Assessing the Quality of the Steps to Reproduce in Bug Reports

Oscar Chaparro[1], Carlos Bernal-Cárdenas[1], Jing Lu[2], Kevin Moran[1], Andrian Marcus[2], Massimiliano Di Penta[3], Denys Poshyvanyk[1], Vincent Ng[2]
[1]College of William and Mary, USA – [2]The University of Texas at Dallas, USA – [3]University of Sannio, Italy

ABSTRACT

A major problem with user-written bug reports, indicated by developers and documented by researchers, is the (lack of high) quality of the reported *steps to reproduce* the bugs. Low-quality steps to reproduce lead to excessive manual effort spent on bug triage and resolution. This paper proposes Euler, an approach that automatically identifies and assesses the quality of the steps to reproduce in a bug report, providing feedback to the reporters, which they can use to improve the bug report. The feedback provided by Euler was assessed by external evaluators and the results indicate that Euler correctly identified 98% of the existing steps to reproduce and 58% of the missing ones, while 73% of its quality annotations are correct.

CCS CONCEPTS

• **Software and its engineering → Maintaining software**.

KEYWORDS

Bug Report Quality, Textual Analysis, Dynamic Software Analysis

ACM Reference Format:
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 European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '19), August 26–30, 2019, Tallinn, Estonia.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3338906.3338947

1 INTRODUCTION

When software does not behave as expected, users and/or developers report the problems using issue trackers [60]. Specifically, problems are frequently reported as *bug reports*, *i.e.*, documents that describe software bugs and are expected to contain the information needed by the developers to triage and fix the bugs in the software.

Along with the observed and expected behavior, bug reports often contain the steps to reproduce (S2Rs) the bug. The S2Rs are essential in helping developers to replicate and correct the bugs [39, 59]. Unfortunately, in many cases, the S2Rs are unclear, incomplete, and/or ambiguous. So much so that developers are often unable to replicate the problems, let alone fix the bugs in the software [1, 26, 31–33, 36, 58, 59]. Recently, developers from more than 1.3k open-source projects wrote a letter to GitHub expressing their frustration that the S2Rs are often missing in bug reports [1], and asking for a solution that would make reporters include them in the reports. In addition, prior research found that low-quality S2Rs lead to non-reproducible bugs [31], unfixed bugs [58], and excessive manual effort spent on bug triage and resolution [26, 31, 33, 59]. Low-quality S2Rs are also the main problem with automated approaches attempting to generate test cases from bug reports [32, 36]. For example, Fazzini *et al.* [32] report that a S2R may refer to an interaction outside the system or it may be ambiguous and correspond to multiple interactions. Similar problems were encountered by Karagöz *et al.* [36], who even proposed the adoption of a semi-formal format to express S2Rs, attempting to alleviate such issues.

Ideally, low-quality S2Rs in bug reports should be identified at reporting time, such that reporters would have a chance to correct them. With that in mind, we propose Euler, an approach that automatically analyzes the textual description of a bug report, assesses the quality of the S2Rs, and provides actionable feedback to reporters about: ambiguous steps, steps described with unexpected vocabulary, and steps missing in the report. In this paper, we present the approach and evaluate an implementation geared towards S2Rs corresponding to GUI-level interactions in Android applications. Euler can be adapted to support any GUI-based system.

Euler leverages neural sequence labeling [35, 38] in combination with discourse patterns [27] and dependency parsing [42] to identify S2R sentences and individual S2Rs. Next, it matches the S2Rs to program states and GUI-level application interactions, represented in a graph-based execution model. A successful match indicates that the S2R precisely corresponds to an app interaction (*i.e.*, it is of high-quality). Conversely, a low-quality S2R may match to multiple screen components or app events, may not match any application state or interaction, or it may require the execution of additional steps. Euler assigns to each S2R quality annotations that provide specific feedback to the reporter about problems with the S2Rs.

We asked external evaluators to assess the accuracy and completeness of the quality reports produced by Euler for 24 bug reports of six Android applications. The results indicate that: Euler correctly identifies 98% of the S2Rs; 73% of Euler's quality annotations are correct; and Euler successfully identifies 58% of the missing S2Rs. The evaluators also provided feedback on the perceived usefulness of the information included in the quality reports, on the additional information that should be in them, as well as on usability. The quantitative results of the evaluation and qualitative analysis of the feedback, allowed us to define specific future work for further improving Euler.

We envision Euler being successfully used in three different scenarios: (1) providing automated feedback to the bug reporter at reporting time, prompting a rewrite of the bug report; (2) providing useful information (*e.g.*, the missing S2Rs) to the developers attempting to reproduce the bug; and (3) supporting automated approaches for test case generation (*e.g.*, Yakusu [32]).

ESEC/FSE '19, August 26–30, 2019, Tallinn, Estonia

ACM ISBN 978-1-4503-5572-8/19/08…$15.00
https://doi.org/10.1145/3338906.3338947

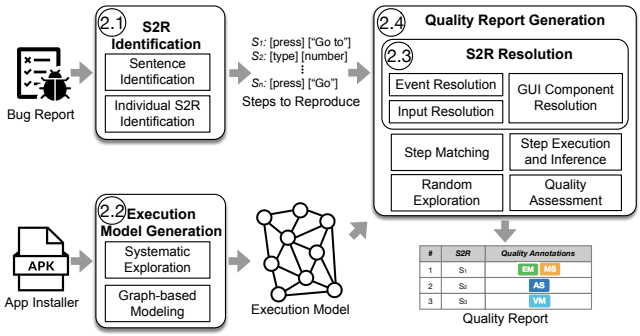

Figure 1: Euler's workflow and main components.

2 ASSESSING S2R QUALITY

We describe Euler (assEssing the qUaLity of the steps to rEproduce in bug Reports), an approach that automatically identifies and assesses the quality of the steps to reproduce (S2Rs) in bug reports. In this paper, we focus on bug reports for GUI-based Android apps, yet Euler can be adapted to work for other platforms. The input of Euler is the textual description of a bug report and the executable file of the Android application affected by the reported bug. The output is a Quality Report (QR), which contains a set of Quality Annotations (QAs) for each S2R, automatically identified from the bug description. The QAs are described in Table 1 of Sec. 2.4. Figure 1 shows Euler's main components and workflow, which are described in the following subsections.

2.1 Identifying S2Rs

The first step in Euler's workflow is the automated identification of sentences describing S2Rs. Then, Euler performs a grammatical analysis on these sentences to identify individual S2Rs. The output is a list of individual S2Rs identified from the bug report.

2.1.1 Identification of S2R Sentences. Euler identifies S2R sentences in a bug report using a neural sequence labeling model [35, 38], which contains the following components:

Model Input. The model input consists of paragraphs in the bug report. Each paragraph is a sequence of sentences, and each sentence is a sequence of words. As there are dependencies between S2R sentences (*i.e.*, often they appear in sentence groups), we use the Beginning-Inside-Outside (BIO) tagging approach [47], where for each sentence in a paragraph, we assign: (1) the label [B-S2R] if the sentence begins a S2R description; (2) the label [I-S2R] if the sentence is inside the S2R description; or (3) the label [O] if it is outside (*i.e.*, not part of) the S2R description.

