Toward Interactive Bug Reporting for (Android App) End-Users

Yang Song
College of William & Mary
USA

Junayed Mahmud
George Mason University
USA

Ying Zhou
University of Texas at Dallas
USA

Oscar Chaparro
College of William & Mary
USA

Kevin Moran
George Mason University
USA

Andrian Marcus
University of Texas at Dallas
USA

Denys Poshyvanyk
College of William & Mary
USA

ABSTRACT

Many software bugs are reported manually, particularly bugs that manifest themselves visually in the user interface. End-users typically report these bugs via app reviewing websites, issue trackers, or in-app built-in bug reporting tools, if available. While these systems have various features that facilitate bug reporting (*e.g.*, textual templates or forms), they often provide *limited* guidance, concrete feedback, or quality verification to end-users, who are often inexperienced at reporting bugs and submit low-quality bug reports that lead to excessive developer effort in bug report management tasks.

We propose an interactive bug reporting system for end-users (Burt), implemented as a task-oriented chatbot. Unlike existing bug reporting systems, Burt provides guided reporting of essential bug report elements (*i.e.*, the observed behavior, expected behavior, and steps to reproduce the bug), instant quality verification, and graphical suggestions for these elements. We implemented a version of Burt for Android and conducted an empirical evaluation study with end-users, who reported 12 bugs from six Android apps studied in prior work. The reporters found that Burt's guidance and automated suggestions/clarifications are useful and Burt is easy to use. We found that Burt reports contain higher-quality information than reports collected via a template-based bug reporting system. Improvements to Burt, informed by the reporters, include support for various wordings to describe bug report elements and improved quality verification. Our work marks an important paradigm shift from static to interactive bug reporting for end-users.

CCS CONCEPTS

• **Software and its engineering → Software maintenance tools**.

KEYWORDS

Bug Reporting, Task-Oriented Chatbots, Android Apps

ESEC/FSE '22, November 14–18, 2022, Singapore, Singapore

ACM ISBN 978-1-4503-9413-0/22/11.
https://doi.org/10.1145/3540250.3549131

ACM Reference Format:
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 *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '22), November 14–18, 2022, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3540250.3549131

1 INTRODUCTION

Bug report management is an important and costly software engineering activity. While certain types of bugs can be reported automatically via a known oracle (*e.g.*, crashes), recent studies have illustrated that more than half of the bugs reported in open source software relate to functional problems with no automatically identifiable oracle [59] and, hence, must be reported manually. High-quality bug reports are essential for bug triage and resolution and they are expected to describe *at minimum* the observed (incorrect) behavior (**OB**), the steps to reproduce the bug (**S2Rs**), and the expected (correct) software behavior (**EB**) [25, 48, 65].

One of the main difficulties that contributes to quality issues in end-user bug reporting is the *knowledge gap* between end-users and developers [44, 53]. That is, there is often a gap between what end-users *know* and what developers *need* [65], generally due to the fact that users are both unfamiliar with the internals of the software and with the explicit types of information that are important for developers (*e.g.*, the OB, EB, and S2Rs).

Most current reporting systems are not designed to address the above-mentioned knowledge gap between end-users and developers. In particular, current systems are typically lacking along two important dimensions: (1) they offer *limited guidance* related to *what* needs to be reported and *how* it needs to be reported; and (2) no *feedback* is offered to reporters on whether the information they provided is correct or complete. In consequence, given the *static nature* of these bug reporting interfaces, the burden of providing high-quality information rests on the reporters.

We posit that an *interactive* reporting solution can help to bridge the developer–end-user knowledge gap. Inspired by prior work on question/answering systems for debugging [47], we argue that a conversational agent (*i.e.*, a chatbot) can successfully guide end-users through the reporting process, while offering interactive suggestions and instant quality verification.

In this paper, we introduce and evaluate a task-oriented dialogue system for **BU**g Repo**RT**ing (or Burt) that is capable of providing instant feedback for each element of a bug description (*i.e.*, OB, EB, and S2Rs), while actively guiding corrections, where needed. Burt combines novel and state-of-the-art techniques for dynamic software analysis, natural language processing, and automated report quality assessment. We designed and developed the current version of Burt to work for Android apps, but its architecture is platform-agnostic and it can be instantiated, with some engineering effort, for other types of GUI-based applications (*e.g.*, web-based, desktop, or iOS-based). In particular, Burt constructs a graph of program states using both crowdsourced app usage data and automated GUI-based exploration techniques. The chatbot then parses and interprets end-user descriptions of various bug report elements by matching them to states and transitions in the constructed graph, and produces graphical suggestions regarding information that is likely to be reported (*e.g.*, the next S2Rs). Additionally, Burt recognizes when end-users provide incomplete or ambiguous information and suggests improvements or clarifications to the users. Traditional task-oriented chatbots typically have direct access to a structured and easily parseable knowledge-base [11]. In contrast, Burt is more complex, as it reconciles high-level descriptions provided by end users and matches these to technical program information, bridging the end-user to developer knowledge gap.

We evaluated Burt empirically, asking 18 end-users, with various levels of prior bug reporting experience, to report 12 bugs from six Android apps using a prototype implementation of Burt. We found that the guidance and automated suggestions/clarifications made by the chatbot were accurate, useful, and easy to use, and the collected bug reports are high-quality. We asked 18 additional end-users to report the same bugs with a template-based bug reporting system (Itrac) and compared the quality of these reports to those reported with Burt. Burt reports have fewer incorrect and missing S2Rs than the Itrac reports. We also found that Burt helps novice bug reporters provide more correct steps, and experienced reporters avoid missing steps.

In summary, the contributions of this paper are as follows:

- Burt, the first task-oriented, conversational agent that supports end-users in reporting bugs (currently for Android apps), with features such as automated suggestions, real-time feedback, prompts for information clarification, and graphical cues.
- The results of an empirical evaluation involving 36 end-users that investigates user experiences, preferences, and attributes of interactive bug reporting with Burt, as well as the quality of the resulting bug reports.
- A replication package [21] that contains a complete implementation of Burt, Burt's app usage/execution data, code and data about Burt's evaluation, and documentation that enables the verification and validation of our work and future research in the topic of bug reporting systems. The package also includes a video illustrating Burt.

Our work opens the door to a new way of thinking about end-user bug reporting, using conversational agents, shifting the state of the art from *static* to *interactive* bug reporting. While Burt is a prototype, we expect that it will serve as the foundation for a new class of interactive bug reporting systems, combining elements of existing static systems with features of conversational agents [36].

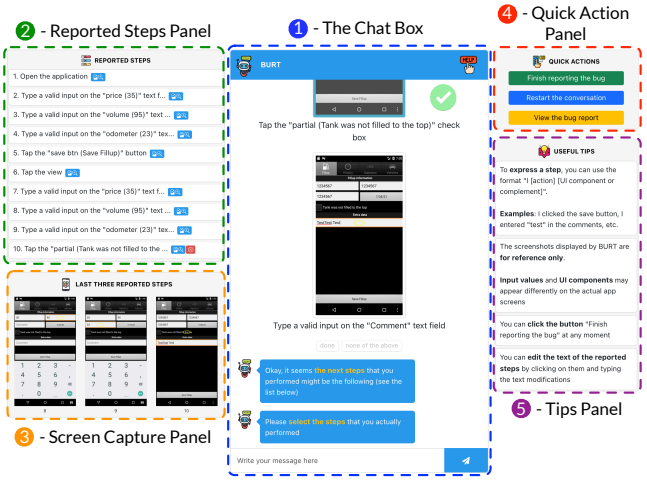

Figure 1: Burt's graphical user interface

2 BURT: A CHATBOT FOR BUG REPORTING

We propose a task-oriented chatbot for **BU**g Repo**RT**ing (Burt). Burt offers a variety of features for interactive bug reporting such as the ability to (i) guide the user in reporting essential bug report elements, (ii) check the quality of these elements, (iii) offer instant feedback about issues, and (iv) provide graphical suggestions.

Burt is designed to collect three key elements for developers during bug triage and resolution [48, 57, 65]: the *observed behavior* (**OB**), the *expected behavior* (**EB**), and the *steps to reproduce* the bug (**S2Rs**). Burt collects these from the user through a dialogue and generates a web-based bug report containing textual descriptions for these elements with attached screen captures of the system.

Burt's design consists of three main components, inspired by the typical architecture of task-oriented dialogue systems [36], which adapt techniques from automated program analysis and natural language processing to facilitate bug reporting. Burt's **Natural Language Parser (NL)** parses the relevant information from end-user responses to the chatbot. The **Dialogue Manager (DM)** dictates the structured conversation flow for Burt's reporting process and handles the presentation of multi-modal (*e.g.*, screenshots and text) information to the user. Finally, the **Report Processing Engine (RP)** maps information parsed from user responses to various states in a program execution model for a given app in order to assess bug element quality. The current version of Burt is designed for Android apps and builds its execution model using a combination of automated app exploration and crowdsourced user traces. In this section, we present Burt's components in detail.

