Automatically Reproducing Android Bug Reports using Natural Language Processing and Reinforcement Learning

Zhaoxu Zhang
University of Southern California
USA
zhaoxuzh@usc.edu

Robert Winn
University of Southern California
USA
rwinn@usc.edu

Yu Zhao
University of Central Missouri
USA
yzhao@ucmo.edu

Tingting Yu
University of Cincinnati
USA
yutt@ucmail.uc.edu

William G.J. Halfond
University of Southern California
USA
halfond@usc.edu

ABSTRACT

As part of the process of resolving issues submitted by users via bug reports, Android developers attempt to reproduce and observe the crashes described by the bug reports. Due to the low-quality of bug reports and the complexity of modern apps, the reproduction process is non-trivial and time-consuming. Therefore, automatic approaches that can help reproduce Android bug reports are in great need. However, current approaches to help developers automatically reproduce bug reports are only able to handle limited forms of natural language text and struggle to successfully reproduce crashes for which the initial bug report had missing or imprecise steps. In this paper, we introduce a new fully automated approach to reproduce crashes from Android bug reports that addresses these limitations. Our approach accomplishes this by leveraging natural language processing techniques to more holistically and accurately analyze the natural language in Android bug reports and designing new techniques, based on reinforcement learning, to guide the search for successful reproducing steps. We conducted an empirical evaluation of our approach on 77 real world bug reports. Our approach achieved 67% precision and 77% recall in accurately extracting reproduction steps from bug reports, reproduced 74% of the total bug reports, and reproduced 64% of the bug reports that contained missing steps, significantly outperforming state of the art techniques.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Android, Bug Reproduction, Reinforcement Learning, Natural Language Processing

ISSTA '23, July 17–21, 2023, Seattle, WA, USA

ACM ISBN 979-8-4007-0221-1/23/07.
https://doi.org/10.1145/3597926.3598066

ACM Reference Format:
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 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '23), July 17–21, 2023, Seattle, WA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597926.3598066

1 INTRODUCTION

In the hyper-competitive world of app marketplaces, app developers strive to provide their users with interesting features and high-quality functionality to distinguish themselves from their competition. An important mechanism for receiving feedback from users is bug reporting systems, such as GitHub Issues Tracker [10] and Google Code [11]. These systems enable users to create bug reports in which they can describe the observed failures[1] and provide reproduction steps. Developers can use this information to help debug their apps. However, the use of this information is complicated by the fact that the bug reports are often informally written in natural language, imprecise, and incomplete [25, 31, 45]. This can make it challenging for developers to reproduce the reported failure, as important steps may be missing or poorly described. Even with well-written bug reports, reproduction can still be challenging since mobile apps often have complex event-driven user interfaces that allow many similar sequences of actions, each of which may or may not lead to the reported failure. Taken together, these aspects can make bug report reproduction a time-consuming and error-prone process, which can undermine the usefulness of the bug reporting process.

The software engineering community has tried to address this problem through the development of automated bug report reproduction techniques (e.g., [23, 41, 42]). These approaches generally have two phases. In the first phase (i.e., bug report analysis), the approaches analyze the natural language in the bug reports in order to identify the steps to reproduce (S2Rs). Each S2R describes an action on a user interface (UI) element in the app under test (AUT). In the second phase (i.e., app exploration), the reproduction approaches attempt to execute each S2R on the AUT. Both phases represent significant challenges that make it difficult to fully automate this process. In the first phase, the natural language is generally unstructured, written by users without a technical background, and

[1] We use the word "failure" and "crash" interchangeably in this paper.

has similar concepts described in a multitude of ways [23, 27]. Even if the first phase could be done perfectly, many bug reports have missing steps [25, 39]. This complicates the second phase, since the approaches must either find some ways to identify plausible missing steps or dead end in their reproduction efforts when no UI element matches the next S2R. This happens, for example, when a missing S2R specifies a step that causes a UI element to appear that is then used by the subsequent S2Rs in the bug report.

Two state of the art approaches for reproducing crashes from Android textual bug reports, Yakusu [23] and ReCDroid [43], define techniques for handling these challenges. To address the challenge in the first phase, both approaches extract S2Rs from bug report text using manually crafted patterns and predefined word lists that map to standard actions. For example, for the phrase "click the Home button," these approaches would identify a click action with the "Home button" as the target. However, these techniques are unable to handle natural language in bug reports with either previously unseen words or different sentence structures. Maca [27] designs a classifier to normalize action words into a standard form, but also uses simple patterns to parse the sentence and therefore has limitations when handling previously unseen sentence structures. To address the challenge in the second phase, both approaches employ a greedy strategy to explore the app and identify possible mappings of S2Rs to UI events. However, the greedy strategy can lead the approaches to prioritize matching an S2R with a UI event that is the most similar, even though choosing a lower similarity match may allow subsequent S2Rs to match better with other UI events. This may occur, for example, when there are inaccurately described or missing steps.

To address the aforementioned challenges, we designed a new approach for reproducing crashes from Android bug reports. Our approach follows the same two-phase architecture as prior approaches, however, for each phase, we developed new algorithms and designs that enable our approach to be *more broadly applicable* – able to handle a wider variety of natural language in bug reports – and *more successful* in matching S2Rs extracted from the bug report to actions in an app's UI. We achieved the first of these by developing a set of analyses that extract S2R information without relying on predefined patterns. To improve reproduction, we formulate the task of matching S2Rs with an app's UI events into a Markov decision process (MDP). Our approach then identifies the reproduction event sequence by leveraging reinforcement learning (RL) algorithms, specifically Q-learning [38]. Our approach employs Q-learning to learn how to match the UI events with the S2Rs in such a way as to bridge missing steps and calculate an overall best match between S2Rs and a UI event sequence that can lead to the observed failure. We implemented our approach as a prototype tool and compared it against ReCDroid and Yakusu. Our results show that our natural language processing (NLP) techniques help our approach to handle a wider variety of bug reports and our RL based exploration leads to a more successful reproduction phase. Together, these two techniques enable our approach to outperform ReCDroid and Yakusu by a significant margin and indicate that our approach significantly improves state of the art bug report reproduction techniques.

In summary, our paper makes the following contributions:

- We designed a novel NLP based analysis to analyze and extract reproduction steps from Android bug reports.
- We designed a new exploration strategy, based on reinforcement learning, for exploring Android apps and finding the match between steps and UI events.
- Based on the above two contributions, we developed a novel approach to reproduce Android bug reports and implemented it as a tool.
- We conducted an empirical evaluation showing the performance of our approach.
- We made the implementation and dataset publicly available for future research work [12].

