [52], Yakusu [24], and ReproBot [50], employ two stages. In the first stage, they extract the entities for each step from the text using natural language processing (NLP) techniques. As illustrated in Figure 1b, these step entities capture the essential information regarding the UI action expressed by a step. They include the type of the UI action (*action*), the description of the action's target widget (*target*) etc. Then, in the second stage, these approaches dynamically explore the app and search for a path of UI events that match the steps.

Assuming the existing approaches can perfectly parse the step entities, their search algorithms still face limitations that hinder their reproduction performance. When searching for the UI events that match the steps, ReCDroid and Yakusu employ a greedy strategy, meaning that they will always prioritize to match a given step with the most similar UI event on the current UI screen. However, such a greedy strategy could lead them to choose an incorrect UI event when the next step is missing or inaccurately described. When reproducing the example bug report, both approaches would face problems finding the first event e_1 in Figure 2, as the corresponding step s_x is not included in the bug report. Consequently, both approaches need to exhaustively explore the UI trying to find the matched UI event on the following pages. Even if they could correctly find event e_1, they would face another challenge to correctly match step s_2 with event e_4. When reaching screen three in Figure 2, both approaches prematurely match step s_2 to an incorrect event e_5 (i.e., click "Add a medicine" button) since they are unaware of the missing step s_y. This eventually cause their failure in reproducing this bug report. The same problems also happen to ReproBot. To find the UI events matching the steps, ReproBot leverages a RL algorithm to guide the dynamic search in the app. Specifically, during the search process, the RL algorithm guides ReproBot to randomly evaluate matches between a step and a less-similar UI event, attempting to find a better-matched event on the following UIs. ReproBot also leverages such a non-deterministic mechanism to reveal the potential missing steps in the bug report. However, as one of the limitations discussed in ReproBot's paper [50], when reproducing the example bug report, ReproBot needs to try different possible matches, which is time-consuming and cannot guarantee finding the correctly matched UI events.

The limitations of existing works motivated us to design a better approach to improve the second stage of the reproduction process, i.e., the search for the UI events to reproduce the steps. We introduce the details of our approach in the next section.

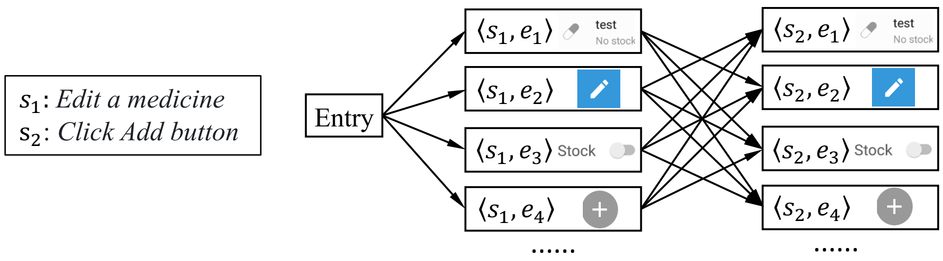
s_1: *Edit a medicine*
s_2: *Click Add button*
Entry
$\langle s_1, e_1 \rangle$ test No stock
$\langle s_1, e_2 \rangle$
$\langle s_1, e_3 \rangle$ Stock
$\langle s_1, e_4 \rangle$
......
$\langle s_2, e_1 \rangle$ test No stock
$\langle s_2, e_2 \rangle$
$\langle s_2, e_3 \rangle$ Stock
$\langle s_2, e_4 \rangle$
......

Fig. 4. The Probabilistic Reproduction Model for the Example in Figure 1a

3 APPROACH

The goal of our approach is to find a path of UI events in an app that will reproduce the steps leading to the failure described in a bug report. As discussed in Section 2, existing approaches are limited by their sub-optimal search strategies that may not find the best matching sequence when bug reports have inaccurate or missing steps. Our first insight is that the quality of the sequence matching can be improved by searching globally through the UI Transition Graph (UTG) of the app, a model of all of the UIs and their transitions (i.e., UI events) [18, 33, 49], for the best possible match of the bug report steps. This can avoid the limitations of existing approaches' dynamic search techniques, which can get trapped in local maxima best matches. However, such a global search is computationally expensive since the scale of modern apps and their UIs can be very large. Our second insight is that the problem of finding the best matching sequence of events can be solved efficiently by (1) modeling the matches between UI events and bug report steps as a Markov model, with the quality of the match represented by the model's transition probability function, and then (2) using dynamic programming techniques to solve for the best paths in the model, which would correspond to the best-matching UI event sequences. However, such an approach can only match UI events to steps that are not missing from the bug report. To address the problem of missing steps, our third insight is that the UTG can again be employed to find sub-paths of UI events that connect pairs of UI events that have been matched to bug report steps.

Our approach, which embodies these insights and mechanisms, is shown in Figure 3. The inputs of our approach are the step entities of the bug report and the existing UTG of the target app. In the first step, described in Section 3.1, our approach builds the Probabilistic Reproduction Model (PRM), the Markov model based representation of the possible matches between UI events and bug report steps. In the second step, described in Section 3.2, our approach performs a dynamic programming based computation to identify the best top-k matching sequences of UI events in the PRM. In the final step, described in Section 3.3, our approach traverses the UTG to bridge the potential missing steps in each of the top-k event sequences and identify the full reproduction event paths.

3.1 Probabilistic Reproduction Model

In this section, we introduce the definition of the Probabilistic Reproduction Model (PRM). The PRM is a Markov model that simulates all possible matches between the steps in a bug report and the UI events retrieved from the UTG of the target app. In general, the Markov model describes a stochastic process, in which the future states only depend on the current state [36]. Following the standard definition of a Markov model [36], we formally define the PRM as $\langle M, T, P \rangle$ where M is the states, T is the transitions between states, and P is the transition probability function that computes the likelihood of each transition. Section 3.1 gives an example of the PRM composed by the steps in Figure 1a and the UI events of target app in Figure 2. In the following paragraphs, we define each part of the PRM and use this example as an illustration.

The **states** $\boldsymbol{M}$ is the node set of the PRM. Each state m is a step-event pair, formally denoted as $m = \langle s, e \rangle$ where s is a step of the bug report and e is a UI event in the UTG. We use $m_0 = Entry$ to represent the initial state, meaning the start of the reproduction process. For a state $\langle s_i, e_j \rangle$, it could be intuitively interpreted as the approach choosing the UI event e_j to reproduce the step s_i during the reproduction. For example, in Section 3.1, the state $\langle s_1, e_1 \rangle$ means reproducing the first step "Edit a medicine" by clicking the "test" medicine button. In the PRM, a state is formed between a step and any UI event with the same type of UI action expressed by the step. For example, in Section 3.1, the step s_1 is paired with every possible click event e_1, e_2 etc. to form the possible states.