2.1 Graphical User Interface (GUI)

We designed Burt as a web-based application that includes both a standard chatbot interface along with additional visual components as illustrated in Fig. 1. The *Chat Box* 1 allows the end-user to provide textual descriptions of the OB, EB, and S2Rs as well as interact with the graphical information that Burt displays (*e.g.*, recommendations of the next S2Rs via screenshots). The *Reported Steps Panel* 2 enumerates and displays the S2Rs that the user has reported. The textual description of the reported steps can be edited and the last reported step can be deleted, if the user makes a mistake

and wishes to correct it. The *Screen Capture Panel* ❸ displays screen captures of the last three S2Rs. The *Quick Action Panel* ❹ provides buttons to finish reporting the bug, restart the bug reporting session, and (pre)view the bug report being created – these can be activated anytime. The *Tips Panel* ❺ displays recommendations to end-users on how to use Burt and how to better express the OB, EB, and S2Rs. The tips change depending on the current stage of the conversation.

2.2 Natural Language Parser (NL)

Burt parses the OB, EB, and S2R descriptions provided by end-users using dependency parsing via the Stanford CoreNLP toolkit [50]. This process obtains the tree of grammatical dependencies [17] between words in a sentence and extracts the relevant words from the tree. This parsing technique is needed by the Report Processing Engine to assess the quality of parsed bug report elements and to help direct the flow of conversation (see Sec. 2.4.2).

Burt first utilizes the heuristic-based approach introduced by Chaparro *et al.* [29] to identify the type of a sentence (*e.g.*, conditional, imperative, or passive voice) for each message received from the user. This approach implements heuristics (based on dependency parsing and part-of-speech tagging [50]) to identify discourse patterns in OB, EB, and S2R descriptions [29]. Once the sentence type is identified, Burt executes a series of algorithms to extract the relevant words from the sentence, based on prior work on quality assessment of S2Rs [28]. In essence, we implemented 16 parsing algorithms that traverse the grammatical trees [17] of end-user sentences which have a different structure depending on the sentence type (*e.g.*, conditional or imperative). Each algorithm parses sentences of one type. All the 16 algorithms implemented for the different types of OB/EB/S2R sentences can be found in our online replication package [21].

Burt parses a single sentence using the following format:
`[subject] [action] [object] [preposition] [object2]`
where the `subject` is the actor (*e.g.*, the user or an app component) performing the `action`, which is an operation or task (*e.g.*, tap, create, crash); the `object` is an "entity" directly affected by the `action`, and `object2` is another "entity" related to the object by the `preposition`. An "entity" is a noun phrase that may represent numeric/textual app input, domain concepts, GUI components, *etc.* Depending on the sentence, its type, and whether it describes an OB, EB, or S2R, the words (*e.g.*, the `subject`, `preposition` or `object2`) extracted from the entity are required or optional.

For example, for the Mileage Android app [12], the OB sentence *"The average fuel economy shows a NaN value"*, written in present tense, is parsed as `[average fuel economy] [shows] [NaN value]`. The EB sentence *"fuel economy statistics should be calculated correctly"*, which uses the modal "should", is parsed as `[average fuel economy] [is] [calculated]`. The S2R sentence *"Save the car fillup"*, written imperatively, is parsed as `[user] [saves] [car fillup]`.

Some sentences describe a combination of OB, EB and S2Rs in a single phrase. For example, the sentence *"The app stopped when I added a new time range"* describes both an OB and a S2R. This sentence is parsed by Burt as `[app] [stopped]` as the OB, and `[add] [new time range]` as the S2R. In this example, Burt extracts the S2R from the sentence as follows. First, it locates the adverb *"when"* in the parsed grammatical tree, then it follows the relationship

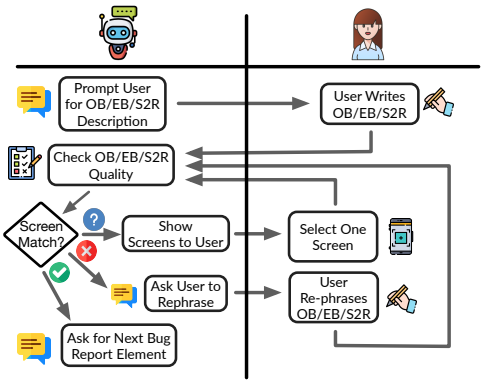

Figure 2: Burt's dialogue flow for quality checking

that leads to the verb *"add"* for which *"when"* is the adverbial modifier. Next, Burt locates the verb's nominal subject "I" and its direct object *"time range"*. If these relationships do not exist in the tree, the sentence is not conditional, as expected. Otherwise, Burt extracts the verb *"add"* as the `action` and the noun phrase *"time range"* as the `object`. In the end, this sentence is parsed as the S2R: `[add] [new time range]`.

When multiple sentences compose a single user message, Burt only parses the initial sentence. When Burt is unable to parse a user message (*e.g.*, because it cannot identify the `subject`), it asks the user to rephrase the sentence. Burt's *Tips Panel* ❺ and user guide suggests patterns to the user to phrase the OB, EB, and S2Rs.

2.3 Dialogue Manager (DM)

Burt's dialogue flow consists of three main phases: OB, EB, and S2R collection. Burt's dialogue is multi-modal in nature, and is capable of suggesting both natural language and graphical elements, such as screenshots, to help guide the user through the reporting process. The DM relies upon the RP engine to assess the quality of bug elements reported by end users (see Sec. 2.4.2). While Burt's dialogue flow proceeds linearly to capture each bug element (the OB, EB, and S2Rs, in that order), the dialogue flow is similar for all elements. There are two main dialogue flows that Burt navigates: (i) performing quality checks on written bug report elements (applies to all bug elements), and (ii) automated suggestion of S2Rs (for S2Rs only). Next we describe these two main dialogue flows.

2.3.1 Dialogue Flow for Bug Element Quality Checks (OB/EB/S2R). Before the dialogue begins, a user must select the target app by clicking on its icon. Then, Burt's dialogue flow for quality checking, illustrated in a modified swimlane diagram in Figure 2, is initiated, starting with the OB. To begin the quality checking process, Burt prompts the user to provide the bug element (OB/EB/S2R). Burt automatically parses the description of the element and the RP engine verifies its quality (see Sec. 2.4.2).

If the OB/EB/S2R is matched to an app screen from Burt's execution model (see Sec. 2.4.1), Burt asks the user for confirmation of the matched screen. If the user confirms, Burt proceeds to the next phase of the conversation (*e.g.*, asking for the EB or next S2Rs), otherwise, Burt asks the user to rephrase the bug element.

If there are no app screen matches, Burt informs the user about the issue and asks her to rephrase the OB/EB/S2R. Once the user

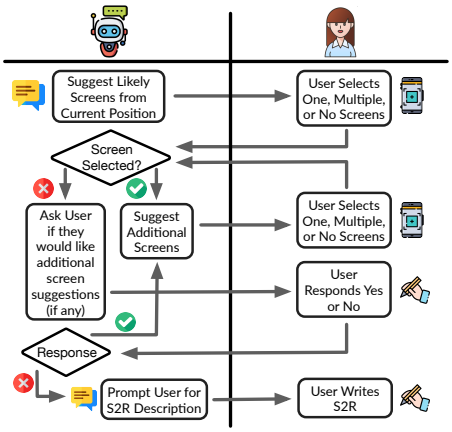

Figure 3: Dialogue Flow for S2R Predictions

provides a new description, the quality verification procedure is re-executed. If there are *multiple* matches, Burt provides a list of up to five app screenshots (derived from the app execution model) that match the description. The user can then inspect the app screens and select the one that she believes best matches her description of the bug element. If none are selected, Burt suggests additional app screens if any. If the user selects one app screen, Burt saves the bug element description and screen, and proceeds to collecting the next bug element. After three unsuccessful attempts to provide a high quality OB description, Burt records the (last) provided OB description for bug report generation. This process proceeds for each bug element starting with the OB. S2Rs are treated slightly differently since Burt can also *predict* S2Rs as we describe next.

2.3.2 Dialogue Flow for Suggesting S2Rs. Burt suggests potential next S2Rs that the user may have performed during actual app usage, depending on the last reported step and the user-selected screen that is having the problem, *i.e.*, the OB screen. Figure 3 illustrates this process. This dialogue flow uses a predictive algorithm that uses Burt's execution model (see Sec. 2.4.3). The suggestions are displayed as a list of app screens, each screen representing a S2R. Each S2R in the list displays the screen capture with a textual description placed below the image. The screen capture is visually annotated with a yellow oval highlighting the GUI component (*e.g.*, a button) executed by the step. The user can select none, one, or multiple of the suggested S2Rs. When a S2R is selected, Burt suggests additional S2Rs if any. When none are selected and Burt has more suggestions, Burt asks the user if she wants to get more suggestions. If so, Burt displays them. Otherwise, Burt prompts the user to describe the next S2R.

2.3.3 Collecting Input Values. User input from type-like steps (*e.g.*, "*I entered 5 gallons*") are extracted by Burt from the `object` or `object2` of the parsed S2Rs, by identifying literal values or quoted text. If the input value is missing or generic (*i.e.*, not a literal or "text"), Burt prompts the user to provide the input. This is only activated if the matched S2R is confirmed by the user as a correct S2R.