2 MOTIVATING EXAMPLE

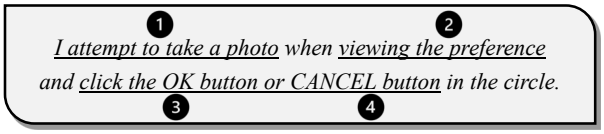


Figure 1: Bug Report Example

In this section we introduce a motivating example that we will use throughout the paper to illustrate our approach and highlight the limitations of existing approaches. The bug report, which is shown in Figure 1, is based on bug reports that we found in our evaluation subjects. The bug report describes a crash that occurs when the user tries to take a photo on the preference page. It reports four steps (as annotated) that need to be taken for reproduction. Although this report seems reasonably clear and is, in fact, typical of most bug reports, it can cause several problems for the state of the art automated bug report reproduction techniques (e.g., ReCDroid [43], Yakusu [23]) when they try to reproduce it. To reproduce a bug report, these automated techniques first analyze the natural language of the bug report to determine the S2Rs. Each S2R is a structured form that contains information describing the step such as UI action and target widget. Although the text is seemingly easy to parse (for a human), these automated techniques would run into several problems when trying to do so automatically. Both techniques try to identify an S2R by matching text in the bug report to a pre-defined vocabulary of words describing UI actions. This would make them fail to extract the first step in our example, which uses a verbal phrase "attempt to take" to express the UI action, an unusual way to describe a UI action that is not present in either approach's vocabulary. Second, after identifying a UI action, both techniques rely on hand-crafted rules to match other S2R entities. However, none of these rules are able to identify the target "photo". Alternatively, a misidentification may also happen if the rules are too simple. For example, ReCDroid would identify the target of the fourth step to be "CANCEL" and "circle". These inaccurate entity identifications would create noise when the automated techniques attempt to match the S2Rs in the AUT. Additionally, both approaches suppose the execution order of S2Rs is the same as their syntactic order in the text. However, this could lead both of them to obtain the wrong order of the first two steps. As indicated by the semantics of the connective word "when" between the first

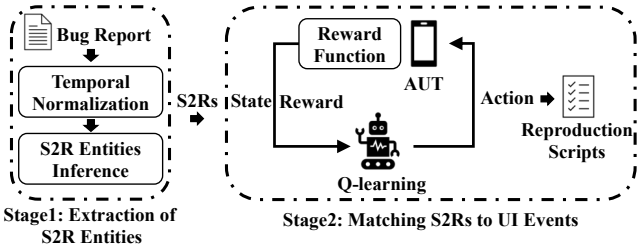


Figure 2: Workflow of Our Approach

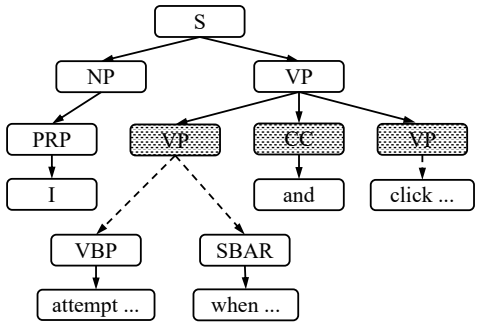
two steps, the second step should happen first and then be followed by the first step.

Even assuming there is no noise in the S2Rs, these approaches would face additional challenges during reproduction when they try to automatically match S2Rs against UI events. First, both techniques employ a greedy approach when matching the S2Rs with UI events, meaning that they always choose the event that is the most closely matched with the S2R. This could lead to a local maximum in the matching process, effectively trapping the exploration and preventing it from finding a better overall match. Second, both techniques assume that missing steps only occur when there is no UI event matched with the S2Rs during the exploration. However, the missing steps could occur exactly when there are events that match the S2Rs. When the above cases happen, prior reproduction tools would either fail to reproduce the failure or reduce to an exhaustive exploration of the AUT.

These limitations of the state of the art directly motivate the design of our approach. Given a bug report sentence, our approach first normalizes the execution order of S2Rs in the sentence. To identify the entities for a S2R, instead of using predefined patterns, we designed an analysis that infers them from a set of more general syntactic parts that could be identified from any natural language sentence. These two steps together allow our approach to have a more accurate understanding of the bug report sentence and are also more widely applicable. To match S2Rs and UI events, we designed an exploration and matching strategy based on MDP and Q-learning. These techniques enable our exploration approach to find a way to bridge possible gaps in the S2Rs and at the same time avoid local search maxima (i.e., that would be found with a greedy approach). In the next section, we explain our approach's design and algorithms for doing this in more detail.

3 APPROACH

The goal of our approach is to automatically reproduce the crashes described in the text of Android bug reports. The input of our approach is S2R sentences — natural language sentences describing reproduction steps, and the AUT. At a high-level, Figure 2 shows the two stages of our approach: Extraction of S2R Entities and Matching S2Rs to UI Events. In the first stage, our approach analyzes the S2R sentences to extract the S2R entities defining each step in the bug report. The entities we extracted include information such as the type of UI action to be performed and the target of the action. Our approach leverages a combination of natural language techniques to systematically analyze S2R sentences to extract the entities more accurately. In the second stage, our approach explores the app to match the S2Rs to UI events in the AUT. We formulate this task as a Markov decision process (MDP) and use the Q-learning algorithm to identify the matched UI events. The combination of MDP and Q-learning allows our approach to effectively bridge missing steps and identify the UI events for reproduction.



(a) Constituency Parsing Tree

2 *I view the preference.*
1 *I attempt to take a photo.*
3 *I click the OK button in the circle.*
4 *(I click the CANCEL button in the circle)*

(b) Reordered Sentences

Figure 3: Example of S2Rs Reordering

3.1 Extraction of S2R Entities

The first phase of our approach analyzes the given sentences and extracts the individual steps from them. For each of these sentences our approach first carries out a temporal normalization analysis that extracts the S2Rs in the sentence, converts them to standalone sentences, and then reorders them based on the implied temporal relationships in the original sentence (Section 3.1.1). The ordered standalone sentences are then each analyzed to infer the entities of an S2R — important reproduction information, such as the target widget, action type, and input values (Section 3.1.2). Our S2R entity inference is notable from related work since it does not require the sentence to match one of a predetermined set of patterns. The final output of the first phase is a list of S2Rs entities, each corresponding to a reproduction step that the second phase of our approach (Section 3.2) will use as it attempts to reproduce the bug report.

3.1.1 Normalizing the Temporal Order of S2Rs. The first step of the extraction analysis normalizes the temporal order of S2Rs described in a bug report sentence. For example, this step would address the problem shown in the text of Section 2, where the bug report sentence has multiple conjuncted S2Rs whose intended execution order is not the same as their syntactic ordering. Reasoning about the temporal relations among S2Rs is challenging due to the use of complex sentence structures, such as nested clauses and phrases. Our insight is that this temporal ordering information can be extracted by accounting for the semantics of connectives utilized between clauses or phrases in a bug report sentence. To take advantage of this insight, we designed an analysis that, given an S2R sentence,

recursively: (1) extracts the conjuncted S2R text spans (i.e., clauses or phrases) and connectives, (2) converts each S2R text span into a standalone sentence, and then (3) reorders the standalone sentences based on the connectives between them. Each round of our analysis generates a pair of reordered standalone sentences. Our analysis is recursively performed on the generated sentences until no more conjuncted S2Rs can be identified. The output of our analysis is a list of the standalone sentences in the inferred order. We next describe the details of each part of our analysis in detail, illustrating the analysis using the sentence in Figure 1.

As a preprocessing step, each round of our analysis takes a given sentence and converts it into a standard structure for representing a natural language sentence. This structure is known as the constituency parsing tree (CPT) and captures the syntax of a sentence from the perspective of constituency grammars [26]. Figure 3a shows the CPT of the sentence in Section 2, which is computed using standard NLP parsers [19]. Each terminal node of the CPT denotes a word in the sentence and each non-terminal node represents a type of constituency, which captures the use of all terminal nodes in its sub-tree as a single unit [28] (e.g., *NP* for a noun phrase).