The **transitions** $\boldsymbol{T}$ is the set of directed edges in the PRM. We use $t_{m_s \rightarrow m_t}$ to represent the transition from a source state $m_s = \langle s_i, e_x \rangle$ to a target state $m_t = \langle s_j, e_y \rangle$. During the reproduction process, such a transition can be intuitively interpreted as, after reproducing step s_i using event e_x, the approach can subsequently reproduce step s_j using event e_y. For example, in Section 3.1, the transition from $\langle s_1, e_1 \rangle$ to $\langle s_2, e_2 \rangle$ means that, after reproducing the first step, the approach can reproduce the second step by clicking the "pencil" button. In the PRM, a transition between two states $\langle s_i, e_x \rangle$ and $\langle s_j, e_y \rangle$ exists if both of the following conditions hold:

C1. $\boldsymbol{j = i + 1}$ This condition means that the state of step s_i can only transit to the state of the next step s_{i+1}. The intuition of this condition is that each step in the bug report should be reproduced sequentially.

C2. $\boldsymbol{reachable(e_x, e_y)}$ This condition means that the event e_y should be reachable from the event e_x through some path in the UTG. The intuition of this condition is that since e_x and e_y match two successive steps in the bug report (as defined above in C1), after executing e_x, it must be possible to execute some sequence of UI events in the UTG that will allow the approach to find and execute (i.e., reach) event e_y.

The **transition probability function**, denoted as $P(t) \longrightarrow c$, shows the probability of a transition t occurring. Intuitively, for a transition from state $\langle s_i, e_x \rangle$ to state $\langle s_j, e_y \rangle$, its probability represents the likelihood that the approach will choose UI event e_y to reproduce step s_j, after reproducing step s_i with UI event e_x. To reflect such a likelihood during the reproduction process, we define this P as the sum of two scores: *Match Similarity Score* and *Transition Distance Score.*

The *match similarity score* measures the transition probability from the perspective of step-event similarity. The intuition is that a step is more likely to be reproduced by a UI event similar to the step description. Hence, when the step-event pair exhibits a higher similarity, the transition probability to such a state is higher. Our approach calculates the similarity between a step and a UI event by comparing the textual description of the widget that the UI event acts upon and the target entity of the step. To do this, our approach first extracts the textual description for the widget of the UI event. Specifically, three attributes of the widget are extracted from the UI as the description:

(1) *label*: This attribute is the text displayed on the widget;
(2) *content description (desc)*: This attribute is the text specified by developers to describe the widget, which is used by accessibility services (e.g., ScreenReader [4]) to explain the widget;
(3) *resource ID (res)*: This attribute is the file name of the linked source file of the icon.

The content of each of these attributes represents a meaningful explanation of the UI widget provided by developers during the app's development. Therefore, our approach uses them as the textual representation of the widget. Given a transition to state $\langle s_i, e_j \rangle$, the match similarity score is calculated as:

$$\text{match similarity score} = max_{w \in \{label, desc, res\}} \{sim(s_i.target, w)\} \tag{1}$$

In Equation (1), the function `sim` computes the semantic similarity between two text spans i.e., the target entity of the step ($s_i.target$) and one of the three attributes (w). Following the classic method, given two text spans, our approach computes their semantic similarity as the cosine distance of their word embeddings, the vector representation of a text, obtained from a pre-trained language model [19, 37, 38]. Then, our approach takes the maximum of the three similarities as the final similarity score. The rationale for choosing the maximum is that our approach cannot know which of these three description sources would be used by the bug reporter, so this mechanism allows for using the best or most informative fit.

The *transition distance score* measures the likelihood of a transition based on the distance of the UI events. The intuition is that when reproducing the steps on the target app, after executing the current step, the UI event to reproduce the next step is expected to be on a nearby screen. On the UTG, this intuition implies the distance between the two matched UI events is likely to be short. Based on this insight, for a transition between two states, $\langle s_i, e_x \rangle$ and $\langle s_j, e_y \rangle$, the transition distance score is computed as:

$$\text{transition distance score} = \frac{1}{shortestPathDistance(e_x, e_y)} \tag{2}$$

In Equation (2), the function `shortestPathDistance` computes the length of the shortest path between two events in the UTG.

3.2 Global UI Events Search

The goal of this stage of our approach is to search for the UI event sequences that globally best match the steps in the bug report. To address this problem, our insight is that these event sequences are most likely to be found in the top-k probable paths on the PRM. As explained in Section 3.1, the likelihood of a UI event matching a step is captured in the probability of each PRM transition. Therefore, by identifying the most probable paths from the PRM's entry state to the state of the last step, our approach can determine the sequences of events most likely to match the whole sequence of steps. Additionally, searching the top-k best paths helps our approach more effectively handle the potentially inaccurate steps and identify the correct event sequence.

Finding the top-k probable paths in the PRM can be formulated as a dynamic programming problem. The core idea of the dynamic programming algorithm [13] is to find an optimal solution to a complex problem by breaking it down into sub-problems and solving them recursively. In our problem, if we define the cumulative probability of the most probable path from the entry to the state m_i as $BestProb(m_i)$, the computation of it can be broken down into sub-problems of computing the $BestProb$ for the predecessor states of m_i. Mathematically, this is represented as:

$$BestProb(m_i) = \max_{m_j \in pred(m_i)} \{BestProb(m_j) + P(t_{m_j \to m_i})\} \tag{3}$$

where P is the transition probability function defined in Section 3.1 and function `pred` returns the predecessors of a state in the PRM. With this formulation, we designed an efficient dynamic programming algorithm to find the top-k most likely UI event sequences.

Algorithm 1 provides the details of our algorithm. The inputs of the algorithm are the PRM (constructed in Section 3.1) and a number k. At a high level, our approach first conducts a forward computation pass (Lines 3 to 8) by traversing the PRM states from the state of the first reproduction step to the state of the last reproduction step. During the forward pass, our approach determines the best cumulative probabilities achieved by PRM paths reaching each state. Next, our approach