2.4 Report Processing Engine (RP)

Burt's RP Engine is composed of three sub-components: (i) the *App Execution Model*, (ii) the *Dialogue Quality Processor* which maps parsed bug report elements to app states from the model, and (iii) the *S2R Response Predictor* which infers likely next S2Rs, given an existing set of S2Rs already mapped to the execution model.

2.4.1 App Execution Model. The app execution model is a graph that stores sequential GUI-level app interactions (*e.g.*, taps, types, or swipes performed on screen GUI components) and the app response to those interactions (*i.e.*, app screens). These interactions and app responses are produced using two strategies: (1) by executing an automated systematic app exploration adapted from CrashScope's *GUI-ripping Engine* [52, 54], and (2) by recording (crowdsourced) app usage information from app end-users or developers. Both the systematic app exploration and app usage data are collected *before* Burt is deployed for use.

App Execution Model Data Collection. This is Burt's platform-specific part and would be constructed differently for non-Android applications. Burt uses a version of CrashScope's *GUI-ripping engine* [52, 54] to generate app execution data in the form of sequential interactions. CrashScope enables dynamic analysis of Android apps that utilizes a set of systematic exploration strategies (*e.g.*, top-down and bottom-up) and has been shown to exhibit comparable coverage to other automated mobile testing techniques [52]. For a detailed description of the engine, we refer the readers to Moran *et al.*'s previous work [52, 54]. As in prior work [23, 24, 31, 37, 40], we instantiate data collection by recording low-level app event traces using the `getevent`, `sendevent`, `uiautomator` utilities included in the Android OS and SDK.

Collecting crowdsourced user app usage data serves two main purposes: (1) increase the coverage of app states and screens in Burt's execution model; and (2) augment the model with scenarios that are common during normal app usage. Section 2.5 describes the procedure that we implemented to collect the crowdsourced data. Crowdsourced data collection leads to the same types of app events as the automatic app exploration does.

App Execution Model Structure. The execution model is a directed weighted graph $G = (V, E)$, where V is the set of unique *app screens* with complete GUI hierarchies [2], and E is a set of *app interactions* performed on the screens' GUI components. In this model, two screens with the same number, type, size, and hierarchical structure of GUI components are considered a single vertex. E is a set of unique tuples of the form (v_x, v_y, e, c), where e is an application event (tap, type, swipe, *etc.*) performed on a GUI component c from screen v_x, and v_y is the resulting screen right after the interaction execution. Each edge stores additional information about the interaction, such as the textual data input (only for *type* events) and the interaction execution order dictated by the app usage (manual or automatic). The graph's starting node has only one outgoing interaction, which corresponds to the application launch. A GUI component is uniquely represented by a type (*e.g.*, a button or a text field), an identifier, a label (*'OK'* or *'Cancel'*), and its size/position in the screen. Additional information about a component is stored in the graph, for example, the component description given by the developer, the parent/children components, and an annotated screen capture of the app highlighting the GUI component being interacted with. The screen captures are used in the screen suggestions made by Burt (see Sec. 2.4.3).

The graph edges have a weight which indicates the likelihood of a given app interaction represented as a state transition. The weights are utilized by the *S2R Response Predictor* (see Sec. 2.4.3), which aims to suggest S2Rs that end-users would perform when normally using given app features. To enable accurate predictions, Burt assigns higher weights to interactions executed by humans than those executed automatically by CrashScope. To accomplish this, Burt sets the weight of an edge to the number of times it was executed in the collected usage data. If an edge is not executed by a human, but was executed by CrashScope's systematic exploration, then edge weight is set to one, even if CrashScope executes the same interaction multiple times. While this weight assignment scheme is straightforward, it proved to be effective (see Sec. 4).

2.4.2 Dialogue Quality Processor. Based on prior work [28], Burt's quality definition is based on the ability to match a textual bug description (OB, EB, or S2R) to the screens (states) and interactions (edges) of the execution model. A textual description is considered to be high-quality if it can be precisely matched to the execution model, otherwise it is deemed low-quality. This definition and Burt's dialogue features that prompt users to improve low-quality descriptions aim to reduce the knowledge gap between the reporters, who are unfamiliar with app internals and may not know how to express a bug, and developers, who define and implement the vocabulary captured in Burt's execution model.

Assessing OB Quality. Burt first builds a query to the app execution model by concatenating the non-empty elements from the parsed description, namely the `subject`, `action`, `object`, and `object2`. Then, it preprocesses the query using lemmatization [50] and attempts to retrieve all matching GUI components via an adapted version of the matching procedure proposed by prior work [28]. This procedure computes the similarity score between the query and the elements from a GUI component, namely the component label, the description, and the ID specified by the original developer. The similarity is computed based on a normalized length of the longest common substring between query and the component elements. If such similarity is greater than or equal to 0.5, then there is a match, otherwise there is a mismatch. If the initial query does not match an app screen, Burt runs a different query by using only the `subject`, since, based on our experience, it may indicate a key GUI component that is directly related with a bug.

Burt keeps a list of the app screens with at least one matching GUI component. Such a list is sorted in increasing order by the distance between the starting state in the execution model and the matched state. If this list is empty, it means the OB description does not use vocabulary from the app screens and needs to be rephrased. If this list contains only one element, it is used to show the user the potential buggy app screen, which the user has to confirm as correct or incorrect. Otherwise, if the list contains multiple elements, it is used to display the possible buggy app screens so that the user decides the appropriate screen. The selected OB screen by the user is tracked in the execution model and is used for (1) EB description matching, (2) the prediction of the next S2Rs, and (3) asking the user if the last provided S2R is the last step to replicate the bug.

Assessing EB Quality. Burt performs the matching approach described above using the parsed EB description against the OB screen confirmed by the user. Burt assumes the OB screen is the one that should work correctly, therefore, it attempts to match the EB description to it. If the user did not select an OB screen, the EB matching is bypassed and the EB description is saved for generating the bug report. If the EB description does not match the OB screen, it means the vocabulary used in the EB description is different from the OB screen, and the EB description should be rephrased. However, rather than prompting the user to rephrase it, Burt asks the user if the OB screen is the one that should work correctly.

Assessing S2R Quality. Burt adapts the step resolution/matching algorithm proposed by Chaparro *et al.* [28] and performs exploration of the execution model driven by the matching of the reported S2Rs. By default, Burt assumes the first S2R performed by a user is launching the app and the current graph state is set to be the first app screen that results from this operation.

For a provided S2R description, starting from the current state, Burt traverses the graph in a depth-first manner and performs step resolution on each state. Step resolution is the process of determining the most likely app interactions that the S2R refers to in a particular state (*i.e.*, app screen). The result is a set of *resolved interactions* for the S2R on the selected states. If the S2R resolution fails for these states (either with a mismatch or a multiple-match result), then it means that either: (1) there are app states not present in the execution model, or (2) the S2R description is of low-quality.

The *resolved interactions* are matched against the interactions (*i.e.*, the edges) from the graph, by matching their source state v_x, the event e, and the component c. If a pair of interactions match, then they are considered to be the same interaction. The matching returns a set of interactions from the graph that match the resolved ones. If this set is empty, it means that the resolved interactions were not covered by the app exploration and the quality assessment returns a low-quality result with a mismatch. If the reason for the mismatch is because of multiple-component or -event match (*i.e.*, the S2R description matches multiple GUI components or map to multiple events), Burt considers the S2R as ambiguous, and Burt indicates that the S2R's `action` corresponds to multiple events, or the `object` or `object2` match multiple GUI components. If there is a no-match, Burt specifies the problematic vocabulary from the S2R elements: `action`, `object`, `object2`, or any combination of these.

Otherwise, if the set of *resolved interactions* is not empty, Burt proceeds with selecting the most relevant interaction that corresponds to the S2R description, by selecting the one whose source state is the nearest to the current execution state in the graph.

2.4.3 S2R Response Predictor. Burt predicts the next S2Rs that a user may have performed in practice. The prediction is executed during the following dialogue scenarios (see Fig. 3): (1) when an OB screen from the execution model has been selected/confirmed by the user, (2) when the S2R collection phase starts, (3) right after the user confirms a matched S2R for her S2R description, or (4) when the user has already selected one or more S2Rs suggestions.

Burt implements a shortest-path approach to predict the next S2Rs. First, Burt determines the paths between the current graph state and the corresponding OB state. Then, Burt computes the likelihood score based on the execution model edge weights.

Burt uses the equation below to compute the score S_p of an n-edge path $p = \{w_1, w_2, ..., w_n\}$, with w_k being edge k's weight:

$$S_p = \frac{1}{n}\sum_k w_k + \frac{1}{n}$$

The first term in the sum is the average weight among all path edges and the second term is a factor that favors shorter paths.