Word Representations. We represent each word by concatenating two components: word and character embeddings. We use pre-trained word vectors from a corpus of 819K bug reports, collected from 358 open source projects, to capture word-level representations. In order to handle vocabulary outside of this corpus, we use a one-layer Convolutional Neural Network (CNN) with max-pooling to capture character-level representations [41]. We model word sequences in a sentence by feeding the above word representations into a Bidirectional Long Short-Term Memory (Bi-LSTM), which has been shown to outperform alternative structures [52]. The hidden states of the forward/backward LSTMs are concatenated for each word to obtain the final word sequence representation.

Sentence Representations. As suggested by Conneau *et al.* [28], we adopt the simple (yet effective) approach of averaging the vectors of words composing a sentence for capturing sentence-level properties. We represent each sentence by concatenating the averaged word representations from the previous step and a one-hot feature vector that encodes the discourse patterns inferred by Chaparro *et al.* [27], which capture the syntax and semantics of S2R descriptions as well as sentences describing the system's observed behavior (OB) and expected behavior (EB).

Inference Layer. In order to model label dependencies, we use a Conditional Random Field (CRF) for inference instead of classifying each sentence independently. CRFs have been found to outperform alternative models [52]. The output of the inference layer is a label for each sentence where the [B/I-S2R] labels indicate a S2R sentence and the [O] label indicates a non-S2R sentence. Section 3.3 details the model implementation, training, and evaluation.

2.1.2 Identification of Individual S2Rs. Once the S2R sentences are identified, Euler uses dependency parsing [42] to determine the grammatical relations between the words in each sentence and extract the individual S2R from them. Euler utilizes the Stanford CoreNLP toolkit [42] for extracting the grammatical dependency tree of S2R sentences. This tree varies across different types of sentences (*e.g.*, conditional, imperative, passive voice, *etc.*). Therefore, Euler implements a set of algorithms that extract the relevant terms from the dependency trees of each sentence type.

An individual S2R complies with the following format:

```
[action] [object] [preposition] [object2]
```

where the *[action]* is the operation performed by the user (*e.g.*, tap, minimize, display, *etc.*), 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 and textual system input, domain concepts, GUI components, *etc.* A S2R example is: *"[create] [entry] [for] [purchase]"*.

We illustrate Euler's algorithm to identify individual S2Rs from conditional S2R sentences. The bug report #256 [2] from GnuCash [7] contains the conditional S2R sentence: *"When I create an entry for a purchase, the autocomplete list shows up"*. To extract the S2R from the parsed grammatical tree, Euler first locates the adverb *'When'* that is the adverbial modifier (*advmod*) of the verb (*'create'*). Then Euler verifies the existence of an adverbial clause modifier (*advcl*) between the verb and its parent word. Next, Euler captures the verb (*'create'*) as the S2R's *[action]*. Otherwise, the sentence is discarded, as it does not follow the grammatical structure of a conditional sentence. Next, Euler locates the nominal subject (*nsubj*), in this case, the word *'I'*. Euler captures the *[object]* by locating the verb's direct object (*dobj*), *'entry'* in the example, and identifies the nominal modifier of the direct object (*nmod:for*) as the *[object2]*, and the *[preposition]* of the nominal modifier (*case*). The resulting S2R is: *"[create] [entry] [for] [purchase]"*.

The final result of the S2R identification is a sequence of S2Rs extracted from the bug report, $S2Rs = \{s_1, s_2, s_3, ..., s_n\}$. The sequence order is determined by the order in which the S2Rs appear in the bug description, from top to down and left to right, except for a few cases, such as "I do x after I do y", where the order is right to left.

2.2 Execution Model Generation

Euler's quality assessment strategy is based on an execution model that captures sequential GUI-level application interactions and the application's response to those interactions.

In its current implementation, Euler utilizes a modified version of CrashScope's *GUI-ripping Engine* [43, 45] to generate a database of application execution data in the form of sequential interactions. This Engine is an automated system for dynamic analysis of Android applications that utilizes a set of systematic exploration strategies and has been shown to exhibit comparable coverage to other automated mobile testing techniques [43]. A detailed description of the engine can be found in Moran *et al.*'s previous work [43, 45].

Euler's next task is the generation of a graph that abstracts the sequential execution database produced by CrashScope's Engine. The granularity of states in this graph is important, as it will serve as an index for matching the identified natural language S2Rs with execution information. For instance, if the graph were built at the activity-level (meaning that each activity recorded by CrashScope represents a unique state in the graph), then there is potential for information loss, as the GUI-hierarchy of a single activity may change as a result of actions performed on it [23].

To avoid information loss, Euler generates a directed graph $G = (V, E)$, where V is the set of unique **application screens** with complete GUI hierarchies, and E is a set of **application 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 (*e.g.*, 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. Similar execution models have been proposed in prior research on mobile app testing [54]. Each edge stores additional information about the interaction, such as the data input (only for *type* events) and the interaction execution order dictated by the systematic exploration. The graph's starting node has one outgoing interaction only, 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 and the parent/children components.

2.3 S2R Resolution

Euler needs to identify the application interaction that most-likely corresponds to a S2R (*a.k.a. step resolution*). Given a S2R s and program state v_x (*i.e.*, graph vertex or screen), Euler determines the most likely interaction $i = (v_x, v_y = null, e, c)$ for s, where e is an event performed on component c from the screen v_x. For *type* events (*i.e.*, text entry events), Euler identifies the input value specified by s, if any. *Step resolution* can fail to resolve the interaction for s. In that case, the result is either a mismatch (*i.e.*, s does not match a possible interaction in the current screen) or a multiple-match (*i.e.*, s matches multiple events or screen components).

2.3.1 Event Resolution. The first step in the Euler's step resolution workflow is determining the event e that a S2R refers to. Euler supports the following Android events: tap, long tap, open app, tap back/menu button, type, swipe up/down/left/right, and rotate to landscape/portrait orientation.

First, Euler finds the *action group* that the *[action]* from the S2R corresponds to. An *action group* is a category for verbs having a similar meaning, used to express an app interaction. Euler finds the action group by matching the *[action]*'s lemma to the lemma of each verb in the group. Euler supports six action groups, namely OPEN, LONG_CLICK, CLICK, SWIPE, TYPE, and ROTATE. Each group has a set of verbs (*e.g.*, edit, input, enter, insert, *etc.* for TYPE). We defined the groups by analyzing the vocabulary used in the bug reports and applications used by Moran *et al.* [43, 44].

When the *[action]* maps to multiple action groups, Euler resolves the correct group by analyzing the *[object]* and *[object2]* from the S2R (*e.g.*, by identifying GUI-component types in them or matching these to screen components using the matching algorithm described in Sec. 2.3.2). Only the groups TYPE, CLICK, and ROTATE have common verbs. If Euler fails to disambiguate the action group, then it flags the S2R's *[action]* as matching multiple events and saves the corresponding action groups for providing user feedback.

If the *[action]* does not match an action group, then the verb is likely to refer to a generic interaction or an application feature (*e.g.*, "*[create] [purchase]*"). In this case, Euler assumes the *[action]* is expressed in the properties of a GUI component (*i.e.*, its ID, description, or label). Then, Euler attempts to resolve a GUI component that matches the whole S2R or the *[action]*, by using the matching approach define in Sec. 2.3.2. If there is a matched component, the action group is determined as CLICK (if the component is tappable), as LONG_CLICK (if the component is long-tappable), or TYPE (if the component is type-able). Otherwise, the event resolution process fails with an event mismatch result.