Algorithm 1: Top-K UI Event Sequences Computation

```
Input: PRM, $k$
Output: top-k event sequences $E_{key}$
1  $best \leftarrow \emptyset$ ; /* table storing the cumulative probability of the optimal path */
2  $back \leftarrow \emptyset$ ;                              /* table storing the back pointers */
   /* Forward computation pass */
3  $best[\text{PRM}.entry] \leftarrow 0$;
4  for PRM state m from the first step to the last step do
5      $Preds \leftarrow$ the predecessors of $m$;
6      $best[m] \leftarrow \max_{m' \in Preds}\{best[p] + \text{PRM}.P(t_{m' \rightarrow m})\}$;
7      $back[m] \leftarrow \text{argmaxK}_{m' \in Preds}\{best[p] + \text{PRM}.P(t_{m' \rightarrow m})\}$;
8  end
   /* Backward decoding pass */
9  $eventSeqs \leftarrow \emptyset$ ;                  /* list storing the top-k event sequences */
10 $candidates \leftarrow$ a max heap with size k;
11 for PRM state m of the last step do
12     $candidates$.insert($[m]$, key=$best[m]$);
13 end
14 while $candidates.length > 0$ do
15     $c \leftarrow candidates$.pop();
16     $m \leftarrow c.lastElement()$;
17     if $m == PRM.entry$ then
           /* Decoding reaches the entry state */
18         $eventSeqs$.append($c$.getMatchedEvents());
19     end
20     for predecessor state $p$ in $back[m]$ do
21         $new\_candidate \leftarrow c.append(p)$;
22         $new\_score \leftarrow best[p]$ + cumlative_prob($new\_candidate$);
23         $candidates$.insert($new\_candidate$, key=$new\_score$);
24     end
25 end
26 return $eventSeqs$
```

conducts a backward decoding pass (Lines 9 to 25) to reveal the PRM paths with the top-k cumulative probabilities.

In order to identify the top-k paths in the PRM, for each state, our approach keeps track of its predecessors that are on the paths achieving the top-k cumulative probabilities during the forward pass (Line 7). This information is later utilized in the backward decoding pass to expand the path candidates. In the backward decoding pass, our approach uses a k-sized max heap to store the top-k probable path candidates. The path candidates are initialized with the states that achieved the top-k cumulative probabilities (Lines 10 to 13). In each round of decoding, our algorithm pops off the candidate path with the highest score from the heap, expands it with its back pointers computed from the forward pass, and reinserts the updated candidate into the heap with its new total score (Lines 20 to 24). The new total score is computed as the sum of $best[p]$ — the best cumulative probability that can be achieved at the expanded state, and $cumulative\_prob(new\_candidate)$ — the current cumulative probability of the candidate path (Line 22). In this way, the new score represents the highest potential cumulative probability that can be achieved by the new path candidate. By doing this, the heap always keeps the overall top-k probable path candidates. The decoding of a path finishes when it reaches the entry state of the PRM (Line 18).

The outputs of this stage of our approach are the UI event sequences that are mostly likely to match the steps provided in the bug report. For the example in Section 3.1, our approach identifies the event sequence $[e_2, e_4]$ as one of the output sequences that correctly matches the two given steps. All such event sequences will be used in the next stage of our approach to determine the full reproduction path.

3.3 Reproduction Event Path Recovery

The goal of this stage of our approach is to identify the UI events that reproduce the steps missing from the bug report, thereby finding the complete reproduction event path. As discussed in related work [31, 48], bug reporters often omit or overlook certain reproduction steps when writing a bug report. These missing steps could happen (1) before the first provided step; (2) in between any two provided steps; or (3) after the last provided step in a bug report. When any of these three situations occur, the UI events identified in Section 3.2, which only match the provided steps, will be insufficient to reproduce the bug. To handle these types of missing steps and discover the full reproduction path, our insight is that the set of paths originating from the initial UI state and leading up to the first matched event, connecting two consecutive events, or extending from the final matched event could be the ones omitted by bug reporters.

Given a sequence of events identified in the prior stage (Section 3.2), our approach first explores the UTG to handle the first two types of missing steps defined above. To handle the first type of missing steps, our approach locates the event paths from the beginning state of the UTG (i.e., the initial UI of the target app) to the UI states containing the first identified event (i.e., the one reproduces the first given step). Then, our approach follows a similar idea to handle the second type of missing steps. Specifically, for each adjacent pair of identified UI events, our approach searches for the paths that connect their corresponding UI states on the UTG. For both cases, our approach identifies the shortest paths [21] between two UI states, which helps our approach to find the most efficient way of reproduction and avoid redundant events [25]. At the end, for each UI event in the sequence, our approach obtains a set of paths that connect it either from the beginning UI state or the previous event. These paths will be joined together to get a set of full paths.

Next, our approach extends each identified path in the UTG to handle the third type of missing steps, i.e., steps omitted after the last provided step. Given a path identified previously, our approach computes a set of paths using a fixed-depth exploration starting from the last UI event of the matched path. Since the number of missing steps after the last provided step tends to be small [31], a fixed-depth exploration is a sufficient and efficient method to address this problem. Note that this procedure may create extra events for reproduction when the bug report does not have the second type of missing steps. In this case, our approach will eliminate the extra ones after triggering the failure when executing the event path in the target app.

4 EVALUATION

In this section, we evaluated our approach by answering the following research questions:

RQ1. How effectively can our approach identify the UI events matching the reproduction steps?
RQ2. How effectively can our approach reproduce the bug reports?
RQ3. What is the running time of our approach?

Table 1. Bug Report Step Information

	Average	**Median**	**Minimum**	**Maximum**
Provided Steps	2.9	3	1	7
Total Steps	4.3	3	1	27
Missing Steps	2.4	1	1	21

RQ4. How effectively can our approach reproduce bug reports with missing steps?
RQ5. How much does each sub-score in the transition probability function help the reproduction?

4.1 Implementation

For evaluation purposes, we implemented our approach into a research prototype, Roam (ReprOduction using App Model). Roam is implemented using Python. In its implementation, we used Neo4J [8] to store and access the UTG, UIAutomator [10] to interact with the Android device, and Spacy [9] to compute the semantic similarity of texts. Our experiments were performed on Android emulators on a physical x86 Ubuntu 20.04 machine with a 3.6GHz 8-core CPU and 32GB of memory. We set the default value for k to be 100, meaning that during the reproduction, our approach computes the top one hundred UI event sequences that match the steps in a bug report. We set two as the default depth for the fixed-depth exploration (as explained in Section 3.3). To facilitate future research, our implementation is available on the website [11].

4.2 Dataset Collection

We used a dataset of 72 real-world Android bug reports for our experiments. The subjects came from the artifacts of five open-source datasets: (1) the evaluation dataset of ReproBot [50]; (2) the evaluation dataset of ReCDroid [51, 52]; (3) the evaluation dataset of Yakusu [24]; (4) an empirical study on Android bug report reproduction [31]; and (5) an Android bug report dataset [48]. Altogether, the open-source datasets provided an initial set of 399 bug reports. For each bug report in these open-source datasets, we attempted to manually reproduce it to make sure the steps were reproducible and duplicates were removed. During this process, we found many bug reports were no longer reproducible due to the following factors: (1) The bug report link had expired; (2) The corresponding APK file of the app was no longer available; and (3) Necessary setup information such as an account or a specific environmental configuration was not provided. We ended up with a dataset of 72 bug reports, 46 (64%) of which had missing steps. Table 1 displays the statistics of the reproduction steps for the bug reports in our dataset. The "Provided Steps" row indicates the number of steps written in the bug report. The "Total Steps" row shows the total number of steps actually required to reproduce the bug report. The "Missing Steps" row shows the number of steps missing from the bug reports for the subset of reports that have missing steps.