Once the paths are ranked by their scores (in descending order), these are modified to include loops, *i.e.*, steps that lead to the same app screen (*e.g.*, *types* for providing input values). Then, only the first five steps for each path are selected. With only the first five steps, all unique paths are kept and only the top-2 paths are saved for being presented to the user. The first one is always presented and if the user does not select any of the steps as being the next S2Rs and wants more suggestions, the second path is presented next. Every time the user selects a suggested step as being the next step, the prediction/suggestion process restarts with new predictions.

2.5 BURT Implementation

BURT is currently implemented as a web application with two major components: the front-end, developed with the React Chatbot Kit [14], and the back-end, developed with Spring Boot [16]. BURT's implementation is tailored for Android applications, however, its underlying techniques are generic enough to be easily implemented for other types of software — the App Execution Model Data Collection is the only platform-specific part.

To collect the crowdsourced app usage traces for BURT, two computer science students, who did not have knowledge of our studied bugs, were instructed to use the apps' features as they typically would do, and recorded traces that exercise key app features. Additionally, two of the paper authors recorded sequences simulating app developers who test the apps. These traces were converted and merged into app execution models for each of the studied apps as described in Sec. 2.4.1. In practice, developers can utilize recorded tests, crowdsourced data, or automated app exploration techniques with a "one-time" cost for building the app execution model.

3 EMPIRICAL EVALUATION DESIGN

We conducted two user studies to evaluate: (1) BURT's perceived usefulness and usability; (2) BURT's intrinsic accuracy in performing bug report element quality verification and prediction; and (3) the quality of the bug reports collected with BURT, compared with reports collected by a template-based bug reporting system. We aim to answer the following research questions (RQs):

RQ$_1$: *What* BURT *features do reporters perceive as (not) useful?*

RQ$_2$: *What* BURT *features do reporters perceive as (not) easy to use?*

RQ$_3$: *What is the accuracy of* BURT *in performing bug element quality verification and prediction during the bug reporting process?*

RQ$_4$: *What is the quality of the bug reports collected by* BURT *compared to reports collected by a template-based bug reporting system?*

To answer the RQs, we selected a set of Android app bugs used in prior research (Sec. 3.1), and asked bug reporters to report these bugs using BURT and to evaluate their experience (Sec. 3.2). We analyzed the conversations the reporters had with BURT and measured how accurate BURT was during the reporting process (Sec. 3.3). Then, we asked additional participants to report the same bugs with a template-based bug reporting system (Secs. 3.4.1 and 3.4.2), and analyzed the collected bug reports to measure their quality based on bug element correctness (Sec. 3.4.3). We present and discuss the results in Sec. 4. Our user studies were approved by an Institutional Review Board (IRB) and conducted remotely due to restrictions related to COVID-19.

Table 1: Apps and bug dataset

App	Bug ID	# of S2Rs	Bug type
APOD	CC3	11	Incorrect color in GUI component
	RB	5	Error message on screen
DROID	CC5	7	Crash
	CC6	12	Crash
GNU	CC9	13	Duplicated GUI component
	RC	5	Crash
GROW	CC5	10	Crash
	RC	8	Crash
TIME	CC1	16	GUI component disappears
	CC4	9	Crash
TOK	CC2	10	Crash
	CC7	6	GUI component does not appear

3.1 Apps and Bug Dataset

We selected 12 Android app bugs from the bug dataset provided by Cooper *et al.* [31]. The apps in the dataset support different app domains and have been studied in prior research [24, 28, 52, 53]. The apps are: AntennaPod (APOD) [3] – a podcast manager, Time Tracker (TIME) [18] – a time-tracking app, Android Token (TOK) [1] – a one-time-password generation app, GnuCash (GNU) [8] – a personal finances manager, GrowTracker (GROW) [9] – a plant monitoring app, and Droid Weight (DROID) [6] – a personal weight tracking app. This dataset provides, for each bug, the APK installer that contains the bug, the description of the incorrect (observed) app behavior (OB), the expected app behavior (EB), and the (minimal) list of the steps to reproduce the bug (S2Rs).

From the 60 bugs (35 crashes and 25 non-crashes) in Cooper *et al.*'s dataset [31], we selected 12 bugs (7 crashes, 1 handled error, and 4 non-crashes) using a stratified random approach (see Table 1). We randomly selected two bugs for each of the six apps, ensuring that the bugs represent a variety of bug types that manifest visually on the device (crashes, GUI issues, functional bugs, *etc.*) and have a diverse number and type of S2Rs (taps, types, swipes, *etc.*). Six bugs contain $5-9$ (minimal) S2Rs, and six bugs contain $10-16$ (minimal) S2Rs (see Table 1). The 12 bugs are reproducible on a specific web-based Android emulator configuration (virtual Nexus 5X with Android 7.0 configured via the Appetize.io [5] service).

3.2 RQ$_1$ & RQ$_2$: BURT's User Experience

We asked participants to report a selected subset of bugs using BURT, and evaluate their experience via an online questionnaire.

3.2.1 BURT Bug Reporter Recruitment. We reached out to 36 potential participants with mixed experience in bug reporting from our personal network, who were not involved in or aware of the purpose of this work. They were offered a $15 USD gift card for participation. From these, 24 users completed the study and data from six participants was discarded due to low-effort answers, thus resulting in valid responses from 18 participants. Four of the six participants did not treat the study seriously, that is, they submitted incomplete reports (e.g., only the OB was reported) and answered all survey questions with the same response. The remaining two

Table 2: Questionnaire for evaluating BURT's user experience

ID	Question
Q1	How often were BURT's screen suggestions useful?
Q2	How often was BURT able to understand your OB/EB/S2Rs?
Q3	How often were you able to understand BURT's messages/questions?
Q4	Was BURT's panel of reported steps useful?
Q5	How easy to use was BURT overall?
Q6	Which of BURT's features did you find easy/difficult to use?
Q7	What additional functionality (if any) would you like to see in BURT?

participants reported completely different bugs to the ones assigned. Five participants had not reported a software bug before, nine had reported five or fewer bugs, and the remaining four had reported more than five bugs. The participants were unfamiliar with BURT and the selected apps/bugs.

3.2.2 Bug Assignment and Reporting. Each of the 18 participants was randomly assigned to report three bugs (from the 12 selected) with BURT, each bug corresponding to a distinct app. The reporters were instructed to report the bugs in a given (random) order to account for potential learning biases. The bug reporting procedure consisted of five tasks which included the users: (i) watching a short instructional video that explained how to use BURT via an example; (ii) familiarizing themselves with the apps on the web-based emulator; (iii) watching a video demonstrating the observed and expected behavior for each assigned bug (with annotations to ensure proper understanding); (iv) reproducing the bugs on the web-based emulator; and (v) reporting each bug with BURT. We aimed to control for participant understanding of the bugs so that the effect of potential misunderstandings was minimized.

3.2.3 BURT's User Experience Assessment. After the participants reported the three assigned bugs, they answered an online questionnaire that was meant to assess BURT's usefulness and ease of use and to obtain feedback for potential improvements to BURT. Table 2 shows the questions asked to the participants, which are inspired by the PARADISE [39] evaluation framework.

To address **RQ$_1$**, we focused on evaluating BURT's four main features: (1) BURT's app screen suggestions for the OB, EB, and S2Rs; (2) BURT's ability to parse and match the OB, EB, and S2R descriptions provided by the user; (3) BURT's messages and questions given to the user; and (4) BURT's panel of reported S2Rs, which allows the user to visualize and edit the reported S2Rs. Questions Q1-Q5 in Table 2 aim to address **RQ$_1$** and used a 5-level Likert scale [55]. We asked the participants to (optionally) provide a justification/rationale for their answers. Each bug involved multiple screen suggestions, OB/EB/S2R user descriptions, and BURT messages/questions. Questions Q1-Q3 refer to the frequency of these user interactions with BURT.

To address **RQ$_2$**, the reporters assessed BURT's overall ease of use (Q5) and indicated BURT's specific features that were easy or difficult to use for them (Q6). Q5 used a used a 5-level Likert scale and Q6 requested an open-ended response. The reporters were also asked to indicate additional features they would like to see in BURT (Q7). Additional open-ended questions were asked (not shown in Table 2) to obtain feedback on how to improve BURT.

3.3 RQ$_3$: BURT's Intrinsic Accuracy

To answer **RQ$_3$**, we analyzed the conversations that the reporters had with BURT to determine: (1) how often BURT was able to correctly match OB/EB/S2R descriptions to the app execution model as confirmed by the reporters; and (2) how often the user selected one or more of the suggested app screens as being correct (*i.e.*, they match the reporters' OB/EB/S2R descriptions). We computed statistics on the (meta)data that BURT collected from the conversations, such as, the type of messages that BURT asked and the type of user responses (as defined by BURT's Dialogue Manager – see Sec. 2.3).

3.4 RQ$_4$: BURT's Bug Report Quality

We describe the methodology to answer **RQ$_4$** in this section.