Given the CPT, our analysis first extracts the conjuncted S2R text spans (i.e., clauses and phrases) and their connectives in the sentence. In order to extract the text spans, our analysis traverses each constituency tag in the CPT from top to bottom and identifies two types of conjunction: coordination and subordination, as defined in standard English grammar rules [26]. To identify coordination, our approach searches for tags whose parent-child structure satisfies the coordination grammar [26]. For example, the grammar for coordinated clauses is defined as "$S \rightarrow S\ CC\ S$", which indicates that a parent constituency *S* (sentence) has three children: two sub-clauses and a connective *CC* between them. By identifying such a parent constituency *S*, our analysis can find the two coordinated sub-clauses and their connectives from its children. To identify subordination, our approach searches for its corresponding constituency tag, *SBAR*, which represents subordinating clause, and takes the words rooted at this tag as the text for a subordinate clause and the first word in the clause as the connective. This step of our analysis returns the first pair of conjuncted text spans it identifies as it traverses the CPT from top to bottom. This ensures that every round of analysis identifies the most outer conjuncted text spans. For example, given the CPT in Figure 3a, our approach identifies two coordinated verbal phrases (VP) as highlighted and the word "and" as the connective in the first round of analysis.

The next step of our analysis transforms each text span identified by the previous step into a standalone sentence. This transformation is necessary since part of the text in the original sentence may be shared by the conjuncted texts (e.g., the subject "I" is shared by all the four S2Rs in the sentence in Figure 1). Only extracting and reordering the conjuncted text will lose such information. To do this, our approach first removes the conjuncted constituency tags (identified in the previous step) and their sub-tree from the original CPT. This gives us the part of the original CPT that is shared by the conjuncted text spans. Then our approach duplicates this part and joins it to the subtree of each conjuncted text to form new CPTs, representing the transformed sentences. To illustrate this transformation, after identifying the two highlighted verbal phrases in Figure 3a, our approach extracts the left subtree of the root node, which indicates the subject "I", duplicates it, and rejoins it to the subtree of each verbal phrase. This forms two standalone sentences.

The last part of our analysis infers the intended execution order of the transformed standalone sentences. Our inference is based on the semantics of the connectives between the sentences. To obtain a comprehensive view of connectives used in English text and their semantics, we referred to the Peen Discourse Treebank (PDTB) [30, 33]. The PDTB contains a large-scale annotation on the English connectives and provides a comprehensive categorization based on their semantics. Our approach focuses on the connectives with semantics in two categories from PDTB: *temporal succession* and *alternative*. Connectives in the first category indicate that the second conjuncted text span happens earlier than the first conjuncted span, and connectives in the second category indicate that the conjuncted text spans could be used alternatively to represent the meaning of the sentence. The reason to focus on these two categories is that connectives within them would affect the temporal order or the selection of S2Rs. To perform the temporal inference, given the connective, our approach first checks which category it belongs to. If the connective belongs to the first category, our approach reverses the order of conjuncted sentences. If the connective belongs to the second category, our approach only selects one from the given sentences. For the example in Figure 1, our approach would reverse the order of standalone sentences generated by the first two S2R text spans as it finds the connective "when" indicating a temporal succession semantics. Figure 3b displays the final sentence list generated by our tool where each sentence only contains one reproduction step and their ordering is normalized.

3.1.2 Inferring S2R Entities from Standalone Sentences. The goal of this part of the approach is to infer S2R entities from the standalone sentences produced by the analysis in Section 3.1.1. These S2R entities will be used in the exploration phase of the approach (i.e., Section 3.2) to navigate the UI of the AUT. We formally define the entities of a S2R as ⟨target widget, UI action, input value, target direction⟩. Taken together, these elements represent the UI event described in the standalone sentence. Prior work [23, 43] targeted the identification of a similar set of S2R entities. However, as we discussed in Section 2, these approaches can only work for predefined sentence patterns since they require a mapping of parts of the pattern to the S2R entities. Our insight is that each of these S2R entities can instead be inferred based on more general syntactic constituents (i.e., subject, predicate, object, and modifier) of a sentence. For example, the predicate of a sentence, regardless of where it appears in a given sentence, can be used to infer the action performed by the user. This insight allows our approach of extracting S2R entities to work on any standalone sentence, regardless of its form, as long as it can be decomposed into these syntactic constituents — a condition that would be satisfied by any standalone sentence produced by our analysis in Section 3.1.1. Our approach leverages this insight as follows. First, given a standalone sentence produced by the analysis in Section 3.1.1, our approach decomposes the sentence into its syntactic constituents using standard open information extraction (Open IE) techniques [15, 16, 20, 21, 34, 40]. In the second step, our approach identifies the text in the sentence that defines the S2R entities by using inference rules based on these syntactic constituencies. In the remainder of this section, we first

give an example of the syntactic constituents that our approach works on and then define the inference rules we use for each entity.

For background purposes, we provide an example of the syntactic constituents using the last sentence in Figure 3b. The predicate of this sentence is "click," the subject is "I", the object is "the CANCEL button," and the modifier is the phrase "in the circle". Notably, not all of them exist in every sentence. To retrieve all these constituents, our approach in practice employs a recent Open IE technique [16]. Next, we explain the rules we defined to infer each S2R entity.

The **target widget** is the text description of the UI widget that the user interacts with in the S2R. The part of the sentence that contains the target widget depends on the voice with which the sentence was written. If the sentence is written in an active voice, the target widget is defined in the sentence object (e.g., "I click the button"). Otherwise, if the sentence is written in the passive voice, the target widget is defined in the subject of the sentence (e.g., "The button is clicked"). Therefore, to identify the target widget, our approach first infers the voice of the sentence. The voice can be determined simply by checking whether the predicate is in a form of a static verb and a verb in its past participle (e.g., is clicked) [1]. Based on this determination, our approach extracts either the subject or object text, which is part of the syntactic constituents defined previously and then uses this text for the target widget.

The **UI action** represents the type of interaction that the sentence describes as performed on the AUT. This naturally corresponds to the predicate part of the sentence. If the sentence does not have a predicate (e.g., "Setting button."), our approach takes click as the default UI action. Otherwise, our approach maps the predicate to one of the five standard actions that can be performed on an Android UI: click, input, rotate, swipe, and scroll. However, the key challenge is that actions may be described using a wide range of words, so there is no direct mapping from the predicate to the known action types. Existing approaches rely on a predetermined list of synonyms for each known action type to identify a direct mapping. The limitation of this approach is that if the predicate uses previously unseen words, it cannot be classified. Our insight is to classify the UI action type of a given predicate based on its semantic similarity with a set of synonyms defined for each standard UI action.

Our approach for action type identification is as follows. We assume the availability of a word list, as used in related approaches (e.g., [43]), that contains a group of synonyms for each standard action type. For example, "press" and "tap" for the click action. To handle action types described using words that are not in the list, we introduce the idea of semantic similarity to the action type identification. Our approach computes the semantic similarity of the predicate with a group representing the action type and its synonyms. The semantic similarity enables our approach to know which type of standard UI action the predicate is most similar to semantically. Following the classic method [29], given two words or phrases, we compute their semantic similarity as the cosine distance of their word embeddings, the vector representation of a text, obtained from a pre-trained language model, such as Word2Vec [29]. Our approach considers the group with the highest semantic similarity to be the inferred action type. However, if the similarity score difference of the top two groups is within a threshold δ then both of the corresponding action types are considered during the reproduction phase (i.e., Section 3.2). This aspect of our approach allows it to more flexibly handle words that could be interpreted in multiple ways. For example, the word "change" could be interpreted as a click action or an input action in different cases. If the inferred action type is rotate, our approach also checks whether its inferred target widget describes the screen or the device. If neither is described, our approach chooses the next highest scored UI action as the result. The reason for this is that intuitively the rotation action is performed on the whole device or the screen instead of a specific widget, so if neither is the target widget, then the action is not rotate. This heuristic helps our approach identify rotation action more accurately.