For each bug report, we constructed the inputs for the approach: step entities and UTG. We followed the established protocol used by ReproBot to construct the step entities from the original bug reports. For subjects that already have the step entities created (i.e., the subjects from ReproBot's evaluation dataset), we directly reused them. To build the UTG of each target app, we followed the standard practice of running a dynamic app crawler to explore the target app and constructed the UTG using the crawling results [25]. We leveraged an open-source crawler, AppCrawler [1] for this purpose.

Table 2. The Match Accuracy Results on Bug Reports (Shown as a Percent)

	Match Accuracy	**Perfect Cases**	**Zero Cases**
PRM-Enumeration	61	56	33
ReproBot	69	47	10
Euler	72	56	14
Roam	**93**	**85**	**0**

4.3 RQ1. Identifying UI Events Matching the Reproduction Steps

4.3.1 Protocol. The goal of this RQ is to assess how accurately our global search mechanism (as defined in Section 3.2) can identify the matching between the steps and UI events.

For this RQ, we compared the performance of Roam with three baselines: (1) ReproBot [50]: It is a state-of-the-art mobile bug report reproduction technique. ReproBot uses a Q-learning algorithm to dynamically search for the UI events matching the given bug report steps on the target app; (2) Euler [16]: It is a state-of-the-art approach for assessing the quality of mobile bug report steps. Euler proposed an n-step look-ahead algorithm on the UTG to find the UI event matching a given reproduction step. At a high level, Euler matches a given step to the most similar UI event on the current UI or UIs within n steps on the UTG. Though Euler is not designed to reproduce the bug report, its algorithm is effective in finding the corresponding UI event of the steps on the UTG. Therefore, we included Euler for comparison in this RQ. In our experiment, we adopted the value $n = 6$ based on Euler's evaluation, which determined it as the best value for finding matching events; and (3) PRM-Enumeration: It is a baseline approach we implemented to show the impact of the dynamic programming algorithm (described in Section 3.2). Specifically, to find the top-K event sequences on the PRM, instead of using the dynamic programming based approach, this baseline enumerates all possible paths on the PRM and ranks them based on the cumulative probability of the path. Note that we did not include Yakusu [24] and ReCDroid [52] in this RQ because they do not provide a one-to-one mapping from steps to UI events, which is necessary to measure their performance for this RQ.

To conduct the experiments, we ran all approaches on the dataset with one hour as the time limit per run. To mitigate the impact of the non-determinism of the Q-learning algorithm [50], we ran ReproBot ten times on each subject. We manually checked the correctness of the UI events found by each approach. For a given bug report, we computed the *match accuracy* as the ratio of correctly identified UI events to the total number of identified UI events.

4.3.2 Presentation of Results. The results of RQ1 are shown in Table 2. For each approach, the "Match Accuracy" column represents its average match accuracy on the dataset. The "Perfect Cases" column represents the percentage of bug reports where the approach's match accuracy is one hundred, indicating that the approach found the correct UI events matching all steps in the bug report. The "Zero Cases" column represents the percentage of bug reports where the approach's match accuracy is zero, indicating that the approach failed to find the correct UI event matching any step in the bug report. The detailed match accuracy of each subject for each tool is included in the replication package [11].

4.3.3 Discussion of Results. According to the results, Roam achieved an average match accuracy of 93%, which is 21% higher than Euler, 24% higher than ReproBot, and 32% higher than PRM-Enumeration. Notably, on 85% of bug reports, Roam was able to correctly find reproducing UI events for all steps. This ratio is 29% higher than PRM-Enumeration and Euler, and 38% higher than

ReproBot. Furthermore, Roam was able to identify the correct UI events for reproducing at least one step in all bug reports. In contrast, other methods were unsuccessful in identifying UI events for reproducing any step in 10% to 33% of the bug reports. Taken together, Roam outperformed the other baseline approaches in identifying the UI events matching the reproduction steps.

We manually investigated the results of all approaches and found that Roam outperformed the other approaches primarily for two reasons. First, by searching for the UI events based on their overall matchability with the steps, Roam avoided being misled by the inaccuracies from a single step. As explained in Section 3.2, Roam searches for the overall best-matched UI events on the PRM. This helped it to accurately find the events that match the whole sequence of steps even if some were inaccurate. On the contrary, Euler was less effective in handling inaccurate steps. This was because it relied solely on each step to search for the corresponding UI event. In situations where the step description was inaccurate, resulting in a higher similarity with an incorrect UI event, Euler was misled into selecting the wrong event on the UTG. The same thing happened to ReproBot. These inaccuracies in the steps led to ReproBot's dynamic search being misled, preventing it from finding the correct events. Second, when searching for UI events on the PRM, the dynamic programming algorithm used by Roam was more effective than the enumeration. We noticed that, due to the vast number of event sequences to be enumerated, PRM-Enumeration failed to complete the search within the time limit for 32% of the bug reports.

Even though Roam had better average match accuracy than the baselines, it still couldn't find the right match for some bug report steps. We investigated these cases manually and identified two main causes. First, in some cases, Roam failed to compute the similarity score between a given step and a UI event, resulting in finding an incorrect UI event. As explained in Section 3.1, Roam requires some textual information from the widget in the UI in order to compute the match similarity score. However, we observed instances where app developers did not provide this information in the UI. Consequently, Roam could not calculate a reasonable similarity score, leading to the failure to find the correct UI event. Second, Roam failed to find the correct UI event matching amalgamated steps, i.e., when reporters combine multiple UI interactions into a single step. For instance, to reproduce the step "uncheck all checkboxes" in bug report collect#2958 [3], it is required to click on each checkbox in a list individually. Since Roam was designed to identify a one-to-one mapping from steps to UI events, it could not correctly identify such a one-to-many relation.

4.4 RQ2. Reproducing Bug Reports

4.4.1 Protocol. The goal of this RQ is to evaluate how effective our approach is in reproducing the bug reports. In this RQ, we compared the performance of Roam with three state-of-the-art reproduction techniques: ReproBot [50], ReCDroid [52], and Yakusu [24]. Note that all three approaches can compute the step entities directly from the bug report text. However, to ensure a fair comparison in this experiment, we provided the same step entities collected in Section 4.2 to all three approaches and Roam. To conduct the experiment, we ran all approaches to reproduce the bug reports in our dataset, with one hour as the time limit for each bug report. We ran ReproBot ten times to mitigate the impact of its non-determinism on the reproduction results. For each successful reproduction reported by a tool, we manually replayed the generated reproduction path to ensure it followed the same sequence described as the bug report steps. We found three false successful reproductions for ReCDroid, four false reproductions on average for ReproBot, and none for the other approaches. We used the *reproduction rate* as the metric, which was computed as the number of successful reproduction divided by the total number of reproduction an approach had run.