3.4.1 ITRAC: A Web Form for Bug Reporting. We implemented a web/template-based bug reporting interface, called ITRAC, using Qualtrics [22]. ITRAC offers the same features of professional issue trackers (*e.g.*, GitHub Issues [7] or JIRA [10]), for reporting the OB, EB, and S2Rs. Specfically, ITRAC provides text boxes with explicit prompts that ask for the bug summary/title and the OB, EB, and S2Rs. In addition, ITRAC prompts the reporter to provide the S2Rs using a numbered list (via a given template). The reporters can write freely their own bug descriptions in the text boxes and also attach images/files. We use ITRAC rather than an existing professional issue tracker to simplify the reporting process for the reporters because they can use ITRAC without having to log into a service.

3.4.2 Bug Reporting with ITRAC. Following the methodology described in Sect. 3.2.1, we recruited 18 more end-users, who did not participate in the BURT study, and asked them to report a subset of bugs using ITRAC. These reporters did not know about BURT, ITRAC, and the selected apps/bugs, and had a similar bug reporting experience to that of the group who reported the bugs with BURT. Five of the new reporters had not previously reported a software bug, eight had reported one to five bugs, and the remaining five had reported more than five bugs.

We assigned the same sets of three bugs used in the BURT study to the new users (trying to match the bug reporting experience) and instructed them to report the bugs using ITRAC in the same order from before. Prior to reporting the bugs, the participants were instructed to: (i) familiarize themselves with the apps by using them on the web-based emulator; (ii) watch a video demonstrating the bugs (with annotations to ensure proper understanding); and (iii) reproducing the bugs on the web-based emulator.

3.4.3 Measuring Bug Report Quality. We estimate the quality of the collected bug reports (via BURT and ITRAC) by assessing the correctness of the OB, EB, and S2Rs described in the reports, based on the quality model proposed by Chaparro *et al.* [28]. Three authors manually compared each collected report with the *ground truth scenarios* from Cooper *et al.*'s dataset [31], which included correct descriptions of the OB and EB and a minimum viable set of S2Rs. Using this methodology, we computed the following: (i) the number of incorrect OB/EB/S2R descriptions; and (ii) the number of missing S2Rs. To limit the effect of subjective assessments, two authors performed the bug report analysis independently and a third author reviewed the results, reaching consensus among all three in case of discrepancies. In order to determine how helpful BURT and ITRAC

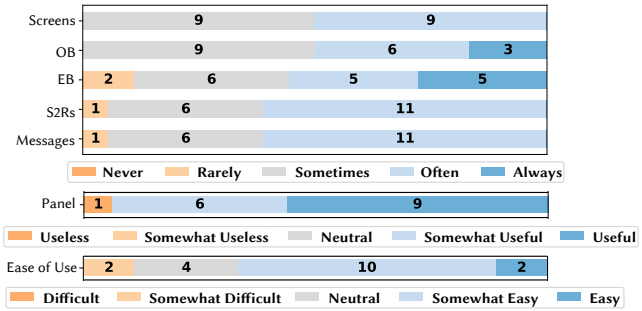

Figure 4: User experience results for BURT (Q1-Q5)

are for novices or more experienced reporters, we analyzed bug report quality across different levels of bug reporting experience.

4 RESULTS AND ANALYSIS

We present and discuss the results of our evaluation for each RQ.

4.1 RQ_1: BURT's Perceived Usefulness

Fig. 4 summarizes the users' answers on: (i) their perceived usefulness of BURT's screen suggestions (row labeled *Screens*); (ii) BURT's ability to understand the user's OB, EB, and S2R descriptions (rows *OB*, *EB*, and *S2Rs*); (iii) how often they were able to understand BURT's messages and questions (row *Messages*); (iv) their perceived usefulness of BURT's panel of reported S2Rs (row *Panel*); and (v) BURT's overall ease of use (row *Ease of Use*).

App Screen Suggestions. Half of the 18 participants (9) agreed that BURT's app screen suggestions were *often* useful, and the other half (9) agreed they were *sometimes* useful. As for their rationales, one participant mentioned that the next S2R screen suggestions *"were useful because they shortened the time it took me to explain how to reproduce the bug"*. Other participants highlighted that the suggestions *"were helpful in making sure I was providing the exact steps I wanted to describe"*, or that BURT *"gave very good suggestions when it could figure out which screen had the bug based on the initial report"*. Some of the participants even hoped that BURT can provide suggestions more frequently. These results illustrate the usefulness of BURT's app screen suggestions.

Some participants noted, though, that *"the suggestions were a little inaccurate"*. We found that the inaccuracies stemmed from BURT not being able to recognize/match the user's OB description because of generic wording, without details (*e.g., "the app crashed"*). Note that the BURT's S2R suggestions are not activated if the OB description is not matched to an app screen, which affected the reporters experience. Also, the participants recommended that it would be useful to have suggestions of *"bug triggering screenshots"*, as currently, BURT's screen captures may not show the bug that the user wants to report. The participants also found some suggestions confusing because the screen captures for the S2Rs highlight *"non-existent buttons"*. This stems from BURT's systematic GUI exploration technique, which can execute events on GUI components such as, layouts or views, which are often not visible to the user.

OB, EB, and S2R Understanding. The reporters have a positive overall impression on how often BURT understood their OB, EB, and S2R descriptions. Specifically, BURT was able to *often* or *always* (*sometimes*) understand the OB/EB/S2R descriptions of 9/10/11 (9/6/6) participants (out of 18). Only two/one participant(s) felt that BURT *rarely* recognized their EB/S2Rs.

Our analysis of the open-ended answers also reveals that some participants were generally satisfied with BURT in terms of bug description understanding. This can be seen in comments such as *"I'm quite satisfied with the recognition rate [for the S2Rs], even better than talking to a real agent"*, *"It always understands my description of the OB/EB when I tried to use keywords from apps"*, *"it was kind of easy for burt to understand my (EB) description"*, and *"It can understand me to describe the error behavior"*. However, several participants had a less positive perception of BURT's bug description understanding stating that it is *"difficult to match* BURT*'s language"*, they *"need to follow specific pattern"* so that BURT is able to understand, and they *"usually had to paraphrase"* their descriptions. These comments stem from our design decision to limit the language that users could use to describe the OB, EB, and S2Rs, and inaccuracies in bug description matching. However, we observed, based on the reporters' comments and their conversations with BURT, that the participants learned how to describe the OB/EB/S2Rs using BURT's preferred formats after reporting the first bug. Still, the reporters' main recommendation was to improve BURT's ability to recognize additional vocabulary and ways of phrasing the OB/EB/S2Rs.

BURT's Messages and Questions. Eleven (of 18) participants *often* understood BURT's messages and questions, while six participants understood them *sometimes*. Only in one case, the reporter *rarely* understood the messages/questions.

The analysis of their rationales reveals that generally BURT's messages/questions were *"very easy to understand"*. One participant wrote that BURT's *"wording was always clear and I could always tell what* BURT *was asking for"*, also echoed by multiple participants. Some participants recommended to improve the messages and questions, as sometimes they were unclear and too similar to each other. For example, for BURT's question *"Was this the last S2R that you performed?"*, the participants suggested to clarify which last S2R BURT was referring to.

The Panel of Reported S2Rs. BURT's panel of reported S2Rs was deemed to be *useful* (*somewhat useful*) by 9 (6) participants. Only one participant found that the panel was *somewhat useless*. The participants commented that the panel was *"Very useful for visualizing a bug report"*, that *"It was good to see what was getting logged"*, and that it was useful *"as a way for me to review that the reproduction steps I entered are complete"*.

Summary of findings for RQ_1: Overall, reporters found BURT's screen suggestions and S2R panel useful. They also had a positive impression of BURT's OB/EB/S2Rs understanding and messages. Improvements are required for BURT to support additional wording of bug report elements and more accurate suggestions.

4.2 RQ_2: BURT's Perceived Ease of Use

Twelve reporters indicated BURT was either *easy* or *somewhat easy* to use. Four reporters were neutral, while two reporters expressed it was *somewhat difficult* to use (see *Ease of use* in Fig. 4).