Once the action group is determined, Euler proceeds with translating such a group into an event. The OPEN action group is translated as an 'open app' event if the *[object]* matches 'app', the current app name, or a synonym (*e.g.*, 'application'). Otherwise, it is resolved as a 'tap' event. The CLICK group is translated as a 'tap back button' event, if the *[object]* or *[object2]* contains the terms 'back', 'leave', or related terms, and as a 'tap menu button' event, if the *[object]* or *[object2]* contains the terms 'menu', 'more options', 'three dots', *etc.* Otherwise, it translated as a 'tap' event. The rest of the action groups are translated to their corresponding event (*e.g.*, TYPE as 'type'). We also use keywords to determine the direction of swipes and rotations (*e.g.*, 'landscape', 'portrait', 'up', 'right', *etc.*).

All the keywords mentioned in this section are based on our experience with Android apps and the analysis of Moran *et al.*'s bug reports and apps [43, 44].

2.3.2 GUI Component Resolution. The next step in Euler's step resolution workflow is determining the GUI component in the current screen that the event should perform on, according to the S2R. This step is completed only for tap, long tap, tap on menu button, and type events, as they are the only ones that require a component. Before describing how the GUI component resolution works, we describe the base algorithm used to match a textual sequence (*i.e.*, a query) to a GUI component.

Matching algorithm. The algorithm's input is a textual query q (*i.e.*, a sequence of terms), a list of GUI components *GC* (sorted in the order of appearance in a screen), and the application event e

identified from the S2R. The output is the GUI component (from *GC*) most relevant to the query. The relevancy is determined by a set of heuristics and a scoring mechanism based on textual similarity. The algorithm comprises the following steps:

(1) If q contains terms referring to the application or device screen (*e.g.*, 'screen', 'phone', *etc.*), then the first non-tappable component of the current screen (from top to down) is selected and returned as the most relevant component.
(2) If q contains terms that refer to a component type, such as *text field* or *button*, then Euler checks if there is only one component in *GC* of that type (the first type found in the query). If that is the case, then the algorithm selects and returns such a component as the most relevant component.
(3) If q does not contain any terms related to component types, then Euler computes a similarity score between q and each component c from *GC* and selects a set of candidates most-relevant to the query. The similarity score is computed as:

$$similarity(s_1, s_2) = \frac{|LCS(s_1, s_2)|}{avg\left(|s_1|, |s_2|\right)} \quad (1)$$

where s_1 and s_2 are two term sequences, $LCS(s_1, s_2)$ is the Longest Common Substring between the sequences at term level (as opposed to character level), and $avg\left(|s_1|, |s_2|\right)$ is the average length of both sequences. If any of the sequences is empty then the similarity score is zero. If two sequences are exactly the same, then the score is 1 (*i.e.*, the maximum score), otherwise, the score varies from 0 to 1.

The similarity score accounts for common terms between the sequences and the order in which they appear. The order is important because the matching process should be as precise as possible for producing an accurate S2R quality assessment. Before computing the similarity, Euler applies lemmatization to the input word sequences (using the Stanford CoreNLP toolkit [42]).

The similarity between q and each component c in *GC* is taken from the similarity computed between q and the component label, description, and id, in that order. Specifically, the first non-zero similarity score obtained from these sources is taken as the similarity between q and c. Only the components whose similarity with q is 0.5 or greater are considered similar to the query, yet Euler recommends candidates in the order of their similarity score, with the highest first.

From the candidate list, Euler determines the component that is most relevant to the query. There are three cases to consider:

(1) There is one candidate. Euler returns such a component and the matching algorithm ends.
(2) There is more than one candidate. To determine the most-relevant component, Euler executes a set of heuristics. For each component, if its type is Layout and it has only one child in the GUI hierarchy, then the child is returned and the process ends. If none of the candidates satisfy the condition above, but all candidates are of the same type (*e.g.*, text fields), then the component with the highest similarity score is returned. Otherwise, Euler analyzes the candidates with respect to the event e. If e is a typing event, and there is one text field among the candidates, then field is returned. Otherwise, if e is a 'tap' or 'long tap' and there is only one button among the candidates, then Euler returns such a component. Otherwise, the algorithm ends with a multiple-match result and the candidates are saved for providing the quality feedback to the user.
(3) There are no candidates. Euler reformulates the query following a query replacement approach, where a set of predefined synonyms for query terms are used as new queries. If there are no synonyms for the query terms, then the algorithm stops and returns a mismatch result. Each query is executed and if any matches a component, then it is returned. Otherwise, the process ends with a mismatch.

Query Formulation and Component Resolution. Euler uses the S2R constituents as queries, depending on the identified event e for a S2R. These queries are executed using the matching algorithm to find the GUI component that the S2R most likely refers to.

For the 'tap', 'long tap', and 'tap on menu button' events, the first formulated and executed query is the entire S2R (*i.e.*, the concatenation of the S2R's *[action]*, *[object]*, and *[object2]*). If the matching algorithm fails to return a component, only *[object]* or *[object2]* are executed as queries. In both cases, if the *[action]* corresponds to a verb that means "selecting" (*e.g.*, "select", "choose", "pick", "mark", *etc.*), then only checkable or pickable components (*e.g.*, drop-down lists or check-boxes) in the current screen are used as search space. The *[object2]*-based query is executed only if the *[object]*-based one fails. If both queries fail, then the query "*[action] + [object]*" is reformulated and executed. If any of these queries fail to match a GUI component, then the step finishes with either a mismatch or a multiple match, depending on the last matching result obtained.

For *type* events, Euler considers the following S2R cases:

(1) A S2R with a literal in *[object]*, a non-literal in *[object2]*, and the *[preposition]* is one of the following: "on", "in", "into", "for", "of", "as", *etc.* For these cases, the *[object2]* is used as query. For example, for the S2R *"[enter] ['10'] [on] [price]"*, the term *price* is used as query.
(2) A S2R with a non-literal in *[object]*, a literal in *[object2]*, and the *[preposition]* is one of the following: "to" or "with". Then, the *[object]* is used as query. For example, for the S2R *"[set] [price] [to] [10]"*, the term *price* is used as query.
(3) A S2R where the *[object]* is a literal and the *[preposition]* and *[object2]* are null (*e.g.*, *"[enter] ['10']"*), Euler selects and returns the focused component in the current screen, if any.

In any case, the resolution process ends with either a resolved GUI component or a mismatch/multiple-match result.

2.3.3 Application Input Resolution. For *type* events, Euler extracts the input values from the *[object]* or *[object2]*. Specifically, Euler identifies literal values or quoted text. If the input value is missing or generic (*i.e.*, not a literal or "text"), then Euler generates a numeric input value from a counter (a simple, yet effective approach).

2.4 Quality Report Generation

Euler's S2R quality assessment algorithm receives as input the identified S2Rs from the bug report and the system execution graph *G*. The output is a Quality Report (QR), providing an assessment and feedback for each S2R. The algorithm comprises four major steps:

(1) step matching; (2) step execution and inference; (3) random application exploration; and (4) quality assessment.

2.4.1 Step Matching. Euler attempts to match the S2Rs with application states and interactions. Starting with the first identified S2R, Euler resolves an interaction using a set of screens from the graph. First, Euler verifies if the first S2R corresponds to an 'open app' interaction. If it does, then Euler marks the S2R as analyzed and proceeds to the next S2R. Otherwise, Euler builds the interaction. Either way, Euler executes an 'open app' event, and the target state from this interaction is marked as the current execution state. Euler makes sure that the current state corresponds to the screen shown on the device.