4.4.2 Presentation & Discussion of Results. Table 3 displays the average reproduction rate of each approach on the whole dataset ("All Reports"), the subset of bug reports that do not have missing

Table 3. Reproduction Rate of Each Approach (Shown as a Percent)

	All Reports	**Reports w/o Missing Steps**	**Reports w/ Missing Steps**
ReproBot	70	83	63
ReCDroid	29	27	30
Yakusu	8	15	4
Roam	**94**	**96**	**93**

steps ("Reports w/o Missing Step"), and the subset of bug reports that have missing steps ("Reports w/ Missing Steps"). The detailed reproduction results of each subject for each tool are included in the replication package [11]. As shown in the results, Roam achieved the highest reproduction rate on the whole dataset, also on the subset of bug reports that have or do not have missing steps. Roam outperformed all state-of-the-art approaches on reproducing bug reports.

We further analyzed the bug reports that Roam was unable to reproduce and found two main reasons for the failure. The first reason was that Roam inaccurately computed the UI events matching the provided steps (as discussed in RQ1). As explained in Section 3.3, Roam recovers the final reproducing path based on the identified UI events. Therefore, if the events were incorrect, Roam produced an incorrect path and failed to reproduce the bug report. The second reason was that Roam failed to discover the missing steps in certain cases. For example, in the bug report phimpme-android#1858 [5], the reproduction steps are "click on the edit option then abort the edit operation". To reproduce this bug report, it required the approach to first click the "edit" button, change some configurations in the Edit page, and then click the "abort" button. However, Roam, without modifying any configurations, clicked the "abort" button immediately after clicking the "edit" button, resulting in a reproduction failure. As explained in Section 3.3, Roam determines whether there is any missing step between two given steps by checking if their corresponding UI events are discontinuous on the UTG. For this bug report, this caused Roam to mistakenly determine that there was no missing step between these two steps. We found that the missing steps in this bug report were not easily noticeable even for humans. When manually reproducing this bug report, the authors initially identified the same events as Roam and it took several tries to eventually find the missing steps.

We observed that in some cases, ReproBot successfully reproduced the bug report but failed to identify the correct matching relation between the steps and UI events, which caused ReproBot to achieve a high reproduction rate but a comparatively lower score on Perfect Cases in RQ1. After further exploration, we found this was because of the random mechanism employed by ReproBot to handle inaccurate or missing steps (as discussed in Section 2). In some cases, this mechanism allowed ReproBot to trigger the bug successfully without identifying the correct match for all steps. For instance, when reproducing NewsBlur#1053 [2], ReproBot inserted a random UI event before matching the final step, "Click register". The purpose of this random event was to account for any potential missing steps. However, the random event was to click the "Register" button (i.e., the one to reproduce the final step), which caused ReproBot to trigger the bug directly without actually matching the final step. As a result, ReproBot successfully reproduced the bug but failed to find the correct matching for the last provided step in this bug report.

4.5 RQ3: Reproduction Running Time

4.5.1 Protocol. The goal of this RQ is to assess the running time of our approach on reproduction. To answer it, we measured the running time of each approach spent on each reproduction. In the

Table 4. The Running Time Results of Each Approach (Shown in Seconds)

	Roam				ReproBot	ReCDroid	Yakusu
	PRM	Event Search	Path Recv & Exec	Total			
Avg.	19	16	470	**505**	1,128	2,674	3,354
Mdn.	1	1	111	**150**	229	3,600	3,600

case where the approach failed to reproduce a bug report, its running time was recorded as the time limit for a reproduction, i.e., one hour.

4.5.2 Presentation & Discussion of Results. The results for all of the tools' running times are displayed in Table 4, with each cell representing the average ("Avg.") and median ("Mdn.") running time per run. For Roam, we also provide the running time for each stage of our approach. Specifically, the column "PRM" represents the running time for constructing the PRM (as explained in Section 3.1); the column "Event Search" represents the running time for identifying the top-k event sequences on PRM (as explained in Section 3.2); and the column "Path Recv & Exec" represents the running time spent on identifying the complete reproduction event paths and executing them in the app (as explained in Section 3.3). The detailed running time of each subject for each tool is included in the replication package [11]. Based on the results, Roam had the shortest running time compared to other tools, with an average of 505 seconds and a median of 150 seconds.

We further evaluated the running time when considering the time for constructing the UTG. After adding the UTG construction time, the average running time of Roam becomes 2,655s, ranking the second best among all baseline approaches. However, it is worth noting that building the UTG is a one-time effort and can be conducted in advance, after which all the bug reports of the same app can reuse the model. This means that the time spent on building the UTG can be distributed across all bug reports for an application. An example of this situation is the subject app *phimpme-android*, which has seven bug reports in our dataset. When amortizing the UTG construction cost, the average running time of Roam is 1,645s versus 2,418s for ReproBot. ReCDroid and Yakusu were unable to reproduce it, resulting in a time of 3,600s.

4.6 RQ4: Reproducing Bug Reports with Missing Steps

4.6.1 Protocol. The goal of this RQ is to assess the reproduction performance of our approach on bug reports that have missing steps. To answer this RQ, we ran Roam and the three state-of-the-art reproduction techniques: ReproBot [50], ReCDroid [53] and Yakusu [24] on the 46 bug reports that contain missing steps following the same protocol as in RQ2. We used the *reproduction rate* introduced in RQ2 as the metric for this experiment.

4.6.2 Presentation of Results. Figure 5 presents the reproduction performance of each approach on the bug reports with missing steps. Each line in the chart shows the reproduction rate for an approach with respect to the minimum number of missing steps in a bug report. For example, when the value of the x-axis equals one, the value of the y-axis represents the reproduction rate of each approach on the bug reports with at least one missing step.