We analyzed the reporter responses regarding which of BURT's features they found easy/difficult to use. In general, the participants expressed that BURT's GUI *"is really helpful"*, *"concise"*, and *"easy to use and understand"*. Multiple reporters indicated that selecting

Table 3: Quality assessment results for bug reports (BRs) collected by BURT and ITRAC

App-Bug ID	# of BRs		Avg. # of S2Rs		Avg. # (%) of incorrect S2Rs		Avg. # (%) of missing S2Rs		# of BRs with incorrect OB		# of BRs with incorrect EB	
	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT	ITRAC	BURT
APOD-CC3	5	5	4.6	7.4	0.6 (16.7%)	0.6 (9.7%)	6.4 (58.2%)	3.4 (30.9%)	0	1	0	3
APOD-RB	4	4	3.3	4.8	0.0 (0.0%)	1.8 (30.6%)	1.0 (20.0%)	1.0 (20.0%)	1	1	1	0
DROID-CC5	6	6	4	4.3	0.3 (11.1%)	0.5 (10.0%)	1.3 (19.0%)	0.7 (9.5%)	1	2	1	1
DROID-CC6	6	6	4.5	8.5	1.3 (44.4%)	1.2 (13.7%)	5.0 (41.7%)	2.0 (16.7%)	0	3	1	1
GNU-CC9	5	5	6.4	10.2	0.8 (26.7%)	0.2 (2.5%)	4.8 (36.9%)	3.2 (24.6%)	1	0	1	0
GNU-RC	3	3	4.7	4.3	0.0 (0.0%)	0.3 (6.7%)	0.0 (0.0%)	0.0 (0.0%)	0	1	0	0
GROW-CC5	4	4	4.8	7.5	1.3 (28.1%)	0.0 (0.0%)	4.5 (45.0%)	3.5 (35.0%)	0	0	0	0
GROW-RC	4	4	5.8	7.5	1.0 (30.0%)	0.5 (7.1%)	1.8 (21.9%)	1.5 (18.8%)	1	3	1	0
TIME-CC1	5	5	7.8	10.4	1.0 (24.0%)	0.2 (2.9%)	6.6 (41.3%)	6.2 (38.8%)	0	1	0	0
TIME-CC4	4	4	4.3	8	1.0 (24.4%)	0.3 (5.0%)	3.0 (33.3%)	1.3 (13.9%)	1	2	2	1
TOK-CC2	4	4	4.8	10.3	0.3 (8.3%)	0.8 (6.8%)	2.5 (25.0%)	0.5 (5.0%)	2	2	0	0
TOK-CC7	4	4	5	5.8	0.5 (16.7%)	0.3 (3.6%)	1.5 (25.0%)	0.8 (12.5%)	1	0	1	0
Overall	**54**	**54**	**5**	**7.5**	**0.7 (20.4%)**	**0.6 (8.3%)**	**3.4 (32.0%)**	**2.1 (19.4%)**	**8**	**16**	**8**	**6**

BURT's app screen suggestions was easy to use and some of them were very enthusiastic about them. One reporter mentioned that *"I liked the screenshots a lot, very easy to report the process to reproduce a bug"*. Other reporters expressed that *"The suggestions & confirmations were very easy to use. When it had the right idea, confirming it was just a matter of clicking a button"*, and that BURT *"guides the user to provide a "step-by-step" view"*. The panel of reported steps was easy *"to explore"* and it was easy to *"remove events"* from it.

The main reason behind usage difficulties was the limited vocabulary that BURT understands, also observed before for **RQ$_1$**. The reporters recommended to let the users upload their own screen captures when BURT is unable to attach screens to the user's bug descriptions, and the ability to delete/modify *any* step.

Finally, for both **RQ$_1$** & **RQ$_2$**, we found no notable differences in BURT's perceived usefulness and ease of use between different levels of user's bug reporting experience.

4.3 RQ$_3$: BURT's Intrinsic Accuracy

We analyzed the 54 conversations that reporters had with BURT to determine how often BURT was able to correctly (1) match OB/E-B/S2R descriptions to the execution model, and (2) suggest relevant OB/S2R app screens to the reporters.

OB Reporting. We found that in 3 of 54 conversations (5.5%), BURT was able to match the reporter's OB description to the correct screen that showed or triggered the bug, as confirmed by the reporter during the conversation. In 35 of 54 conversations (64.8%), BURT matched the OB description to multiple app screens. In those cases, BURT suggested the top-5 matched screens so that the reporter selected the one s/he was referring to. In 29 of these 35 reports (80%), the reporter selected one of the suggested screens, while in the remaining 6, the suggested screens were irrelevant. For the remaining 16 of the 54 conversations (29.6%), BURT was not able to match the OB description with any app screen because of incorrect OB wording from the user and inaccuracies in BURT's message parser and processing. Overall, BURT was able to correctly match their OB descriptions in 32 of 54 of the conversations (59.3%).

EB Reporting. As described in Sect. 2.4.2, BURT can only match the reporter's EB description when there is a matched/selected OB screen. Otherwise, BURT collects the EB description from the user as is. In the 32 cases when BURT can verify EB quality, BURT was able to match the EB against the OB screen in 17 cases (53.1%) without having to ask the reporter for confirmation. In 6 of the 32 cases (18.8%), the users confirmed the matched OB screen when BURT asked them about that. In the remaining 9 cases (28.1%), BURT was not able to parse the provided EB description.

S2R Reporting. BURT matched a written S2R with a step from the execution model 205 times in total across the 54 conversations (3.8 times per conversation on avg.). In 157 of these cases (76.6%), BURT was able to match S2Rs correctly. BURT predicted and suggested the next S2Rs in 146 cases (4.6 times per conversation on avg.) for the 32 conversations where there was a matched/selected OB screen. We found that the reporters selected 1.6 of the 3.9 suggested S2Rs (on avg.) in 91 cases (62.3%). In 13 of the 32 conversations, the reporter always selected S2Rs from the suggested list, meaning at least one suggestion was correct. In all the 54 conversations, BURT asked the user to rephrase their S2Rs 176 times (3.9 times per conversation on avg.). We found that in at least 59 of these cases (33.5%), the user made a mistake or described the step incorrectly (*e.g.*, *"incorrect result"* or *"no more steps"*).

Summary of findings for RQ$_3$: The results support the users' ratings (**RQ$_1$**) on how often BURT's OB/S2R screen suggestion were useful and how often BURT was able to understand the user's OB/E-B/S2R descriptions. The accuracy assessment revealed cases where BURT's struggles to parse and match the users' descriptions, however, BURT is able to continue with rephrasing prompts. The overall accuracy indicates that the techniques we used in building BURT's components are adequate. Improvements are planned for future work to improve BURT's accuracy.

4.4 RQ$_4$: Bug Report Quality

Table 3 summarizes the quality measures of the $54 \times 2 = 108$ bug reports, collected with ITRAC and BURT, for the 12 bugs in our dataset (each bug is reported in 3 to 6 reports).

S2R Quality. Overall, as shown in Table 3, BURT reports contain fewer incorrect S2Rs than ITRAC reports on avg. (8.3% vs. 20.4%) and fewer missing S2Rs (19.4% vs. 32%), compared to the ground-truth scenarios of the 12 bugs. We performed an analysis to verify whether there there statistically significant differences between BURT and ITRAC on the percentage of incorrect and missing S2Rs. We applied the Wilcoxon signed-rank test [41] and Cliff's delta

(CD) [30] on the results, across the 12 bugs (at 95% confidence level), since we have paired ordinal measurements (for each bug) that do not necessarily follow normal distributions. We found that Burt's bug reports have fewer incorrect (p = 0.0261) and fewer missing steps (p = 0.0025) than Itrac's reports, with a large effect size (CD = 0.5 and 0.527, respectively).

The main reasons for incorrect S2Rs are generic/unclear step wording (4 in Burt and 36 Itrac reports), duplicate S2Rs (13 in Burt reports, zero in Itrac reports), and extra S2Rs (10 in Burt and one in Itrac reports). Examples of steps with unclear/generic wording include *"Add comment"* or *"I searched for tech"*, where the user either refers to high-level app features, which map to multiple steps that are not explicit, or does not specify which GUI components should be used and/or which action should be applied on them. Extra S2Rs are irrelevant reported steps (*e.g.*, "*I did nothing else*"). We identified two main reasons for duplicate S2Rs: (1) user mistakes; and (2) duplicate app screens suggested by Burt and selected by the users. The latter stems from the design of Burt's execution model that considers structural variations of the same screen as different screens (see Sec. 2.4.1). An example is when the users employ different keyboard layouts (*e.g.*, numeric vs. alphanumeric) to enter input values on the same screen.

OB/EB Quality. More Burt reports have an incorrect OB description compared to Itrac reports (16 vs. 8 out of 54 reports), while a comparable number of Burt and Itrac reports have an incorrect EB description (8 vs. 6). We found that there is no statistically significant difference between the number of Burt and Itrac bug reports with incorrect expected behavior (p = 0.1586), with a small effect size (CD = 0.222) in favor of Burt. Fewer Itrac reports than Burt reports have an incorrect observed behavior (p = 0.0352), with a medium effect size (CD = 0.361).

The incorrect OB/EB descriptions (in 18 Burt reports and 10 Itrac reports total) occurred either because the participants did not provide enough details about the bug (*e.g.*, *"the app crashed"*) or they described their inability to perform an action rather than describing the bug itself (*e.g.*, *"I can't add/delete a comment" vs. "Crash when trying to add/delete a comment"*).

For the 18 Burt reports, we found that, in 14 cases the users described the OB/EB incorrectly to begin with and Burt correctly prompted them to rephrase them. Nonetheless, they still reported an incorrect OB/EB. In four cases, Burt accepted the incorrect OB/EB, and in only three of the cases, Burt prompted incorrect OB/EB reporting after the user correctly described them. This is mainly due to Burt's current limitation on the OB/EB wording.

Summary of findings for RQ4: Overall, Burt bug reports contain higher-quality S2Rs than Itrac bug reports, and comparable EB descriptions. The results indicate that improvements to Burt are needed to better collect OB descriptions from the reporters.

Novice vs. Experienced Bug Reporters. Our original expectation was that Burt would help novice reporters more than Itrac, as the experienced reporters likely used template-based reporting systems before.