The **input value** is the text to be entered for an input action. This information can be inferred by analyzing the object and the modifier part of a sentence. Due to the way actions are described in the English language, input values are associated with their targets using prepositional phrases. This is captured by the modifier of a sentence. However, depending on the preposition used in the modifier, the input value may be contained in different constituents of the sentence. For example, for the sentence "I enter A on B", the input value A is captured by the object. However, for the sentence "I fill A with B", the input value B is contained in the modifier. Therefore, our approach identifies the input value based on the preposition used in the modifier. If the modifier starts with the preposition "with", our approach extracts the text following it as the input value. If the modifier starts with a preposition in {"in", "on", "into", "onto", "at"}, our approach uses the object as the input value. It also replaces the previously identified target widget with the modifier text. In the case that there is no modifier identified in the given sentence, which happens when reporters only specified the input value or target widget as the object of the sentence (e.g., "I enter a number 13"), our approach takes the object text as the input value. Our approach employs two heuristics to refine the identified input value text to make it more precise. First, if the input value text contains numbers, our approach only keeps the number as the input value. The reason for this heuristic is that reporters may include other words or phrases when describing the input number as the underlined phrase in the previous example. However, the input value to reproduce the failure needs to be very precise, in this case, only the number is entered. Second, if the text uses a text-based description of a special value, such as "space", our approach replaces it with the corresponding literal value.

The **target direction** defines the intended direction for a scrolling action (i.e., up or down) or a swiping action (i.e., left or right). The direction is also expressed in the object portion of a sentence. For example, "I scroll down". Given the simplistic nature of this entity, we found it sufficient for our approach to directly search for directional keywords in the object text and use those as the target direction.

3.2 Matching S2Rs to UI Events

The second phase of our approach explores the AUT to match the S2Rs to UI events that reproduce the failure. As explained in Section 2, for state of the art techniques, their exploration-based process of finding such UI events is limited due to the problems of local optimums and missing steps. Our approach addresses these

challenges by defining the matching problem as a Markov decision process (MDP) and using a type of reinforcement learning algorithm, Q-learning, to find the desired UI event sequence.

The combination of MDP and Q-learning is well-suited for our problem domain and addressing these challenges. The expected reward used by Q-learning combines the immediate reward for the current action and the accumulated reward of potential future actions. This enables our approach to avoid local optimums by allowing it to consider and evaluate a UI event that itself may not be the closest match with the S2Rs but leads to subsequently better matched UI events. For missing steps, a similar mechanism can be leveraged. In this case, a no operation (no-op) is added to the actions in the MDP for a given state. When combined with the Q-learning future rewards, this effectively allows our approach to evaluate the total expected rewards if it assumes the next step is missing. The no-op action can be added repeatedly to simulate the possibility of multiple missing steps. Taken together, the combination of Q-learning and MDP enables our approach to break out the local optimum and bridge missing steps effectively.

In the following sections, we first provide a formal definition of our instance of the MDP, which maps it to the problem of S2R matching (Section 3.2.1). Next, we explain how our approach utilizes Q-learning on this MDP to find a UI event sequence that reproduces the bug report (Section 3.2.2).

3.2.1 Formulation of Markov Decision Process. In our approach, we define an instance of the MDP to describe the process of matching S2Rs with UI events in the AUT. The MDP is formally defined as a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$ where $\mathcal{S}$ is the state set, $\mathcal{A}$ is the action set, $\mathcal{P}$ is the transition function, and $\mathcal{R}$ is the reward function. We define each part of the MDP for our problem as follows:

$\mathcal{S}$: *States* The state represents the MDP at each time step. We define a state of the MDP as $s = \langle H, RS, n \rangle$. The first element H is the current view hierarchy (VH) of the AUT, which includes all the widgets on the UI as well as their attributes. The second element RS is a list of the remaining S2Rs that are not yet matched. The last element n is an integer indicating the remaining amount of no-op actions that are available for the matching process. Our approach defines a finite limit on no-ops actions to avoid the situation where the reproduction phase can endlessly explore an app along all paths with no likely matches. Allowing an infinite number of missing steps would be unrealistic, implying a bug report that had an infinite (or extremely large number) of missing steps. We define the initial state as $s_0 = \langle H_0, RS_0, n_0 \rangle$ where H_0 is the VH of the UI where the reproduction starts, RS_0 is a list containing all of the S2Rs in the bug report identified by the first stage analysis of our approach and n_0 is the total number of allowed no-op actions during whole matching process. We define the terminal state of the MDP as s_t, which has no available actions, i.e., it is the ending of the matching process. This happens when our approach matches all S2Rs and used all available no-op actions.

$\mathcal{A}$: *Actions* In the context of S2R matching problem, the action represents a match between a UI event and an S2R. (Note, this terminology can be confusing since in Section 3.1.2, action refers to the standard Android operations, such as click and swipe.) We formally represent an action a as $\langle e, rs \rangle$ where e is the UI event that is chosen to be executed on the AUT and rs is the step matched with the UI event. To handle missing steps, we define a special action, no-op, which is denoted as $\langle e, - \rangle$ where the dash represents a placeholder for a missing step. We will discuss how our approach identifies available actions at a given state in Section 3.2.2.

$\mathcal{P}$: *Transition Function* The transition function of an MDP takes in the current state s and the selected action a and returns the next state s'. In our problem domain, the transition to next state happens in two steps. First, our approach executes the UI event $a.e$ on the AUT and extracts the new VH produced by the app. Then, our approach dequeues the selected step from $s.RS$ or decreases $s.n$ if a is a no-op action. The new VH and the possibly updated values of RS and n are returned as the new state.

$\mathcal{R}$: *Reward Function* The reward function generates a value indicating the quality of the action, i.e., the quality of the match between an S2R and a UI action. Our reward function evaluates an action as a sum of three subscores: similarity score, exploration penalty, and failure state penalty.

The *similarity score* measures the similarity between an action's UI event and S2R. It guides our approach to explore UI events closely related to the descriptions in the bug report. The computation of similarity scores is done in two ways. First, for actions matching with UI events interacting with a specific widget, i.e., click and input events, the similarity score is computed as the textual similarity between the description of the widget on the UI and the target widget entity of the S2R. This textual similarity indicates how related the UI event is to the S2R. To do this, our approach analyzes the VH of the UI and extracts three attributes of the widget as its description: (1) *text*: This attribute is the text appearing on the widget, which is readily available to bug reporters; (2) *id*: This attribute contains the file name of the linked resources used in the widget, such as icons. The file name is normally meaningful and descriptive of the content it contains [23]; and (3) *content description*: This is the description of the widget defined by the developers. Its content is used by Android accessibility service (e.g., screen reader) when describing a widget to people with disabilities [9]. Therefore it is supposed to contain a meaningful and informative explanation of the widget. Our approach computes the semantic similarity of the text in the target widget entity with each of these three description texts, using the same approach described in Section 3.1. Our approach uses the highest of these three values as the overall similarity score, as long as the value is above a threshold d that represents the similarity of non-synonym words [43]. However, if the value is not above this threshold then our approach considers them not to be synonyms and assigns a default negative score r_d. The rationale for using the maximum of the three values is that our approach cannot know which of these three descriptions sources would be used by the bug reporter, so this mechanism allows for using the best or most informative fit. Second, for actions that match with rotate, scroll, swipe, or the no-op, our approach assigns their similarity score as the default score r_d. The reason for this is that for these actions, the matched UI event does not interact with a specific UI widget or the matched S2R is not a concrete step from the bug report. Therefore, our approach cannot compute a score to show the relevance or similarity for the UI event and the S2R.