4.6.3 Discussion of Results. As shown in Figure 5, Roam achieved a 93% reproduction rate on bug reports with at least one missing step. This number is 30% higher than ReproBot's, 63% higher than ReCDroid's, and 89% higher than Yakusu's. In addition, despite the increasing number of missing steps in a bug report, Roam consistently outperformed the state-of-the-art approaches. In particular, on bug reports with at least three missing steps, Roam's reproduction rate was 80%,

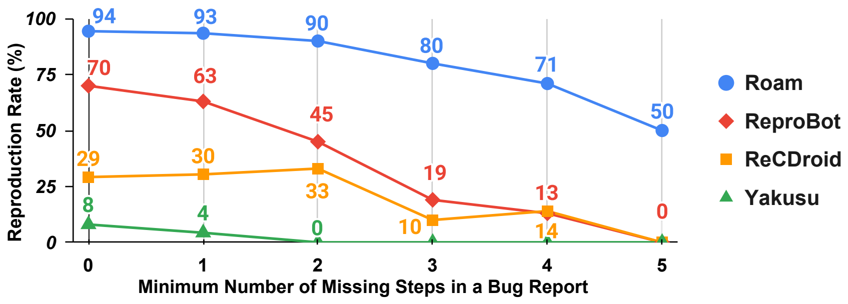

Fig. 5. Reproduction Rate of Each Approach on Bug Reports with Missing Steps

only decreasing by 15% compared to its performance on all bug reports. In contrast, the other approaches experienced much larger drops in their reproduction rates, with ReproBot dropping by 73%, ReCDroid dropping by 66%, and Yakusu dropping by 100%, compared to their performance on the entire dataset. In addition, for bug reports with five or more missing steps, none of the state-of-the-art reproduction techniques was able to reproduce them while Roam still had a 50% reproduction rate.

We investigated why Roam outperformed other approaches in handling missing steps and found two reasons. First, the PRM assisted Roam in accurately identifying the UI events matching the provided steps even if there were missing steps among them. As explained in Section 3.1, the PRM considers all matches between the given steps and UI events on the app, accounting for the possibility of missing steps. Therefore, with the PRM, Roam was able to accurately locate the UI events to reproduce the provided steps even if they were incomplete. This provided an accurate foundation for Roam to further discover the reproduction path. Second, the UTG helped Roam bridge the missing steps more effectively. As explained in Section 3.3, after identifying the UI events matching the provided steps, Roam used the UTG of the target app to discover potential gaps caused by missing steps. In this process, the UTG provided a clear guidance on what UI events could connect two provided steps, helping Roam to discover the missing steps more effectively. In comparison, existing approaches relied on random exploration or exhaustive search to uncover the missing steps while dynamically exploring the app. These techniques were less efficient and less effective compared with Roam's.

4.7 RQ5: Contribution of Sub-Scores in the Transition Probability Function

4.7.1 Protocol. The goal of this RQ is to evaluate the contribution of each sub-score defined in the transition probability function of the PRM (as introduced in Section 3.1) to the overall performance of our approach. To answer this RQ, we created two variants of Roam: Roam-Sim and Roam-Dist. Roam-Sim only used the *match similarity score* as the transition probability when building the PRM while Roam-Dist only used the *transition distance score*. We ran both variants and Roam on the whole dataset and measured the *match accuracy* (as introduced in RQ1) and the *reproduction rate* (as introduced in RQ2).

4.7.2 Presentation & Discussion of Results. Table 5 presents the average match accuracy and reproduction rate for each approach. The detailed results for two variants on each subject are included in the replication package [11]. As shown in Table 5, both variants performed worse than the original Roam in identifying the UI events and reproducing the bug reports. Compared with Roam, Roam-Sim's average match accuracy was 5% lower, and Roam-Dist's was 16% lower. In addition, Roam-Sim's reproduction rate was 29% lower, and Roam-Dist's was 56% lower than Roam.

Table 5. The Match Accuracy and Reproduction Rate for RQ5

	Avg. Match Accuracy (%)	Reproduction Rate (%)
Roam-Sim	88	65
Roam-Dist	77	38
Roam	**93**	**94**

The performance difference between Roam and the two variants demonstrated the contribution of both sub-scores to the overall performance of Roam.

We further analyzed the causes behind the performance decline for both variants. For Roam-Sim, we found it less effective in finding the correct UI event sequence when there were multiple UI events similar to the same step. Roam-Sim searched for the UI events purely based on their similarity with the steps. Therefore, if there were multiple UI events similar to the same step, Roam-Sim considered all of them possible to reproduce the step. As a result, the correct event sequence was ranked lower. Ultimately, Roam-Sim either failed to find the correct events because of the low ranking or did not have enough time to execute them, causing a failure of reproduction. For Roam-Dist, without considering the similarity with the reproduction steps, its reproduction process was reduced to an exhaustive search through all event paths in the app. In most cases, such an exploration was extremely time-consuming and unable to reproduce the bug within the time limit.

4.8 Threats to Validity

4.8.1 External Validity. A potential threat to the external validity of our results is the representativeness of the selected bug reports. We have attempted to overcome this threat by searching for subjects from the artifacts of five open-source datasets. It is worth noting that all the subjects are real-world Android bug reports either from GitHub Issues [6] or Google Code [7]. Another potential threat is the representativeness of the UTG used in our experiments. In the evaluation, we tried our best to create the UTG as completely as possible using a dynamic crawler. However, there could be a potential threat that these UTG may not be representative of the UTG used by real-world developers in terms of completeness. With a more complete UTG, there might be additional event paths that introduce noise into the search and prevent our approach from finding the correct event path to reproduce the bug report. To better understand the potential implications of this threat, we conducted a small-scale robustness study on the UTG. Specifically, we randomly selected ten subjects that our approach successfully reproduced and manually enriched their UTGs to make them as complete as possible. Then, we reran our approach using the new UTGs and found that the reproduction results remained consistent with the original results.

4.8.2 Internal Validity. A potential threat to internal validity is the biases that could be introduced when identifying the step entities. To mitigate this threat, the first two authors independently conducted the same step entity identification process following the established practice from the existing study [50]. Then, they compared their results and resolved the discrepancies via discussion. Another threat is the potential biases that could influence the manual validation of the UI events in Section 4.3. To address this threat, in the manual validation process, we referred to the reproduction UI events information in the original dataset if provided. This information was collected by the authors of the original dataset, which could help reduce bias.

5 RELATED WORK

In this section, we review the related works from three topics: Mobile Bug Report Reproduction, Mobile Bug Reporting and Management, and UI Transition Graph for Mobile App Analysis.

Mobile Bug Report Reproduction There have been several works focusing on textual mobile bug report reproduction. ReCDroid [52] and Yakusu [24] are the first two works in this direction. Both approaches employ a dynamic and greedy search on the app to find the UI events to reproduce the steps. Recently, Zhang et al. proposed ReproBot [50], which employs a reinforcement learning algorithm to guide the dynamic search for the UI events. Different from these existing works, our approach leverages the UTG of the target app to globally search for the UI event path to reproduce the bug report steps. Our approach first constructs a Markov Model to represent all possible matches between the steps and UI events in the UTG, then uses a dynamic programming algorithm to find the top-k most likely event sequences for reproducing the steps. Our approach further leverages the UTG to discover the potential missing steps in the bug report and identify the full reproduction event paths. After a comprehensive comparison with these existing works, our evaluation results show that our approach performs better in finding the correctly matched events and reproducing the bug report than the existing techniques.