Starting from the current state, Euler traverses the graph in a depth-first manner until n levels have been reached. Euler performs step resolution on each state (Sec. 2.3). 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) more states in the graph need to be inspected, hence, the parameter n should increase; (2) there are app states uncovered by the systematic exploration (*i.e.*, not present in the execution model); or (3) the S2R is of low-quality. The parameter n needs to be calibrated per each app. Euler discovers additional app states via random app exploration (Sec. 2.4.3).

Ideally, only one interaction is resolved for the S2R (*i.e.*, on one state only). However, it is possible to resolve multiple interactions, each one on different app states. This is due to variations in the states resulting from different interactions. For example, when providing various app inputs, one screen could have a slightly different GUI hierarchy. The resolved interactions are matched against the interactions from the graph, by matching their source state v_x, the event e, and the component c. If a pair of interactions match on these properties, 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, then it means that the resolved interactions were not covered by the systematic exploration approach, and Euler assumes they are new interactions in the graph. Euler proceeds with selecting the most relevant interaction that corresponds to the S2R, by selecting the one whose source state is the nearest to the current execution state in the graph. In particular, Euler computes a relevant $score = 1/(d+1)$, for each interaction, where d is the distance, in terms of number of levels apart in the graph, between the current state and the source state of the interaction. Euler selects the interaction with the highest *score* as the one that matches the S2R. This decision is made to minimize the number of steps required for reaching the state where the interaction is executed, as described below.

2.4.2 Step Execution and Inference. Each identified interaction from the graph is executed in the device. Any new application screens/interactions are added to the graph during the execution.

The identified interaction in the graph for a S2R could be located in a state far away from the current state. This means that Euler needs to execute intermediate interactions for reaching the state where the interaction is executed on. There may be more than one way to reach such a state. Therefore, Euler selects the shortest path between the current state and the state where the interaction occurs. The interactions in the shortest path are assumed by Euler as inferred steps, missing in the bug report. Euler executes each one of the interactions in the shortest path. At each state, Euler determines the enabled components in the device screen and only the interactions to such components are executed, in the order that they were executed by the systematic exploration approach or the current execution. All the interactions executed correspond to the list of inferred interactions or missing steps in the bug report.

Table 1: Quality annotations for the S2Rs in bug reports

High Quality (HQ):
A step that precisely matches an application interaction
Low Quality (LQ) - Ambiguous Step (AS):
A step that matches more than one GUI component or event
Low Quality (LQ) - Vocabulary Mismatch (VM):
A step that does not match any application interaction
Missing Step (MS):
A step required to reproduce the bug, not described in the bug report

2.4.3 Random System Exploration. As mentioned before, if the step resolution fails for all the inspected states, then it means that the systematic app execution approach (Sec. 2.2) failed to discover app states/screens. To address this issue, Euler performs a random app exploration, starting from the current app screen (shown on the device). The goal is to discover additional app states that could lead to successfully resolving the interaction for a S2R. To do so, Euler identifies the components (different than Layouts and List Views) that have not been executed in the current screen, and randomly selects and executes one clickable component from this set.

The random exploration is performed iteratively y times. At each iteration, x interactions are executed, unless there are no components left to interact with in the current screen. Right after each iteration, Euler updates the graph, the app is restored to the state before the random execution, and the S2R matching, execution, and inference are performed again on the graph's new version. If the S2R is matched against the graph (Sec. 2.4.1), then no more iterations are executed. Else, the random exploration process continues.

2.4.4 Quality Assessment. Euler assigns a set of Quality Annotations (QAs) to each S2R. The QAs are defined in Table 1. If the S2R is resolved/matched against the execution model successfully, then Euler labels the S2R as High Quality (HQ) - *a.k.a.* Exact Match (EM).

If there are inferred application interactions between the previous S2R and the current one, then the current S2R is labeled with Missing Steps (MS). The inferred steps are attached to the annotation for informing the reporter about them. The feedback given to the users is that there are application interactions missing in the bug report that should be executed before the current S2R. Note that the MS annotation does not indicate a problem about this S2R but about the entire list of S2Rs.

If a S2R is not resolved in any of the graph states because of a multiple-component or -event match, then it is labeled as an Ambiguous Step (AS). The feedback given to the users is that either the S2R's *[action]* corresponds to multiple events, or the *[object]* or *[object2]* match multiples GUI components. Examples of matched events or components are shown to the user.

If a S2R is not resolved in any of the graph states because of a mismatch of the S2R with the application, then it is labeled with Vocabulary Mismatch (VM). In the feedback given to the user, Euler

specifies the problematic vocabulary from the S2R constituents (*i.e.*, the *[action]*, *[object]*, *[object2]*, or any combination of these).

Euler generates a web-based Quality Report (QR) with the quality assessment for the S2Rs in a bug report, containing the feedback described above (Fig. 2). The user can click on the matched/inferred interactions to open a pop-up window showing a screen capture of the app and highlighting the GUI component interacted with.

Legend for the Quality Annotations

Exact Match: EM Ambiguous Step: AS Vocabulary Mismatch: VM Missing Steps: MS

#	Identified S2R	Quality Annotations
1	Add favorites	EM This S2R matches the following app interaction: 1. Tap the "item fav (Add to favorites)" text view MS There are app interactions that are missing in the bug report and should be executed before this S2R: 1. Tap the image button 2. Tap the "Chaos Communication Camp Opening" view
2	Go into favorites	EM This S2R matches the following app interaction: 1. Tap the "item starred list (Show favorites)" text view MS There are app interactions that are missing in the bug report and should be executed before this S2R: 1. Tap the "Navigate up" image button 2. Tap the image button 3. Tap the drop down list 4. Tap the list view
3	Select event	AS This S2R matches multiple actions (e.g., "long click" or "click").
4	Remove event in event details screen	VM The term "event in event details screen" does not match a GUI component from the app.
5	Hit BACK button to return	EM This S2R matches the following app interaction: 1. Tap the back button

Figure 2: Euler's Quality Report for Schedule #154 [17].

3 EMPIRICAL EVALUATION

We conducted an empirical evaluation to determine how accurately Euler identifies and assesses the quality of S2Rs in bug reports, and to understand the perceived usefulness, readability, and understandability of the information included in Euler's Quality Reports (QRs). We aim to answer the following research questions (RQs):

RQ$_1$ *What is the accuracy of Euler in identifying and assessing the quality of the S2Rs in bug reports?*

RQ$_2$ *What is the perceived usefulness and quality of the information provided in Euler's quality reports?*

The answer to RQ$_1$ will inform us on improvements to Euler's accuracy. RQ$_2$ will inform us on the presentation and perceived usefulness of the information provided by the QRs.

In order to answer the RQs, we selected a set of bug reports (Sec. 3.1), collected human-produced reproduction scenarios for them (Sec. 3.2), used Euler to identify and assess the quality of each S2R (Sec. 3.3), and asked external evaluators to assess Euler's QRs (Sec. 3.4). We analyze the resulting evaluation data and answer the RQs using a set of evaluation metrics, defined in Sec. 3.5.