The exploration penalty is designed to encourage our approach to select meaningful UI events during exploration (i.e., the ones that result in changes on the UI). For example, in the UI with a tab, it is

Algorithm 1: Matching S2Rs to UI Events

Input: the app under test AUT, steps to reproduce $S2Rs$, timeout t, error message m
Output: reproduction UI events

```
1  Q ← ∅ ;                                  /* initialize Q-table */
2  while ¬timeout(t) do
3      s_{i+1} ← s_0 ;                      /* reset to initial state */
4      E ← ∅;
5      while true do
6          s_i ← s_{i+1};
7          A ← inferActions(s_i);
8          if randomNum() < ε then
9              a_i ← rand({a ∈ A | a ≠ arg max_{a'} Q(s, a')});
10         else
11             a_i ← arg max_a Q(s, a);
12         end
13         E ← E ∪ a_i.e;
14         s_{i+1} ← P(a_i, s_i, AUT);
15         if success(m) then
16             return E;
17         end
18         r ← R(a_i, s_i, s_{i+1});
19         Q(s_i, a_i) ← (1 − α)Q(s_i, a_i) + α(r + γQ*(s_{i+1}, a));
20         if s_{i+1} is a terminal state then
21             break;
22         end
23     end
24 end
```

meaningless to click the current tab button again. To do this, our approach evaluates the VH before and after executing the selected UI event. If all the widgets and their attributes on the VH remain the same, our approach gives an exploration penalty to the action. Note that our definition of state includes the attributes of each widget in the view hierarchy. Therefore, UI events, such as clicking checkboxes or inputting, will change the attributes of widgets (in this case, the "checked" or "text" attribute will be changed) and our approach will not generate an exploration penalty for these events.

The *failure state penalty* r_f is assessed when the selected action leads to a terminal state that does not reproduce the observed failure. This penalty discourages the exploration of event sequences that do not lead to the observed failure.

3.2.2 Match S2Rs to UI Events. This part of our approach utilizes Q-learning to find a UI event sequence that matches the given S2Rs and reproduces the observed failure of the bug report. Our approach is described in Algorithm 1. As input, our approach takes the app under test (AUT), steps to reproduce (S2Rs), a time budget (t), and the error message (m) of the observed failure as inputs. Our approach then iteratively explores the AUT, guided by the Q-learning, to find a UI event sequence that reproduces the failure.

The iterative exploration starts from the initial state s_0 (Line 3). At each iteration, our approach first identifies all the possible actions ($\mathcal{A}$), available in the current state s_i (Line 7). Recall that an action in the MDP is actually a possible match between an S2R and a UI event. Our approach identifies the set of possible matches from two sources. First, the match set includes the cross product of all available UI events in the UI and the next S2R in $s_i.RS$, if the two represent the same UI action. In the case that our approach predicted two possible types of UI actions for a S2R, it includes matches for both of them. Second, the match set also includes a possible no-ops match for all UI events if the no-op limit has not been reached at s_i (i.e., $s_i.n > 0$).

After identifying the set of all possible matches, our approach selects one match and performs the matched UI event on the AUT. Our approach selects the match using the epsilon-greedy algorithm [36], a standard method for Q-learning. Specifically, with probability $1 - \epsilon$, our approach will choose the match with the highest Q-value (Line 11) or with probability ϵ, select a random one from the matches with lower Q-value (Line 9). In practice, our approach sets the ϵ with a low value so that our approach can focus on exploring the current optimal choice (i.e., the match with highest Q-value) in most time in order to match more S2Rs along this path, but also have chance to explore other random actions to break out the local optimum. It is worth emphasizing that, the epsilon-greedy policy gives our approach a chance, at any given state, to explore lower-scored matches which could be a match between a UI event that is not the most similar to the S2R or a match with a missing step holder. By doing this, our approach could evaluate their future rewards, which enables it to effectively escape the local optimal during the exploration and, in combination with the no-op, bridge missing steps. In the case that a previously unexplored state is matched, the initial Q-value for all the actions is set to its *similarity score* as defined in Section 3.2.1. By doing this, our approach is given a pre-knowledge of the potential reward of each action so it can be more effective in selecting a better match with less exploration. For the scroll, swipe, or rotate actions, since a similarity score is not defined, the approach assumes a similarity threshold d (defined in Section 3.2.1) as their initial Q-value. This adjustment ensures that these impactful actions will be explored. The selected UI event is stored in an event list, which is returned in the case of successful reproduction (Line 13). By calling the transition function, our approach executes the selected UI event on the AUT and updates the current state (Line 14).

After executing the UI event in the selected match, our approach then determines its reward using the reward function $\mathcal{R}$ (as defined in Section 3.2.1) and updates its Q-value using the Bellman function (Line 19). Note that α and γ are standard parameters defined in the Bellman function. In the case of encountering a terminal state (Lines 20 to 21), our approach resets the current state and restarts the exploration. Our approach terminates either when the timeout occurs or the failure is successfully reproduced (Line 16). The success of the reproduction is determined by checking whether the AUT crashes with the specified error message (Line 15).

4 EVALUATION

In this section, we evaluated the effectiveness and runtime of our approach. We also compared our approach against two state of the art Android bug report reproduction approaches, ReCDroid and Yakusu. Our evaluation addressed the following research questions:

RQ1 How effective is our approach in identifying S2Rs?
RQ2 How effective is our approach in reproducing Android bug reports?
RQ2.1 How helpful is the NLP technique for reproduction?
RQ2.2 How helpful is the RL-based exploration for reproduction?
RQ3 What is the running time of our approach?

4.1 Approach Implementations

For the purposes of the evaluation, we implemented a prototype, ReproBot, of our approach. The core algorithms of our approach are implemented in Python, and we leveraged functionality from several well-known libraries: Graphene [16] and OpenIE5 [3] to identify syntactic constituents in natural language text; UIAutomator [13] to interact with Android apps, and Spacy [24] to compute the semantic similarity of texts. Our implementation is publicly available via the project website [12]. We obtained the implementations of ReCDroid [43] and Yakusu [23], two state of the art reproduction tools on Android bug reports, from their public websites[5, 7]. Our experiments were performed on Android emulators on a physical x86 Ubuntu 20.04 machine with eight 3.6GHz CPUs and 32G of memory.

4.2 Dataset Collection

To evaluate our tool, we built a dataset containing 77 bug reports. We collected the bug reports from the artifacts of four related works: (1) the evaluation dataset of ReCDroid [42, 43]; (2) the evaluation dataset of Yakusu [23]; (3) an empirical study on Android bug report reproduction [25], and (4) an Android bug report dataset [39]. Altogether, there were 199 bug reports from these four sources. For each bug report, the authors interacted with the corresponding app to both ensure that the selected bug report was reproducible and to remove any duplicates. As part of this process, we also separately documented the steps that were missing from the bug report but that were necessary to reproduce the reported failure, and recorded the specific error message written to the Android device log when the failure occurred, which was then used by our approach to determine whether a bug report's crash was successfully reproduced. During this process, a total of 99 bug reports were found to be no longer reproducible and 23 were duplicates, leaving us with 77 bug reports. The lack of reproducibility of a given bug report was generally attributable to one or more of the following reasons: the lack of available apks or bug reports (due to an expired link), environmental configuration issues that were not resolvable, or failure to obtain the necessary setup information for the app (e.g., personal accounts or specific inputs). Additionally, we manually analyzed and identified the ground truth for each S2R within the 77 bug reports, specifically its UI action, target widget, input value, and target direction. The average, median, minimum and maximum steps of the ground truth of the bug reports were: 3, 3, 1, and 9. However, after manually reproducing each bug report and identifying the actual number of required steps for each bug report reproduction, we found that the average number of required steps increased to 6, the median to 5, and the maximum to 26, while the minimum remained at 1.