We compared the quality of the bug reports across different levels of user's bug reporting experience. While we did not observe notable differences in terms of OB/EB quality, we found differences in S2R quality, which we discuss. Table 4 shows the S2R quality results for three groups: novice bug reporters (with *no* prior reporting), intermediate reporters (who had reported 1-5 bugs), and experienced reporters (who had reported 6+ bugs).

Table 4: S2R quality by bug reporting experience

Reporting	# of BRs		Avg. # of S2Rs		Avg. % of incorrect S2Rs		Avg. % of missing S2Rs	
experience	Itrac	Burt	Itrac	Burt	Itrac	Burt	Itrac	Burt
Novice	15	15	3.5	6.7	33.6%	6.7%	45.6%	31.3%
Intermediate	24	27	5.2	7.5	20.1%	11.5%	35.5%	20.0%
Experienced	15	12	6.1	8.6	7.6%	3.2%	12.8%	3.2%
Overall	**54**	**54**	**5**	**7.5**	**20.4%**	**8.3%**	**32.0%**	**19.4%**

Regarding incorrect S2Rs, experienced and intermediate reporters produced about twice as many incorrect S2Rs with Itrac, compared to Burt (33.6% vs. 6.7%, and 20.1% vs. 11.5% on avg., respectively). At the same time, novices produced about five times more incorrect steps with Itrac than with Burt (7.6% vs. 3.2% on avg.). This indicates that Burt helps novices most to avoid incorrect S2Rs.

Table 4 tells a different story for missing S2Rs. Novices and intermediate reporters missed ≈1.5 times fewer S2Rs with Burt, compared to Itrac, while experienced reporters missed four times fewer S2Rs with Burt. Surprisingly, this indicates that Burt helps experienced reporters most to avoid missing steps.

We do not speculate on the reasons behind these observations, as more in-depth studies are needed for proper explanations.

5 LIMITATIONS AND THREATS TO VALIDITY

Before Burt is deployed for use, either systematic app exploration data or crowdsourced app usage data needs to be collected to construct the app execution model. The evaluation results indicate that Burt performs reasonably well with the data collected by CrashScope and only four people. However, we expect that additional data (more covered states and scenarios) would improve Burt's quality verification of reported elements and screen/step suggestions, enabling the reporting of different bug types, under a variety of reproduction scenarios. To confirm our expectations, additional studies are needed for future work.

Burt is evaluated in a lab setting where reporters were exposed to the bugs through videos, rather than letting them find the bugs while using the apps, as users would do in real life. As in prior studies [28, 53], we adopted this setting mainly to reduce participant effort and fatigue. To address the lack of knowledge about the apps/bugs, we instructed the users to get familiar with the apps by using them and with the bugs by reproducing them on the emulator before they reported the bugs. We addressed potential bug misunderstandings via 2/3-word annotations added to the videos.

A diverse group of reporters participated in the studies, who have different levels of bug reporting experience. Since we offered the reporters a monetary incentive for their participation and some of them are students from our institution(s), they may have been motivated to diligently provide high-quality bug reports, which may not necessarily be the case in a real-life scenario. However, we expect this factor to have a minimal impact on the results since (1) we used the same procedure to recruit both Burt and Itrac users, and (2) the bug reporting experience in both reporter groups are almost the same (only two Itrac users have a different experience).

Our evaluation did not consider how easy or difficult it is (for developers) to understand and reproduce the ITRAC and BURT bug reports. Instead, we focused on assessing bug report quality, as done by prior work [28]. Assessing bug report understanding and reproduction is in our plans for future work. Additionally, we did not account for the complexity of the bugs in our dataset. However, we selected bugs of diverse types and distributions of the S2Rs. Our future work will investigate how bug complexity affects BURT.

Finally, given the relatively expensive nature of our evaluation, we limited it to 12 bugs from six apps, reported by 36 participants, which affects the external validity of our conclusions. A larger evaluation, possibly performed on a larger sample of apps, bugs, and participants is in our plans for future work.

6 RELATED WORK

We discuss BURT's advancements in relation to prior work.

Issue/Bug Reporting Systems. A variety of systems currently enable end-users and developers to manually report software bugs, namely, issue/bug trackers (*e.g.*, GitHub Issues [7] or JIRA [10]), built-in bug reporting interfaces in desktop and web apps (*e.g.*, Google Chrome [15]), in-app bug reporting frameworks (*e.g.*, BugSee [19]), app stores [4, 20], and Q&A platforms [26]. These systems typically consist of web/GUI forms (with text-based templates) that allow reporters to provide bug descriptions, indicate bug/system metadata, and attach relevant files. Some of these systems collect technical information (*e.g.*, configuration parameters) and offer screen recording that enable graphical bug reporting.

While existing systems provide features that facilitate bug reporting, they offer limited guidance to bug reporters, lack quality verification of bug report information, and do not provide concrete feedback on whether this information is correct and complete. These are some of the main reasons for having low-quality bug reports, which have important repercussions for developers [65, 66].

Researchers have explored improving bug reporting interfaces, as we do in this work. Moran *et al.* [53] proposed FUSION, a web-based system that allows the user to report the S2Rs graphically by selecting (via dropdown lists) images of the GUI components and actions (taps, swipes, *etc.*) that can be applied on them. More recently, Fazzini *et al.* proposed EBUG [34], a mobile app bug reporting system similar to FUSION that suggests potential future S2Rs to the reporter while they are writing them. Record-and-replay tools [37, 43, 51, 56] offer the ability to record user actions during app usage (*e.g.*, when a bug is found) and replay them later.

BURT offers two main advancements over prior techniques like FUSION. First, BURT was designed to support end-users with little or no bug reporting experience. For example, FUSION was not created to specifically cater to end-users, as inexperienced users found it *more difficult* to use as compared to alternatives [53]. Second, whereas past systems helped to provide structured mechanisms to facilitate the reporting process (*e.g.*, through drop-down selectors) they do not offer *interactive* assistance when reporting a bug. BURT offers such interactivity through its automated suggestions, real-time quality assessment, and prompts for information clarification.

Bug Report Quality Analysis. Surveys and interviews with developers and end-users [48, 57, 65] have identified the observed software behavior (OB), the expected behavior (EB), and the steps to reproduce (S2Rs) the bugs as essential bug report elements for developers during bug triage and resolution. Unfortunately, such elements are often missing, unclear, or ambiguous, as indicated by numerous studies and developers [13, 27, 33, 35, 38, 46, 64, 65], which have a negative impact on bug report management tasks.

In consequence, researchers have proposed techniques to better capture and manage high-quality information in bug reports. Prior work [25, 29, 32, 58, 62, 63, 65] proposed ways to automatically identify different essential elements in bug reports (*e.g.*, S2Rs [49, 61]), analyze their quality, and give feedback to reporters about potential issues in them. In particular, Zimmermann *et al.* [65] proposed an approach to predict the quality level of a bug report based on factors such as readability or presence of keywords. Hooimeijer *et al.* [42] measured quality properties of bug reports (*e.g.*, readability) to predict when a report would be triaged. Zanetti *et al.* [60] proposed an approach based on collaborative information to identify invalid, duplicate, or incomplete bug reports. Imran *et al.* [45] proposed an approach to suggest follow-up questions for incomplete reports. Song *et al.* [29, 58] proposed a technique to detect when the OB, EB, and S2Rs are absent in submitted bug reports. Chaparro *et al.* [28] evaluated the quality of the S2Rs in bug reports through the EULER tool, which integrates dynamic app analysis, natural language processing, and graph-based approaches.

Our work builds upon prior research for the automated quality verification of bug descriptions by developing quality checks for new types of bug elements (*i.e.*, OB/EB) and by designing dialogue flows capable of guiding the user during the bug reporting process.

7 CONCLUSIONS

BURT is a task-oriented chatbot for interactive Android app bug reporting. Unlike existing bug reporting systems, BURT can guide end-users in reporting essential bug report elements (*i.e.*, OB, EB, and S2Rs), provide instant feedback about problems with this information, and produce graphical suggestions of the elements that are likely to be reported.

Eighteen end-users reported 12 bugs from six Android apps and reported that, overall, BURT's guidance and automated suggestions/clarifications are accurate, useful, and easy to use. The resulting bug reports are higher-quality than reports created via ITRAC, a template-based bug reporting system, by other 18 reporters. Specifically, BURT reports contain fewer incorrect and missing reproduction steps compared to ITRAC reports. We observed that BURT is most helpful to novice reporters for avoiding incorrect S2Rs. Surprisingly, BURT seems to be most useful to experienced reporters for avoiding missing reproduction steps.

The reporters provided feedback for refining the supported dialog, by including support additional wordings to describe the OB, EB, and S2Rs. The studies also revealed areas of improvement for BURT with respect to the verification of the reported elements.

ACKNOWLEDGEMENTS

We thank all the study participants for their time and feedback. This work is supported by the NSF grants: CCF-1955837 and CCF-1955853. Any opinions, findings, and conclusions expressed herein are the authors and do not necessarily reflect those of the sponsors.