3.1 Bug Report Sample

We used 24 bug reports from six Android apps [43, 44]: (1) *Aard Dictionary*, a dictionary and Wikipedia reader [3], (2) *Droid Weight*, a body weight tracker [6], (3) *GnuCash*, a finance expense manager [7], *Mileage*, a vehicle mileage tracker [12], *Schedule*, a conference scheduler [16], and (6) *A Time Tracker* [19]. The apps were selected to cover different domains, as well as involve multiple events (*e.g.*, taps, types, swipes, *etc.*) for using their functionality. These apps are also well-studied, having been utilized in several past works on mobile testing and bug reporting [43, 44].

We collected the entire set of issues (*i.e.*, 785, excluding pull requests) from the issue trackers of the six apps. We randomly sampled 56 issues (*i.e.*, about 10% of the data for each app except GnuCash, which had the largest issue set, and its sample amounts to 5% of the issues). We read the issues and discarded 32, which correspond to new feature requests, enhancements, *etc.*, or bug reports with no S2Rs included. The remaining 24 issues correspond to bug reports, and out of these, 20 describe reproducible bugs and 4 describe non-reproducible bugs. The reports describe different types of bugs, namely crashes (5 reports), functional problems (14 reports), and look-and-feel problems (5 reports). The reports include 88 S2Rs total, 3-4 S2Rs per report on avg., with min. 1 and max. 8.

We manually inspected the 88 S2Rs and estimated that 68 steps are of high-quality, 16 are ambiguous, and four use unexpected vocabulary, while there are many missing steps.

3.2 Ideal Reproduction Scenarios

In order to assess the quality of the S2Rs from the sampled reports, we need a baseline: the ideal list of S2Rs (*a.k.a.* ideal reproduction scenarios). To build the scenarios, we asked six graduate students to reproduce the reported bugs by following the S2Rs provided in the reports. Each bug was reproduced by two students. For each bug report, a student had to (1) (re)install the buggy version of the app on an Android emulator, and (2) try to replicate the reported bug, while writing (in a spreadsheet) each specific step followed. In some cases, the students attempted to replicate the bug more than once. On each attempt, they annotated the detailed reproduction sequence, including any missing steps in the bug report. In most cases, the students succeeded reproducing the reported bug on their second attempt (for the reproducible bugs). The scenarios across the two students per report were highly similar, if not the same. We found only small variations in the scenarios for a single bug (*e.g.*, input values, or cases such as *tap back button* vs. *tap cancel button*).

From the collected reproduction scenarios, we created the ideal reproduction scenario (*i.e.*, the ideal S2Rs) for each bug report, which includes the set of missing steps in the report and the correspondence for each app interaction/step (in the scenario) with the S2Rs from the report. For each reproducible bug, we selected the steps that are more clearly-written, among the submitted scenarios. When necessary, we decomposed the steps into atomic app interactions and added step details (*e.g.*, the location of the GUI-components). We also normalized the vocabulary (*e.g.*, 'hit' or 'press' are changed to 'tap'). For each report describing a non-reproducible bug, we selected the two most similar scenarios to the bug report scenario, and performed the same normalization procedure.

3.3 Euler Implementation and Calibration

We implemented Euler's S2R identification component by adapting the NCRF++ toolkit [53]. We trained the word embeddings with dimension 200 on 819K bug reports collected from 358 open source projects using the *fastText*'s skip-gram model implementation [25]. We used data from Chaparro *et al.* [27] to train the model, using

data from GUI-based systems only. The character embedding layer consists of one convolution layer with kernel size of 3. The size of the character vectors is 50, the size of bi-LSTM vectors is 40, and the size of the discourse patterns vector is 154 [27].

For learning, we use a mini-batch of size 4 using stochastic gradient descent with a 0.05 decayed learning rate to update the parameters. The learning rate is 0.015. We apply 0.5 dropout to the word embeddings and character CNN outputs. We find the best hyperparameters by performing a 10-fold cross validation with 80%, 10%, and 10% of the data for model training, validation, and testing, respectively. The model is trained for up to 500 epochs, with early stopping if the performance (based on F1 score) on the validation set does not change within 25 epochs. The model achieves 73% precision and 81% recall at identifying S2R sentences.

We implemented the remaining Euler components using the Stanford CoreNLP library [42], Chaparro *et al.*'s implementation of the discourse patterns [27], and CrashScope [43]. We used the bug reports by Moran *et al.* [43, 44] to test our implementation and calibrate the parameters. In particular, Euler executes 3 random exploration iterations, with 10 steps each. The depth of graph exploration for the step matching is 6 levels from the current program state. These represent the best parameters, according to our tests.

3.4 Methodology

To address our RQs, we asked human evaluators to assess the quality reports generated by Euler. The study *subjects* (*a.k.a.* participants) are six PhD students, one business analyst, three professors, one postdoc, and one MSc student. The participants have been selected through direct contacts of the authors, taking into account that (i) participants require to have some development experience; and (ii) they need to be available for a task of about two hours.

Based on the ideal reproduction scenario, we created a reproduction screencast showing how the bug can be reproduced, or, for the non-reproducible bugs, how the sequence of steps could be followed. For each bug report, each participant had the following information available: (1) the original bug report; (2) the quality report generated by Euler; (3) the ideal reproduction scenario; and (4) a screencast showing how the bug can be reproduced on a device. Before starting the task, we instructed the participants in a training session (also made available to them through a video), in which we explained the quality annotations and the task to be performed.

We randomly assigned six bug reports to each participant, for which he/she had to evaluate the QR; each QR is evaluated by three participants. The survey questionnaire, implemented through Qualtrics [15], consists of a demographics section and a section for each QRs to evaluate. In the demographics section we ask questions about years of experience on (i) non-mobile app development, (ii) mobile app development, (iii) Android app development in particular, and (iv) use of Android phone. We also ask approximately how many bug reports the participant has ever reported.

For each QR, the questionnaire contains two sections. The first section contains, for each S2R, three questions, for answering RQ_1:

(1) A yes/no question for checking whether Euler correctly identified the S2R (in case of a negative answer, questions (2) and (3) are skipped).
(2) For each annotation produced by Euler for a given S2R, an agree/disagree question aimed at checking its correctness. In case the answer was negative, the respondents were instructed to explain their answer in a free-text form.
(3) In case of missing steps, a third (four check-box) question is formulated for assessing whether Euler's suggested list of missing steps is: (i) correct; (ii) contains extra steps; (iii) is lacking one or more steps; or (iv) some steps are incorrectly ordered. We ask the respondents to use a free-text form to provide an explanation for their answer.

The second section of the survey addresses RQ_2, by asking:

(1) Whether Euler's quality report is easy to read and understand (using a 5-level Likert scale [46]).
(2) Whether the quality report is likely to help users to better write bug reports (using a 5-level Likert scale).
(3) Four free-text questions to indicate what information was perceived useful, useless, and what information should be added to or dropped from the QR.

3.5 Metrics

For addressing RQ_1, we measure Euler's precision and recall at identifying the S2Rs from the bug report by comparing Euler's output with the perfect reproduction scenario (Sec. 3.2). We also measure the proportion of correctly identified S2Rs judged by the participants. Since we involve three participants for each bug report, we consider the correctness assessment provided by the majority.

Regarding the QAs for each step, we compute, for each QA type (Tab. 1), the proportion of annotations judged as correct. We consider the assessment of the majority of participants requiring at least two positive answers. Note that, in this case, a respondent might not have answered question #2 if she judged the S2R as incorrectly identified.