Many research works focused on improving a specific part of the automatic reproduction process. Liu et al. developed a machine learning model Maca [34] to classify the UI action type of a reproduction step. Huang et al. introduced a multi-modal neural network ScopeDroid [30] to compute the similarity between a UI event and a reproduction step. Different from both works, our approach focuses on improving the search of the reproducing UI events during the reproduction process. Both two works are complementary to our approach and provide future directions for better reproduction techniques.

There are also research efforts on replaying video mobile bug reports. Feng et al. introduced GIFdroid [25], an automatic tool to replay screen video recordings on Android apps. GIFdroid uses image-processing techniques to identify keyframes from the video and map them to the actual app's UI. Bernal-Cárdenas et al. introduced V2S, another method to replay app video recordings. V2S employs a deep neural network to identify the UI actions from the videos and replay them on apps. Unlike these two works, our approach focuses on reproducing textual steps from Android bug reports.

Mobile Bug Reporting and Management A great body of work has been dedicated to assisting with bug reporting. Fusion, developed by Moran et al. [40], leverages dynamic analysis to obtain UI events of the app under test (AUT) and help create more actionable events in bug reports during the testing stage. Fazzini et al. proposed EBUG [23] to assist reporters in writing more accurate reproduction steps by using information from static and dynamic analysis of the AUT to predict the next step. Demibud [17] introduced by Chaparro et al. uses manually built discourse patterns to check whether a bug report misses descriptions for *observed behavior*, *expected behavior*, or *steps to reproduce*. To help users report these elements, Yang et al. proposed an interactive bug reporting system, BURT [44] to provide instant feedback of problems with the elements and graphical suggestions of the said elements. Euler developed by Chaparro [16] assesses the quality of documented reproduction steps by mapping them onto the UTG of the app. There are also many works that empirically study the bug-reporting process [22, 42, 43, 54, 55] and detect duplicate bug reports [20, 28, 29, 32, 41, 46, 47]. Different from these works, our work focuses on reproducing the steps documented in the bug reports.

UI Transition Graph for Mobile App Analysis The UTG has been widely used in research works on mobile app review [18], UI testing [12, 26, 27, 35, 45] and bug reporting [14, 23, 25, 39]. StoryDroid developed by Chen et al. [18] utilizes the UTG of apps in the market to measure their

UI similarity. This helps developers to easily review similar apps in the market and avoid conflicts of functionalities in their app design. Stoat proposed by Su et al. [45] uses the UTG to automatically generate UI test cases for mobile app in order to achieve a higher code coverage and UI coverage. NaviDroid developed by Liu et al. [35] leverages the UTG of an app to guide human testers more effectively explore the app by providing hint moves on the UI. The UTG helps it to effectively plan the exploration paths to previously unvisited UI pages. Different from these works, our work uses the UTG to globally search for the UI events on the app to reproduce the bug report steps.

6 CONCLUSION

In this work, we propose a new approach to effectively reproduce mobile bug reports by globally searching the UI events in the target app's UI Transition Graph. Our approach leverages a Markov model to simulate all possible ways to reproduce the steps using all UI events supported by the app. From this model, our approach employs a dynamic programming algorithm to compute the top-k most likely event sequences to reproduce the given steps. Then it further combines the path information from the UTG to identify the plausible missing steps in order to find the full reproducing paths. We conducted an empirical evaluation of our approach and compared its performance with the state-of-the-art approaches. As shown by the results, our approach is more effective in reproducing the bug reports and handling the missing steps in a bug report than the state-of-the-art techniques. It can also more accurately identify the UI events that match with the bug report steps.

7 DATA-AVAILABILITY STATEMENT

In a replication package [11], we provide the source code and documentation of Roam, the dataset we used for the evaluation, and the detailed evaluation results.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under Grant No. 2211454.

REFERENCES

[1] 2015. App Crawler. https://github.com/Eaway/AppCrawler.
[2] 2017. IllegalArgumentException: unexpected url. https://github.com/samuelclay/NewsBlur/issues/1053.
[3] 2019. Accessing General settings form Admin settings does not display disabled submenus. https://github.com/getodk/collect/issues/2958.
[4] 2023. Android Accessibility. https://support.google.com/accessibility/android/answer/7158690?hl=en.
[5] 2023. Favourite photos edit option issue. https://github.com/fossasia/phimpme-android/issues/1858.
[6] 2023. Github Issue Tracker. https://github.com/issues.
[7] 2023. Google Code Issue Tracker. https://code.google.com/archive/.
[8] 2023. Neo4j Graph Database and Analytics. https://neo4j.com/.
[9] 2023. Spacy: Industrial-Strength Natural Language Processing. https://spacy.io/models.
[10] 2023. UI Automator. https://developer.android.com/training/testing/ui-automator.
[11] 2024. Replication Package. https://doi.org/10.5281/ZENODO.11068809
[12] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313
[13] Richard Bellman. 2010. *Dynamic Programming*. Princeton University Press, USA.
[14] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM, Seoul South Korea, 309–321. https://doi.org/10.1145/3377811.3380328

[15] Pamela Bhattacharya, Liudmila Ulanova, Iulian Neamtiu, and Sai Charan Koduru. 2013. An Empirical Analysis of Bug Reports and Bug Fixing in Open Source Android Apps. In *2013 17th European Conference on Software Maintenance and Reengineering*. 133–143. https://doi.org/10.1109/CSMR.2013.23 ISSN: 1534-5351.

[16] Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng. 2019. 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*. ACM, Tallinn Estonia, 86–96. https://doi.org/10.1145/3338906.3338947

[17] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 396–407. https://doi.org/10.1145/3106237.3106285

[18] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. StoryDroid: Automated Generation of Storyboard for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 596–607. https://doi.org/10.1109/ICSE.2019.00070

[19] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 1724–1734. https://doi.org/10.3115/v1/D14-1179

[20] Nathan Cooper, Carlos Bernal-Cardenas, Oscar Chaparro, Kevin Moran, and Denys Poshyvanyk. 2021. It Takes Two to Tango: Combining Visual and Textual Information for Detecting Duplicate Video-Based Bug Reports. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 957–969. https://doi.org/10.1109/ICSE43902.2021.00091

[21] E. W. Dijkstra. 1959. A Note on Two Problems in Connexion with Graphs. *Numer. Math.* 1, 1 (dec 1959), 269–271. https://doi.org/10.1007/BF01386390

[22] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for me! characterizing non-reproducible bug reports. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, Hyderabad, India, 62–71. https://doi.org/10.1145/2597073.2597098