REFERENCES

[1] 2021. Android Token. https://f-droid.org/en/packages/uk.co.bitethebullet.android.token/.
[2] 2021. Android's Layout and Layout Validation. https://developer.android.com/studio/debug/layout-inspector.
[3] 2021. AntennaPod. https://play.google.com/store/apps/details?id=de.danoeh.antennapod&hl=en_US&gl=US.
[4] 2021. App Store: Ratings, Reviews, and Responses. https://developer.apple.com/app-store/ratings-and-reviews/.
[5] 2021. Appetize.io. https://appetize.io/.
[6] 2021. Droid Weight. https://fossdroid.com/a/droidweight.html.
[7] 2021. GitHub Issue Tracker - https://github.com/features.
[8] 2021. GnuCash. https://play.google.com/store/apps/details?id=org.gnucash.android&hl=en_US&gl=US.
[9] 2021. GrowTracker. https://f-droid.org/en/packages/me.anon.grow/.
[10] 2021. JIRA Bug Reporting System - https://www.atlassian.com/software/jira.
[11] 2021. Microsoft Azure Chatbot Architecture. https://docs.microsoft.com/en-us/azure/architecture/reference-architectures/ai/conversational-bot.
[12] 2021. Mileage. https://fossdroid.com/a/mileage.html.
[13] 2021. An open letter to GitHub from the maintainers of open source projects. https://github.com/dear-github/dear-github.
[14] 2021. React Chatbot Kit. https://fredrikoseberg.github.io/react-chatbot-kit-docs/.
[15] 2021. Report an issue or send feedback on Chrome. https://support.google.com/chrome/answer/95315.
[16] 2021. Spring Boot. https://spring.io/projects/spring-boot.
[17] 2021. Stanford Dependencies. https://nlp.stanford.edu/software/stanford-dependencies.html.
[18] 2021. A Time Tracker. https://f-droid.org/en/packages/com.markuspage.android.atimetracker/.
[19] 2021. https://www.bugsee.com.
[20] 2021. Write a review on Google Play. https://support.google.com/googleplay/answer/4346705.
[21] 2022. Online replication package. https://github.com/sea-lab-wm/burt.
[22] 2022. Qualtrics: Survey Software. https://www.qualtrics.com/core-xm/survey-software/.
[23] Carlos Bernal-Cárdenas, Nathan Cooper, Madeleine Havranek, Kevin Moran, Oscar Chaparro, Denys Poshyvanyk, and Andrian Marcus. 2022. Translating Video Recordings of Complex Mobile App UI Gestures Into Replayable Scenarios. *IEEE Transactions on Software Engineering* (2022), to appear.
[24] 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 (ICSE'20)*. 309–321.
[25] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What Makes a Good Bug Report?. In *Proceedings of the 16th International Symposium on the Foundations of Software Engineering (FSE'08)*. 308–318.
[26] Aaditya Bhatia, Shaowei Wang, Muhammad Asaduzzaman, and Ahmed E Hassan. 2022. A Study of Bug Management Using the Stack Exchange Question and Answering Platform. *IEEE Transactions on Software Engineering* 48, 2 (2022), 502–518.
[27] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. 2010. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*. 301–310.
[28] 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 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19)*. 86–96.
[29] 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 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*. 396–407.
[30] Norman Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.
[31] Nathan Cooper, Carlos Bernal-Cárdenas, 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 *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering (ICSE'21)*. 160–161.
[32] Steven Davies and Marc Roper. 2014. What's in a bug report?. In *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*. 26:1–26:10.
[33] Mona Erfani Joorabchi, Mehdi Mirzaaghaei, and Ali Mesbah. 2014. Works for Me! Characterizing Non-reproducible Bug Reports. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'14)*. 62–71.
[34] 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), to appear.
[35] 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 International Symposium on Software Testing and Analysis (ISSTA'18)*. 141–152.
[36] Jianfeng Gao, Michel Galley, and Lihong Li. 2019. Neural Approaches to Conversational AI. *Foundations and Trends in Information Retrieval* 13, 2-3 (2019), 127–298.
[37] Lorenzo Gomez, Iulian Neamtiu, Tanzirul Azim, and Todd Millstein. 2013. RERAN: Timing- and touch-sensitive record and replay for Android. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. 72–81.
[38] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. 2010. Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*. 495–504.
[39] Melita Hajdinjak and France Mihelič. 2006. The PARADISE evaluation framework: Issues and findings. *Computational Linguistics* 32, 2 (2006), 263–272.
[40] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvanyk, and Kevin Moran. 2021. V2S: a tool for translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE'21)*. 65–68.
[41] Myles Hollander, Douglas A Wolfe, and Eric Chicken. 2013. *Nonparametric statistical methods*. John Wiley & Sons.
[42] Pieter Hooimeijer and Westley Weimer. 2007. Modeling Bug Report Quality. In *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE'07)*. 34–43.
[43] Yongjian Hu, Tanzirul Azim, and Iulian Neamtiu. 2015. Versatile yet lightweight record-and-replay for android. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'15)*. 349–366.
[44] Da Huo, Tao Ding, Collin McMillan, and Malcom Gethers. 2014. An empirical study of the effects of expert knowledge on bug reports. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*. 1–10.
[45] Mia Mohammad Imran, Agnieszka Ciborowska, and Kostadin Damevski. 2021. Automatically Selecting Follow-up Questions for Deficient Bug Reports. In *In proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'18)*. 167–178.
[46] Gün Karagöz and Hasan Sözer. 2017. Reproducing failures based on semiformal failure scenario descriptions. *Software Quality Journal* 25, 1 (2017), 111–129.
[47] Amy J. Ko and Brad A. Myers. 2008. Debugging Reinvented: Asking and Answering Why and Why Not Questions about Program Behavior. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*. 301–310.
[48] Eero I. Laukkanen and Mika V. Mäntylä. 2011. Survey Reproduction of Defect Reporting in Industrial Software Development. In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'11)*. 197–206.
[49] 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 International Symposium on Software Testing and Analysis (ISSTA'20)*. 128–140.
[50] Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. 2014. The Stanford CoreNLP Natural Language Processing Toolkit. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (ACL'14)*. 55–60.
[51] Kevin Moran, Richard Bonett, Carlos Bernal-Cárdenas, Brendan Otten, Daniel Park, and Denys Poshyvanyk. 2017. On-device bug reporting for android applications. In *Proceedings of the IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft'17)*. 215–216.
[52] Kevin Moran, Mario Linares-Váquez, Carlos Bernal-Cárdenas, Christopher Vendome, and Denys Poshyvanyk. 2016. Automatically Discovering, Reporting and Reproducing Android Application Crashes. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'16)*. 33–44.
[53] Kevin Moran, Mario Linares-Vásquez, Carlos Bernal-Cárdenas, and Denys Poshyvanyk. 2015. Auto-completing Bug Reports for Android Applications. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*. 673–686.
[54] Kevin Moran, Mario Linares-Vasquez, Carlos Bernal-Cardenas, Cristopher Vendome, and Denys Poshyvanyk. 2017. CrashScope: A Practical Tool for Automated Testing of Android Applications. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE'17)*. 15–18.
[55] Abraham Naftali Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers.
[56] Zhengrui Qin, Yutao Tang, Ed Novak, and Qun Li. 2016. MobiPlay: A Remote Execution Based Record-and-replay Tool for Mobile Applications. In *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*. 571–582.
[57] Tommaso Dal Sasso, Andrea Mocci, and Michele Lanza. 2016. What Makes a Satisficing Bug Report?. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'16)*. 164–174.

[58] Yang Song and Oscar Chaparro. 2020. BEE: a tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'21)*. 1551–1555.

[59] Lin Tan, Chen Liu, Zhenmin Li, Xuanhui Wang, Yuanyuan Zhou, and Chengxiang Zhai. 2014. Bug characteristics in open source software. *Empirical Software Engineering* 19, 6 (2014), 1665–1705.

[60] Marcelo Serrano Zanetti, Ingo Scholtes, Claudio Juan Tessone, and Frank Schweitzer. 2013. Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. 1032–1041.

[61] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically extracting bug reproducing steps from android bug reports. In *Proceedings of the International Conference on Software and Systems Reuse (ICSR'19)*. 100–111.

[62] Yu Zhao, Ting Su, Yang Liu, Wei Zheng, Xiaoxue Wu, Ramakanth Kavuluru, William GJ 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 (TOSEM)* 31, 3 (2022), 1–33.

[63] 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 41st IEEE/ACM International Conference on Software Engineering (ICSE'19)*. 128–139.

[64] Thomas Zimmermann, Nachiappan Nagappan, Philip J. Guo, and Brendan Murphy. 2012. Characterizing and predicting which bugs get reopened. In *Proceedings of the International Conference on Software Engineering (ICSE'12)*. 1074–1083.

[65] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. 2010. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering* 36, 5 (2010), 618–643.

[66] Thomas Zimmermann, Rahul Premraj, Jonathan Sillito, and Silvia Breu. 2009. Improving bug tracking systems. In *Proceedings of the 31st International Conference on Software Engineering (ICSE'09)*. 247–250.