For MS annotations, we measure the proportion of MS annotations suggesting correct, extra, lacking, and unordered missing steps. We also use majority assessment.

To address RQ_2, for each bug report we have two questions, expressed in a 5-level Likert scale. We compute the cumulative number of responses for each of the five levels and we represent them using an asymmetric stacked bar chart.

Regarding the free-text questions related to the usability/quality of the QRs information, we categorized the responses using a card-sorting approach [50] and analyzed each category.

3.6 Results and Analysis

Table 2 summarizes the evaluation results of Euler's quality assessment and feedback[1]. It reports the number of S2Rs (identified by Euler) for each bug report (3rd column - # S2Rs), the (correct/total) number of quality annotations for all S2Rs (4th column - # QAs), the (correct/total) number of annotations across the quality categories from Table 1 (5th-10th columns), and the number of MS annotations for which there are unreported and extra steps in the list of missing/inferred steps (8th and 9th columns, respectively).

[1]Our replication package [14] contains evaluation data and additional results that enable the replication of the evaluation. The package includes bug reports, ideal reproduction scenarios, identified S2Rs in the reports, Euler's quality reports, study survey, Euler's calibration data, and detailed results.

Table 2: Accuracy results for Euler's Quality Annotations (QAs).

App	# of Bug rep.	# S2Rs	# QAs	# AS	# HQ	# MS			# VM
			Correct/Tot.	Correct/Tot.	Correct/Tot.	Correct/Tot.	Not Reported	Extra	Correct/Tot.
Aard Dictionary	2	6	5/8	0/1	4/4	1/3	0	3	-
A Time Tracker	5	22	23/29	-	15/18	5/8	2	8	3/3
Droid Weight	2	6	7/7	-	5/5	1/1	1	1	1/1
GnuCash	9	35	34/53	1/2	16/26	12/19	5	13	5/6
Mileage	4	12	12/17	2/2	4/6	4/5	0	5	2/4
Schedule	2	8	10/10	1/1	5/6	3/3	0	3	1/1
Total	**24**	**89**	**91/124**	**4/6**	**49/64**	**26/39**	**8**	**33**	**12/15**
%	-	-	**73%**	**67%**	**77%**	**67%**	**21%**	**85%**	**80%**

3.6.1 S2R Identification Results. Euler identified 89 S2Rs in the 24 bug reports (Table 2). Only four S2Rs were judged as incorrect, resulting in 96% overall precision. More specifically, the precision is 100% for 20 bug reports, with the exception of four: Aard Dict. #81 (80%) [5], A Time Tracker #1 (75%) [20], GnuCash #471 (80%) [8], and Schedule #169 (67%) [18]. In 73/89 (*i.e.*, 88%) answers there is a perfect consensus among the evaluators across bug reports. We also found that two S2Rs were not identified by Euler (*i.e.*, 98% recall).

We manually analyzed the four misidentified S2Rs and found that the sentences where they were identified from follow the grammatical structure of an S2R (*i.e.*, conditional, imperative, *etc.*), but either: (1) they do not describe an S2R (*e.g.*, "*Change so the week... is restored*", from A Time Tracker #1, is addressed to the developer for fixing the bug); (2) they indicate an app behavior (*e.g.*, "*when dictionary is being verified*" from Aard Dict. #81); (3) they are generic actions (*e.g.*, "*When I perform these sequences of events*"); or (4) they indicate steps to further show how the app correctly behaves in certain circumstances (*e.g.*, "*It shows up again, when you leave the account...*" from GnuCash #471). The two S2Rs not identified by Euler are misspelled or written using noun phrases.

3.6.2 Quality Assessment Results. Table 2 shows that (overall) 73% of the provided QAs/feedback were considered correct by the evaluators, with a percentage ranging between 67% and 80% of correct MS and VM annotations, respectively. The participants reached a perfect consensus in 56% of the cases. For 12 bug reports, Euler achieves 100% accuracy. For the remaining 12 reports, Euler's accuracy ranged from 0% to 80%. We determined the causes of such performance by manually analyzing the participants' answers and Euler's algorithm for those 12 cases, across the QA types.

For two bug reports (*i.e.*, Aard Dict. #104 [4] and GnuCash #620 [10]), Euler incorrectly produced two AS annotations (*i.e.*, for two S2Rs). According to the participants' explanations of their judgment, we found that the annotations were confusing to them, specifically, it was not clear which components Euler's feedback was referring to. For instance, for the Aard Dict. #104's only S2R: "*Tap link to another Wikipedia article*", Euler produced the AS annotation: "*This S2R matches multiple GUI components (e.g., the "1st Link" and "2st Link " views)*". In this case, Euler reached a Wikipedia webpage with multiple links having the labels shown in the annotation. This webpage was unknown to the participants (as it was not shown in the video), hence they did not understand the suggested matched components. In addition, we found that the AS annotation produced for GnuCash #620's 1st S2R: "*Set the color of an account*" did not suggest the correct GUI component (*i.e.*, the color picker in the "creating/editing accounts" screens). The cause for such a mismatch lies in the priority that Euler gives to resolved interactions from program states closer to the current one. One possible improvement is to weight in the similarity score obtained by the resolved components across multiple program states, in such a way that candidates with higher similarity in screens further away from the current one are more likely to be suggested.

In six bug reports, Euler incorrectly assessed the quality of 15 S2Rs as High quality (HQ), which means that the interactions matched/suggested by Euler do not correspond to the S2Rs. We manually analyzed these cases, and found four main reasons: (1) the similarity threshold defined in Sec. 2.3.2 (*i.e.*, 0.5) is too restrictive for some reports; (2) the similarity used to resolve an S2R to a screen (*i.e.*, Formula 1) does not account for small term differences between the S2R (*i.e.*, the query) and GUI components; (3) the synonyms for some terms, used to reformulate the query, may incorrectly boost the similarity score of unexpected GUI components; and (4) the quality of screen information for some applications is low.

We illustrate the first three problems with the report A Time Tracker #35 [22]. The first S2R for this report was identified as "*Restore backup*" and the expected component for the S2R is the menu option "*Restore from backup*", whose similarity to the S2R is 0.4 (the LCS is 'restore' and the average size of both strings is 2.5 - see Formula 1). Because the similarity is lower than the threshold, the component is not returned as a candidate. Next, using the predefined query synonyms, Euler reformulates the query by expanding the S2R to "*Restore back up*" which returns the menu option "*Back up to SD card*", whose similarity to the query is 0.54. In this case, the synonym for backup, "back up", boosted the similarity of the menu option, which was returned as most similar to the S2R. To address these problems, we plan to improve Euler's similarity formula for cases with little term variations, by utilizing shared term frequency and how many terms are in-between the shared terms. To illustrate the fourth problem, consider the case of the 5th S2R from GnuCash #701 [11]: "*Click 'save'*". The incorrect matched component for this S2R was the button "Delete" from the "delete account" screen. The component ID given by GnuCash developers was "btn save", which matches the query. In this case, the mismatch could be used as feedback for developers about problems with the app screens information. Studying the impact of low-quality app information on Euler is subject of future work.

The three S2Rs (from three bug reports) for which Euler incorrectly detected a vocabulary mismatch (VM), involve more than one interaction. For instance, for the 2nd S2R from GnuCash #616 [9]: "*Select export to 'Google Drive'*", Euler failed to match "Google Drive" because of uncovered application states/screen and imprecise S2R parsing and matching.