4.3 Experiment 1

4.3.1 Protocol. The first experiment evaluated the accuracy of ReproBot in identifying S2Rs and compared these results with those of ReCDroid and Yakusu. To do this, we ran ReproBot, ReCDroid, and Yakusu on our dataset and verified the correctness of all identified S2Rs against the ground truth. An S2R extracted by a tool was determined to be correct if all of the following conditions were satisfied: (1) the action type was an exact match with the action type from the ground truth; (2) the target widget, if it existed, included target widget words from the ground truth; (3) the input value, if it existed, was an exact match with that from the ground truth; and (4) the target direction, if it existed, was an exact match with that from the ground truth. As metrics for the accuracy of the three tools, we calculated precision by dividing the number of correctly extracted S2Rs by the total number of extracted S2Rs, and calculated recall by dividing the number of correctly extracted S2Rs by the total number of S2Rs in the ground truth.

Table 1: S2R Extraction Results

	S2Rs	Correct S2Rs	Precision	Recall
ReproBot	294	198	67%	77%
ReCDroid	220	116	53%	45%
Yakusu	260	147	57%	57%

4.3.2 RQ1 S2R Entity Extraction Results. Experiment 1's results are presented in Table 1. For each approach, the column "S2Rs" denotes the total number of S2Rs extracted by the given approach, and "Correct S2Rs" denotes the total number of correctly extracted S2Rs. Note that the total number of S2Rs in the ground truth was 256. The results show that ReproBot could identify more S2Rs and was more accurate, in terms of precision and recall, than both ReCDroid and Yakusu in doing so.

We also investigated the specific reasons for our tool's inaccurate S2Rs and found five categories of root causes: (1) The UI action word list used by ReproBot was not comprehensive. For example, because ReproBot was unable to find a semantically similar word in the word list for the click action, our approach misclassified the word "navigate" as an input action. Our approach was unable to accurately classify the UI action of a given predicate word in these cases. (2) The imprecision of the underlying NLP techniques was also found to be a cause of failure in our tool. Specifically, the underlying techniques occasionally failed to identify the syntactic constituents accurately, which caused our tool to misidentify the target widget. Additionally, the underlying tool sometimes failed to identify conjunction relations between clauses, which caused our tool to miss the S2Rs in those clauses. (3) A few S2Rs were found to have target widgets described via the predicate of the sentence. In these cases, our tool failed to extract the target widgets. An example of this is "I finish the dialog", where it expresses a step to press the "finish" button in the AUT. ReproBot was unable to extract the target widget in this case since it assumes the predicate is used for describing the UI action. (4) Bug reports were occasionally found to have utilized a special or unique way to express required input values to trigger a crash. Our tool relies on bug reports expressing

input values in the specific literal form for input, rather than using descriptive language, and, thus, failed to extract input values correctly when these situations occurred. For example, one bug report included "enter a number larger than Integer.MAX_VALUE". ReproBot did not identify the correct input value because it was not directly specified in the bug report. (5) A few bug reports contained text that either did not relate to or was not required to reproduce the described crash. In these situations, our tool often extracted extra S2Rs from such sentences, which were not intended to be a part of the desired reproduction steps. Our approach assumes that all text contained within a given bug report is intended to be included in the reproduction process, which can lead to failure if this was not the case. Our tool is unable to identify sentences that should not be considered a part of the reproduction process.

4.4 Experiment 2

4.4.1 Protocol. The second experiment evaluated the effectiveness (RQ2) and running time (RQ3) of our approach. To carry out this experiment, we ran ReproBot on the collected dataset, and then compared its results against those of ReCDroid and Yakusu. In addition to RQ2, we introduced two sub RQs that evaluated the contributions of the two parts of our approach independently. RQ2.1 evaluated the S2R extraction (Section 3.1). For this RQ, we created a variant of our approach, RB_A, that did not use the S2R extraction method described in Section 3.1, and instead, utilized the S2Rs extraction method from ReCDroid to drive the reproduction phase. RQ2.2 evaluated the S2R matching approach (Section 3.2). For this RQ, we created another variant of our approach, RB_B, which did utilize the S2Rs extracted by our approach, but used ReCDroid's exploration algorithm instead of the Q-Learning approach defined in Section 3.2. Note that we used ReCDroid's implementation for these variants since its implementation dependencies were more readily updated and its results in RQ2 showed it could be more easily adapted to run on newer and more modern subjects.

To compute effectiveness, we ran the three approaches and two variants on each of the bug reports in the dataset. Following previous related works [23, 43], each tool had a time limit of 3,600 seconds per bug report, and if the tool exceeded this time or threw an exception, then we considered the reproduction to have failed. For each successful reproduction reported by a tool, we manually replayed the identified event sequence to determine if it correctly reproduced the target failure. Since our Q-Learning based reproduction is non-deterministic, we ran ReproBot and RB_A three times on each bug report. If a tool succeeded at reproducing a given bug report at least once, then the reproduction of the said report was deemed a success. To address RQ3, we measured the running time of all the approaches on each bug report. For ReproBot, we took the average of the three runs.

Table 2: Bug Report Reproduction Results

	SR	MSR
ReproBot	57 (74%)	32 (64%)
ReCDroid	37 (48%)	21 (42%)
Yakusu	9 (12%)	3 (6%)

4.4.2 RQ2 Effectiveness. The reproduction results of each approach are presented in Table 2. The column "SR" denotes the number and percentage of our dataset's 77 bug reports that could be successfully reproduced by each approach, and the column "MSR" denotes the number and percentage of the subset of 50 bug reports with missing steps that could be successfully reproduced by each approach. As shown by the results, ReproBot was able to reproduce more bug reports overall and reproduce more among the subset of bug reports that had missing steps than ReCDroid and Yakusu. Among the bug reports successfully reproduced by ReproBot, 98% were reproduced more than once, which demonstrates the stability of the performance of our tool. Specifically, of the three runs of ReproBot, 1 bug report was successfully reproduced once, 9 twice, and 47 all three times. Taken together, these results demonstrate that ReproBot can reproduce Android bug reports more successfully than current state of the art approaches (i.e., ReCDroid and Yakusu).

One possibly confounding factor that we mention here is that Yakusu's implementation was tightly coupled with specific versions of Espresso and the Android OS. In turn many of our more modern apps required more recent versions of Android. While many of these apps could run with Yakusu, we confirmed with the authors of Yakusu that this issue could affect the generalizability of Yakusu on new apps.