[23] Mattia Fazzini, Kevin Patrick Moran, Carlos Bernal-Cardenas, Tyler Wendland, Alessandro Orso, and Denys Poshyvanyk. 2022. Enhancing Mobile App Bug Reporting via Real-time Understanding of Reproduction Steps. *IEEE Transactions on Software Engineering* (2022), 1–1. https://doi.org/10.1109/TSE.2022.3174028

[24] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Amsterdam Netherlands, 141–152. https://doi.org/10.1145/3213846.3213869

[25] Sidong Feng and Chunyang Chen. 2022. GIFdroid: automated replay of visual bug reports for Android apps. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1045–1057. https://doi.org/10.1145/3510003.3510048

[26] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI Testing of Android Applications Via Model Abstraction and Refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 269–280. https://doi.org/10.1109/ICSE.2019.00042

[27] William G.J. Halfond. 2008. Web Application Modeling for Testing and Analysis. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), Doctoral Symposium.*

[28] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. 2020. Duplicate Bug Report Detection Using Dual-Channel Convolutional Neural Networks. In *Proceedings of the 28th International Conference on Program Comprehension*. ACM, Seoul Republic of Korea, 117–127. https://doi.org/10.1145/3387904.3389263

[29] Abram Hindle, Anahita Alipour, and Eleni Stroulia. 2016. A contextual approach towards more accurate duplicate bug report detection and ranking. *Empirical Software Engineering* 21, 2 (April 2016), 368–410. https://doi.org/10.1007/s10664-015-9387-3

[30] Yuchao Huang, Junjie Wang, Liu Zhe, Song Wang, Chunyang Chen, Mingyang Li, and Qing Wang. 2023. Context-aware Bug Reproduction for Mobile Apps. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, Melbourne, Australia.

[31] Jack Johnson, Junayed Mahmud, Tyler Wendland, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2022. An Empirical Investigation into the Reproduction of Bug Reports for Android Apps. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Honolulu, HI, USA, 321–322. https://doi.org/10.1109/SANER53432.2022.00048

[32] Alina Lazar, Sarah Ritchey, and Bonita Sharif. 2014. Generating duplicate bug datasets. In *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*. ACM Press, Hyderabad, India, 392–395. https://doi.org/10.1145/2597073.2597128

[33] Changlin Liu, Hanlin Wang, Tianming Liu, Diandian Gu, Yun Ma, Haoyu Wang, and Xusheng Xiao. 2022. ProMal: precise window transition graphs for android via synergy of program analysis and machine learning. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 1755–1767. https://doi.org/10.1145/3510003.3510037

[34] Hui Liu, Mingzhu Shen, Jiahao Jin, and Yanjie Jiang. 2020. Automated classification of actions in bug reports of mobile apps. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 128–140. https://doi.org/10.1145/3395363.3397355

[35] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided Bug Crush: Assist Manual GUI Testing of Android Apps via Hint Moves. In *CHI Conference on Human Factors in Computing Systems*. ACM, New Orleans LA USA, 1–14. https://doi.org/10.1145/3491102.3501903

[36] S.P. Meyn and R.L. Tweedie. 1993. *Markov Chains and Stochastic Stability*. Springer-Verlag, London.

[37] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. https://doi.org/10.48550/arXiv.1301.3781 arXiv:1301.3781 [cs].

[38] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed Representations of Words and Phrases and their Compositionality. In *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. Curran Associates Inc., Red Hook, NY, USA, 9.

[39] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Christopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A Practical Tool for Automated Testing of Android Applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, Buenos Aires, 15–18. https://doi.org/10.1109/ICSE-C.2017.16

[40] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing bug reports for Android applications. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, Bergamo Italy, 673–686. https://doi.org/10.1145/2786805.2786857

[41] Per Runeson, Magnus Alexandersson, and Oskar Nyholm. 2007. Detection of Duplicate Defect Reports Using Natural Language Processing. In *29th International Conference on Software Engineering (ICSE'07)*. IEEE, Minneapolis, MN, USA, 499–510. https://doi.org/10.1109/ICSE.2007.32 ISSN: 0270-5257.

[42] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, Vol. 1. ACM Press, Cape Town, South Africa, 485. https://doi.org/10.1145/1806799.1806870

[43] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. 2016. What Makes a Satisficing Bug Report?. In *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, Vienna, Austria, 164–174. https://doi.org/10.1109/QRS.2016.28

[44] Yang Song, Junayed Mahmud, Ying Zhou, Oscar Chaparro, Kevin Moran, Andrian Marcus, and Denys Poshyvanyk. 2022. Toward Interactive Bug Reporting for (Android App) End-Users. In *In Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22)*. ACM, Singapore, Singapore, 13. https://doi.org/10.1145/3540250.3549131

[45] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 245–256. https://doi.org/10.1145/3106237.3106298

[46] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, Vol. 1. ACM Press, Cape Town, South Africa, 45. https://doi.org/10.1145/1806799.1806811

[47] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*. ACM Press, Leipzig, Germany, 461. https://doi.org/10.1145/1368088.1368151

[48] Tyler Wendland, Jingyang Sun, Junayed Mahmud, S. M. Hasan Mansur, Steven Huang, Kevin Moran, Julia Rubin, and Mattia Fazzini. 2021. Andror2: A Dataset of Manually-Reproduced Bug Reports for Android apps. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, Madrid, Spain, 600–604. https://doi.org/10.1109/MSR52588.2021.00082

[49] Wei Yang, Mukul R. Prasad, and Tao Xie. 2013. A Grey-Box Approach for Automated GUI-Model Generation of Mobile Applications. In *Fundamental Approaches to Software Engineering (Lecture Notes in Computer Science)*, Vittorio Cortellessa and Dániel Varró (Eds.). Springer, Berlin, Heidelberg, 250–265. https://doi.org/10.1007/978-3-642-37057-1_19

[50] Zhaoxu Zhang, Robert Winn, Yu Zhao, Tingting Yu, and William G.J Halfond. 2023. Automatically Reproducing Android Bug Reports Using Natural Language Processing and Reinforcement Learning. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2023)*.

[51] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William G. J. Halfond, and Tingting Yu. 2022. ReCDroid+: Automated End-to-End Crash Reproduction from Bug Reports for Android Apps. *ACM Transactions on Software Engineering and Methodology* 31, 3 (July 2022), 1–33. https://doi.org/10.1145/3488244

[52] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 128–139. https://doi.org/10.1109/ICSE.2019.00030

[53] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William G.J. Halfond. 2019. ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports. In *Proceedings of the ACM/IEEE International Conference on Software Engineering (ICSE)*.

[54] Hao Zhong. 2022. Enriching Compiler Testing with Real Program from Bug Report. In *37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–12. https://doi.org/10.1145/3551349.3556894

[55] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering* 36, 5 (Sept. 2010), 618–643. https://doi.org/10.1109/TSE.2010.63

Received 2023-09-28; accepted 2024-04-16