3.6.3 Analysis of MS annotations. We analyze the results for the MS annotations, which include the steps inferred by Euler.

The participants reached a perfect consensus in 53% of the cases with MS annotations. For six bug reports, Euler incorrectly flagged 13 S2Rs as having missing steps (*i.e.*, they were assigned an MS annotation). For the remaining 26 S2Rs (*i.e.*, 66.7%), from 16 bug reports, the MS annotation was correct (*i.e.*, indeed there are missing steps). For the 13 S2Rs with incorrect MS annotations, all the MSs suggested by Euler are unnecessary for bug reproduction. For the 26 S2Rs with correct MS annotation, Euler suggested extra MSs for 20 of them (*i.e.*, 77%), according to the external evaluators. This means that 33 S2Rs, in total, were judged to have extra missing steps (see the 8th column of Table 2), which represents 85% of the cases. In addition, for 8 S2Rs total, the list of suggested missing steps lacks additional steps (*i.e.*, not detected by Euler), which represents 21%. In all MS annotations, the order of the suggested MSs is correct, meaning that Euler suggests feasible execution paths. However, in all cases, the suggested MSs lack some or have extra steps.

In order to further understand the ability of Euler at inferring and detecting missing steps (MSs), we compared the steps suggested by Euler against the MSs from the ideal bug reproduction scenarios, and computed precision and recall. Euler is designed to favor high recall, because it would be easier for a reporter to just select from the list of missing steps, the ones she actually did and failed to report, as opposed to trying to infer what steps may be missing. Across the 24 bug reports, Euler inferred and suggested 293 MSs (14 steps per bug report on avg.), and there are 158 MSs (6.6 steps on avg.) in the ideal reproduction scenarios. Our analysis reveals that 92 (4.8 on avg.) suggested MSs are correct (*i.e.*, true positives), which represents 31% precision & 58% recall. The results mean that Euler was able to infer more than half of the expected MSs.

We analyzed the 13 cases (from 6 bug reports) for which Euler incorrectly indicated missing steps, and from the correct MS cases, the 20 cases with extra steps. Our analysis reveals two main reasons for such cases, namely, excessive application exploration, and imprecise S2R resolution/matching. Regarding the first limitation, we found that the systematic and random exploration strategies execute more interactions than needed. While this is done by design, trying to uncover as many program states/screens as possible, it leads to excessive inferred steps. Regarding the second problem, any mismatch in the first S2Rs from a bug report can divert Euler's execution, thus producing even more mismatches or no matching at all for the remaining S2Rs. In the latter case, the random exploration takes place, thus producing unnecessary inferred steps. We found that the reason for such mismatches comes from the inability of the similarity scoring formula (*i.e.*, Formula 1) to match the query with single-term text sequences (from the components), and also, from the fact that, in some cases, the random exploration is executed late (after the first S2R matching fails) and unexpected components, with similar vocabulary to the S2R, are returned. Improving the systematic application exploration to uncover as many program states as possible may help to alleviate this problem.

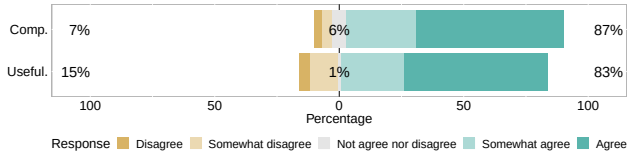

Figure 3: Perceived Comprehensibility & Usefulness of QRs

Finally, we manually analyzed 10 bug reports for which Euler obtained the lowest recall (within the [33% - 78%] range) when inferring the expected MSs. The main reasons for these cases include: (1) incorrect detection of the S2Rs' order from the bug report, and (2) failing to handle special S2Rs. We illustrate the first issue with A Time Tracker #10 [21]. The S2R sentence "*If I press the Back button while viewing the Preferences*" implies the S2Rs "view preferences" and "press back button" are executed in that order. Euler failed to identify the correct order in this case, provoking to not execute one of the MSs: "*click OK button*". One exemplar of the second problem is repetitive S2Rs (*e.g.*, "*enter few fill ups*" from Mileage #53 [13]), which in its current version, Euler does not support.

3.6.4 Perceived Usefulness. Figure 3 shows an asymmetric stacked bar chart depicting the perceived comprehensibility and usefulness of Euler's quality reports. The figure shows positive results. In particular, the study participants agree and somewhat agree that:

- The quality reports are easy to understand (in 59% and 28% of the cases, respectively — 87% on aggregate).
- The quality report can help users write better bug reports (in 58% and 25% of the cases, respectively — 83% on aggregate).

To better understand these results, we analyzed the participants' answers to the open-ended questions about useful/useless information in the QAs, and information that should be added/removed. Our card-sorting analysis resulted in the following categories of useful information produced by Euler:

- Explicit, clear, and fine-grained S2R feedback. One participant mentions that the matched/suggested "*S2Rs are pretty descriptive and would guide the user to complete better the bug description*". Another participant states that the suggested S2Rs "*are a good example of how to write steps-to-reproduce*". Another person acknowledges that "*Developers/maintainers would find this tool *very* useful for their debugging process*". Finally, one participant mentions that Euler "*provides detailed feedback for every single step, not just the overall feedback on all steps*"
- Feedback about incorrect S2R vocabulary. For example, one participant indicated that the tool correctly "*flags words such as 'find' and 'fix' that do not directly translate to an app action*". Another person notes that Euler detects cases that "*the user could improve including some more details in his error report*".
- Feedback about missing steps. For instance, one participant mentions that the suggested missing steps "*can guide the user to list them better*" in the report. Another person mentions that "*They help avoid the guessing part when reproducing the bug*".
- Screenshots for the matched interactions. Some evaluators considered the screenshots helpful for "*identifying the right scenario*

for reproducing the bug" and they were found to "*complement the description of each suggested missing step*".

Regarding information that should be added to Euler's quality reports, some participants suggest that Euler should assess the quality of the application observed and expected behaviors, "*because the user described them but they are not clear*". Other participants suggest clearer wording of the S2Rs (*e.g.*, "instead of "Tap the 'menu save (Export)' text view" » 'Tap Export in menu' "), and visual improvements to the quality report (*e.g.*, adding "*image or some sort of representation of 'navigation drawer' to help locate the button*").

The participants also provided feedback about useless/unclear information in Euler's QAs that should be improved or discarded. Besides incorrect feedback, resulting from Euler's inaccuracy, the participants remarked that:

- Some feedback is unclear. For example, one participant mentions that "*Some missing steps have strange names*". We confirmed that some AS annotations are confusing and found that the suggested missing steps may be give "*little information so the user always needs to click on them and see the image to fully understand the nature of the step*". Euler phrases the suggested/matched S2Rs based on the app internal data. In some cases, this information is not available or may be only clear for the developer (*i.e.*, using "btn save" instead "Save button").
- Setup application steps are not needed. One participant commented that "*some steps that describe the app initial configuration... are not needed to reproduce the bug*". Another participant says "*they are irrelevant for the bug*".

3.7 Threats to Validity

We discuss the threats that may affect the validity of the evaluation.

The main threat to the *construct validity* is the subjectivity introduced in the ideal bug reproduction scenarios given to the study participants. To minimize bias, we created them based on the bug reproduction performed by third-parties, *i.e.*, a group of graduate students. Another threat concerns the procedure to assess Euler's usefulness. Our methodology only assesses the perceived usefulness of Euler. Investigating how users actually benefit from Euler when reporting bugs is subject of future work.