We further analyzed the reasons for the 20 bug reports that our tool was unable to reproduce. Among them, eight reports could not be reproduced because the total number of missing steps in the bug report exceeded the limit of no-op actions set in ReproBot. For example, the reproduction of AnkiDroid-6432 [8] required 26 steps, 20 of which were missed in the bug report. Two reports, Transistor-149 [2] and Materialistic-1067 [4], could not be reproduced since the reproduction required pressing a button very quickly and the crash was non-deterministic. One report, FDroidClient-1821 [6] was failed to be reproduced because ReproBot failed to extract an S2R from the bug report due to the failure of the underlying tool. ReproBot failed to bridge the missed step, since the step is to click a specific button in a long list containing many other possible buttons. ReproBot failed to infer the correct button in this case. The remaining nine bug reports could not be reproduced because the S2R text description written in the bug report did not match with the reproducing UI events. This happens because reporters may use their own words when describing the steps and the widgets do not have a meaningful description. In either case, even though the S2R is given in the bug report, ReproBot could not find a match between the given S2R and the correct UI event.

We also analyzed the bug reports that our tool was not able to reproduce for all three runs. Specifically, we found one bug report that our approach was only reproduced once and none of ReCDroid and Yakusu could reproduce it. We found that the reason is that the provided S2Rs do not match with the UI events. Therefore, our approach spent a long time figuring out how to match these S2Rs and was only managed to reproduce it once. Among the eight bug reports that ReproBot successfully reproduced twice, the same reason was also found on four bug reports. For the other four bug reports, ReproBot generates a UI event sequence that triggers the same error message as the failure reported by the bug report but in a different way. We did not consider them as true

successful reproductions. The potential reason is that ReproBot stops exploring the AUT once it found the AUT crashes with the specified error message. Therefore, it may trigger the crash when performing random exploration.

4.4.3 RQ2.1&2.2 Contribution of Different Components. Our results showed that RB_A and RB_B both performed worse than ReproBot. RB_A only reproduced 45 bug reports and RB_B only reproduced 35 bug reports. Note that ReproBot was able to reproduce all bug reports that RB_A and RB_B were able to reproduce, with exception of one bug report that only RB_A was able to reproduce. The difference in reproduction results between ReproBot and the two variants showed the contribution of the combination of both the NLP techniques and the RL-based exploration. In comparison to ReCDroid, RB_A reproduced eight more bug reports in total (i.e., 45 to 37), and RB_B reproduced two less in total (i.e., 35 to 37). Specifically, RB_A succeeded on 14 bug reports where ReCDroid failed, and ReCDroid succeeded on 6 bug reports where RB_A failed. Moreover, RB_B succeeded on 1 bug report where ReCDroid failed, and ReCDroid succeeded on 3 bug reports where RB_B failed.

We further investigated RB_A's result to understand the reasons behind the changes in its performance compared with ReproBot. We observed that the reduced effectiveness of RB_A can be primarily attributed to the inaccurate S2Rs extracted by ReCDroid. For example, when ReCDroid misidentified an extra input S2Rs, our RL-based exploration method was not able to match it with any UI event. However, even with mistakes in identifying S2Rs, RB_A still outperformed ReCDroid by reproducing 8 more bug reports in total. This showed that, in many cases, our RL matching algorithm was able to make up for the inaccuracy in S2Rs.

We also explored the reasons why RB_B did not reproduce more bug reports and found the reasons to be twofold. First, ReCDroid's exploration algorithm was found to be less effective in handling low-quality steps and missing steps. When a report contained poorly-written or missing steps, the algorithm took a long time to discover the correct matching, which often caused a timeout. Second, ReCDroid's algorithm failed to leverage some useful information, such as *resource id* or *content description* for a widget from the VH, during the exploration. The algorithm, instead, only utilized the displayed text on a given UI element as a reference when matching it with an S2R, which was not enough. In conjunction, these limitations prohibited RB_B from effectively utilizing the S2Rs extracted by our approach.

4.4.4 RQ3 Running Time. The runtime distribution of each approach is shown in Figure 4. The result of each approach is visualized within a given row, with the minimum and maximum running time being 0 and 3,600 seconds, respectively. Each black line within an approach's visualization represents a running time of the approach for a variable amount of bug reports and the length each line is proportional to the frequency of the recorded running time in the dataset. Therefore, the shape of a given approach's visualization is determined by its running time distribution over all bug reports, where a visualization's width is larger in ranges with more recorded running times. On average, ReproBot spent 1,334 seconds (30 on S2R Extraction and 1,304 on S2R Matching) on each reproduction, which was the fastest among all three techniques. In contrast, ReCDroid spent 1,991 (4 on S2R Extraction and 1,987 on S2R Matching) and Yakusu spent 3,245 seconds (496 on S2R Extraction and 2,749 on S2R Matching), on reproduction. These results show that ReproBot was faster than ReCDroid and Yakusu with regards to reproduction runtime.

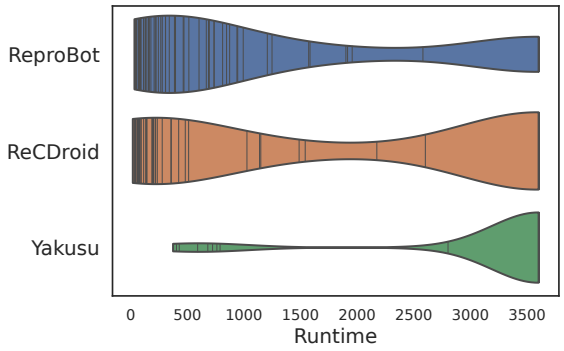


Figure 4: Runtime Distribution of the Three Approaches

4.5 Threats to Validity

4.5.1 External Validity. The primary threat to external validity is the representativeness of the bug reports in the evaluation dataset. We attempted to overcome this threat by using bug reports from related research works, which aided in avoiding biases that could potentially be introduced by our own search for reports. Note that similar to ReCDroid and Yakusu, our approach is unable to reproduce non-crash bug reports. Therefore, we did not collect non-crash bug reports in the evaluation dataset.

4.5.2 Internal Validity. One threat to the internal validity is the potential effect on the experimental results of the randomness of the RL algorithm used by ReproBot for S2R matching. To reduce this threat, we ran our tool and RB_A three times and reported the running results for each run. An additional threat is potentially incorrect ground truth for RQ1. This threat was mitigated by having several authors individually define the RQ1 ground truth and come to a consensus on the finalized version. Note that neither ReCDroid nor Yakusu carried out a similar evaluation, so this information was not provided in their online datasets. The last internal validity threat is regarding the issues we observed when running Yakusu on our dataset. These issues came from the limitations of Yakusu's implementation. As much as possible, we worked to set up compatible environments that could run the app and tool.

5 RELATED WORK

ReCDroid [43] and Yakusu [23] are the most closely related works. Their approaches rely on manually-crafted patterns to analyze bug reports and greedy-based exploration to discover the UI events for reproduction. In contrast, our approach incorporates a combination of advanced NLP techniques to analyze the bug report, which could extract S2Rs more accurately. Moreover, our approach leverages RL to guide the search for UI events, which is more effective at identifying the reproducing UI events. Section 2 discusses both in more detail, and we compare against both ReCDroid and Yakusu in the evaluation.

There are some research works focusing on studying and analyzing Android bug reports. Johnson et al. [25] conducted an empirical study on 180 Android bug reports to identify challenges to reproducing Android bug reports. Chaparro et al. [18] conducted an empirical study on how users report observed behavior; reproduction steps and expected behavior; and identified discourse patterns used by reporters. Based on these identified patterns, they designed an automatic tool to detect which information were missing in bug reports. However, none of them automatically extracted S2Rs from the bug report or reproduced the report. Chaparro et al. developed Euler [17], an automatic technique to assess the quality of S2Rs in Android bug reports. Euler resolves S2Rs from bug reports using simple grammar patterns. Different from Euler, our approach holistically analyzes the bug report's sentences to extract S2Rs, which is more broadly applicable.

Several previous research works have focused on augmenting Android bug reports or facilitating the reporting process. Liu et al. proposed a machine learning based classifier, Maca [27], which classifies action words of S2Rs into standard categories (click, input etc.) using both the textual information and the AUT information. Maca could potentially complement our technique by standardizing the action. However, Maca used simple grammar patterns to extract S2Rs, which limits its capability when it cannot accurately parse the S2Rs. Fusion, developed by Moran et al. [31], leveraged dynamic analysis to obtain UI events of the AUT and help create more actionable events in bug reports during the testing stage. Fazzini et al. proposed EBUG [22] to assist reporters to write more accurate reproduction steps by using information from static and dynamic analysis of the AUT to predict the next step. Yang et al. proposed an interactive bug reporting system, BURT [35], which provides a guided reporting of essential bug report elements (i.e., the observed behavior, expected behavior, and steps to reproduce the bug), instant feedback of problems with the elements, and graphical suggestions of the said elements. These three approaches mainly help improve the quality of the bug report at the moment when users write the report, but do not reproduce them. Our technique is complementary to these techniques by automatically reproducing the reports for developers.

Several works used RL for automatically testing Android apps [14, 32, 37] or web applications [44]. However, different from these works, we are the first work to adapt RL to the Android bug report reproduction domain.

6 CONCLUSION

In this paper, we proposed a novel approach to automatically reproduce Android bug reports. Our approach leverages advanced natural language processing techniques to holistically and accurately analyze a given bug report and adopts reinforcement learning to effectively reproduce it. The empirical evaluation shows that our approach is able to accurately extract reproduction steps from the bug report and effectively reproduce the bug report. Our approach outperformed state of the art techniques.

ACKNOWLEDGMENTS

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

REFERENCES

[1] 2005. Wikipedia Page of Passive Voice. https://en.wikipedia.org/wiki/English_passive_voice#Identifying_the_English_passive.
[2] 2017. Bug Report – Transistor 149. https://github.com/y20k/transistor/issues/149.
[3] 2017. OpenIE 5.1 Github repository. https://github.com/dair-iitd/OpenIE-standalone.
[4] 2018. Bug Report – Materialistic 1067. https://github.com/hidroh/materialistic/issues/1067.
[5] 2018. Yakusu Official Website. https://sites.google.com/view/yakusumobile/home.
[6] 2019. Bug Report – FDroidClient 1821. https://gitlab.com/fdroid/fdroidclient/issues/1821.
[7] 2019. ReCDroid Github repository. https://github.com/AndroidTestBugReport/ReCDroid.
[8] 2020. Bug Report – AnkiDroid 6432. https://github.com/ankidroid/Anki-Android/issues/6432.
[9] 2023. Android Accessibility. https://support.google.com/accessibility/android/answer/7158690?hl=en.
[10] 2023. Github Issue Tracker. https://github.com/issues.
[11] 2023. Google Code Issue Tracker. https://code.google.com/archive/.
[12] 2023. ReproBot Website. https://sites.google.com/usc.edu/reprobot/home.
[13] 2023. UI Automator. https://developer.android.com/training/testing/ui-automator.
[14] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement Learning for Android GUI Testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, Lake Buena Vista FL USA, 2–8. https://doi.org/10.1145/3278186.3278187
[15] Nikita Bhutani, H V Jagadish, and Dragomir Radev. 2016. Nested Propositions in Open Information Extraction. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Austin, Texas, 55–64. https://doi.org/10.18653/v1/D16-1006
[16] Matthias Cetto, Christina Niklaus, André Freitas, and Siegfried Handschuh. 2018. Graphene: Semantically-Linked Propositions in Open Information Extraction. In *Proceedings of the 27th International Conference on Computational Linguistics*. Association for Computational Linguistics, Santa Fe, New Mexico, USA, 2300–2311.
[17] 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
[18] 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
[19] Danqi Chen and Christopher Manning. 2014. A Fast and Accurate Dependency Parser Using Neural Networks. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics, Doha, Qatar, 740–750. https://doi.org/10.3115/v1/D14-1082
[20] Luciano Del Corro and Rainer Gemulla. 2013. ClausIE: Clause-Based Open Information Extraction. In *Proceedings of the 22nd International Conference on World Wide Web - WWW '13*. ACM Press, Rio de Janeiro, Brazil, 355–366. https://doi.org/10.1145/2488388.2488420
[21] Oren Etzioni, Michele Banko, Stephen Soderland, and Daniel S. Weld. 2008. Open Information Extraction from the Web. In *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, Vol. 51. Morgan Kaufmann Publishers Inc., Hyderabad, India, 2670–2676. https://doi.org/10.5555/1625275.1625705
[22] 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
[23] 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
[24] Matthew Honnibal, Ines Montani, Sofie Van Landeghem, and Adriane Boyd. 2020. spaCy: Industrial-strength Natural Language Processing in Python. (2020). https://doi.org/10.5281/zenodo.1212303
[25] 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

[26] Daniel Jurafsky and James H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition* (3rd ed.). Vol. 13. Prentice Hall PTR, USA.

[27] 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

[28] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank. *Comput. Linguist.* 19, 2 (jun 1993), 313–330.

[29] 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.

[30] Eleni Miltsakaki, Rashmi Prasad, Aravind Joshi, and Bonnie Webber. 2004. The Penn Discourse Treebank. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC'04)*. European Language Resources Association (ELRA), 4.

[31] 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

[32] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement Learning Based Curiosity-Driven Testing of Android Applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 153–164. https://doi.org/10.1145/3395363.3397354

[33] Rashmi Prasad, Nikhil Dinesh, Alan Lee, Eleni Miltsakaki, Livio Robaldo, Aravind Joshi, and Bonnie Webber. 2008. The Penn Discourse TreeBank 2.0.. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Vol. Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08). European Language Resources Association (ELRA), Marrakech, Morocco, 8.

[34] Michael Schmitz, Robert Bart, Stephen Soderland, and Oren Etzioni. 2012. Open Language Learning for Information Extraction. In *Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning*. Association for Computational Linguistics, Jeju Island, Korea, 523–534.

[35] 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.

[36] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.

[37] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A Reinforcement Learning Based Approach to Automated Testing of Android Applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*. ACM, Lake Buena Vista FL USA, 31–37. https://doi.org/10.1145/3278186.3278191

[38] Christopher JCH Watkins and Peter Dayan. 1992. Q-learning. In *Machine learning*. 279–292. https://aclanthology.org/P13-1045

[39] 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

[40] Fei Wu and Daniel S Weld. 2010. Open Information Extraction Using Wikipedia. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Uppsala, Sweden, 118–127.

[41] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically Extracting Bug Reproducing Steps from Android Bug Reports. In *Reuse in the Big Data Era*, Xin Peng, Apostolos Ampatzoglou, and Tanmay Bhowmik (Eds.), Vol. 11602. Springer International Publishing, Cham, 100–111. https://doi.org/10.1007/978-3-030-22888-0_8

[42] 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

[43] 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

[44] Yan Zheng, Yi Liu, Xiaofei Xie, Yepang Liu, Lei Ma, Jianye Hao, and Yang Liu. 2021. Automatic Web Testing Using Curiosity-Driven Reinforcement Learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 423–435. https://doi.org/10.1109/ICSE43902.2021.00048

[45] 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 2023-02-16; accepted 2023-05-03