Euler's calibration impacts the *internal validity* of our conclusions. As explained in Sec. 3.3, we used different bug reports (to the ones used in the evaluation) to test and find the best parameters of the approach. Another threat is subjectivity and error-proneness of the human-based evaluation. To mitigate this threat, we relied on three evaluators per bug report, and decided upon majority.

Given a relatively expensive nature of our evaluation, we limited it to 24 bug reports and three evaluators for each report, which affects the *external validity* of our conclusions. A larger evaluation, possibly performed by a diverse (in terms of experience) sample of evaluators on additional bug reports, would be desirable.

4 RELATED WORK

Bug Report Quality Assessment. Zimmermann *et al.* [59] conducted a survey exploring the most useful information in bug reports and proposed a supervised approach to predict the overall quality level of a bug report (*i.e.*, bad, neutral, or good). This approach relies on features, such as readability, presence of certain keywords, code snippets, *etc.* Dit *et al.* [30] and Linstead *et al.* [40] measured the semantic coherence in bug report discussions based on textual similarity and topic models. Hooimeijer *et al.* [34] measured quality properties of bug reports (*e.g.*, readability) to predict when a report would be triaged. Zanetti *et al.* [55] identified valid bug reports, as opposed to duplicate, invalid, or incomplete, by relying on reporters' collaboration information. To enhance bug reports, Moran *et al.* focused on augmenting S2Rs via screenshots and GUI-component images [44]. Zhang *et al.* [56] enriched new bug reports with textually similar sentences from past reports.

Textual Analysis of Bug Reports. Existing research focused on determining the structure of bug reports and its importance in bug triaging and fixing [24, 29, 39, 48, 49, 59, 60]. The work by Ko *et al.* [37] on linguistic analysis of bug report titles is complemented by discourse pattern identification in bug descriptions [27]. Sureka *et al.* [51] analyzed the part-of-speech and distribution of words in titles to find vocabulary patterns for predicting bug severity.

Test Case Generation and Crash Reproduction from Bug Reports. Fazzini *et al.* [32] and Karagöz *et al.* [36] proposed approaches to generate executable test cases from bug reports. Zhao *et al.* [57] proposed a technique to reproduce crashes from bug reports. Different from these approaches, Euler is capable of automatically identifying S2Rs in free-form bug report text, and inferring steps missing in the report. Euler is complementary to these techniques, as it is aimed at improving the quality of reported S2Rs. High-quality S2Rs can help improve the effectiveness of these approaches.

5 CONCLUSIONS AND FUTURE WORK

Euler is an approach for the automated quality assessment of the steps to reproduce (S2Rs) in bug reports. Euler identifies individual S2Rs in bug reports with high accuracy (98%), and produces a quality report (QR), where for each S2R, provides quality annotations (QAs), indicating whether the S2R is well-written, ambiguous, or uses unusual vocabulary. The QR includes a list of missing S2Rs, automatically inferred by Euler, that are needed for reproducing the reported bug. External evaluators found the QRs easy to understand (they agreed in 87% of the cases), while they rated the accuracy of the QAs. 73% of the QAs were deemed accurate, while Euler reported 58% of the missing S2Rs (albeit with a 31% precision). The evaluators consider Euler to be potentially useful (they agreed in 83% of the cases) in helping reporters improve their bug reports.

Future extrinsic studies will confirm the reported perceived usefulness. Before such studies, improvements to Euler's accuracy and to the information included in the QRs are planned, based on the evaluators' feedback. Specifically, we plan to improve the quality of Euler's QR, with: (i) more complete application step sequences; and (ii) additional screenshots to help guide reporters. We also plan to tune Euler's matching algorithm to account for minor variations between text sequences and matches in different parts of the execution model. As discussed in Section 3.6, optimizations to the application exploration strategies are also planned.

ACKNOWLEDGMENTS

This work was partially supported by the NSF grants IIS-1528037 and CCF-1815186, 1815336, 1525902, 1848608, and 1526118.

REFERENCES

[1] 2016. An open letter to GitHub from the maintainers of open source projects. https://github.com/dear-github/dear-github.
[2] 2017. GnuCash's bug report #256. https://tinyurl.com/y3df92g7.
[3] 2019. Aard Dictionary. https://tinyurl.com/mwpxshz.
[4] 2019. Aard Dictionary's bug report #104. https://tinyurl.com/y3xhlky3.
[5] 2019. Aard Dictionary's bug report #81. https://tinyurl.com/y3xvqf3j.
[6] 2019. Droid Weight. https://tinyurl.com/lxazk36.
[7] 2019. GnuCash. https://tinyurl.com/ku9dqq8.
[8] 2019. GnuCash's bug report #471. https://tinyurl.com/y6luonwp.
[9] 2019. GnuCash's bug report #616. https://tinyurl.com/y5edsasv.
[10] 2019. GnuCash's bug report #620. https://tinyurl.com/y3pw69ac.
[11] 2019. GnuCash's bug report #701. https://tinyurl.com/y4e4ny9a.
[12] 2019. Mileage. https://tinyurl.com/cw3uttu.
[13] 2019. Mileage's bug report #53. https://tinyurl.com/y6mo92cm.
[14] 2019. Online replication package. https://seers.utdallas.edu/projects/s2r-quality.
[15] 2019. Qualtrics online survey system. https://tinyurl.com/y4fumc6g.
[16] 2019. Schedule. https://tinyurl.com/bsw89ud.
[17] 2019. Schedule's bug report #154. https://tinyurl.com/y3pg92fr.
[18] 2019. Schedule's bug report #169. https://tinyurl.com/y46l44vr.
[19] 2019. A Time Tracker. https://tinyurl.com/lt4ztgp.
[20] 2019. A Time Tracker's bug report #1. https://tinyurl.com/y4skjrp6.
[21] 2019. A Time Tracker's bug report #10. https://tinyurl.com/y4a698hb.
[22] 2019. A Time Tracker's bug report #35. https://tinyurl.com/y3tvylgs.
[23] Young-Min Baek and Doo-Hwan Bae. 2016. Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'16)*. 238–249.
[24] 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.
[25] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching Word Vectors with Subword Information. *Transactions of the Association for Computational Linguistics* 5 (2017), 135–146.
[26] 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.
[27] 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.
[28] Alexis Conneau, Germán Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. 2018. What you can cram into a single vector: Probing sentence embeddings for linguistic properties. *CoRR* abs/1805.01070 (2018).
[29] 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.
[30] Bogdan Dit, Denys Poshyvanyk, and Andrian Marcus. 2008. Measuring the semantic similarity of comments in bug reports. In *Proceedings of the 1st International Workshop on Semantic Technologies in System Maintenance (STSM'08)*. 265–280.
[31] 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.
[32] 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.
[33] 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)*, Vol. 1. 495–504.
[34] 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.
[35] Zhiheng Huang, Wei Xu, and Kai Yu. 2015. Bidirectional LSTM-CRF Models for Sequence Tagging. *CoRR* abs/1508.01991 (2015).
[36] Gün Karagöz and Hasan Sözer. 2017. Reproducing failures based on semiformal failure scenario descriptions. *Software Quality Journal* 25, 1 (2017), 111–129.
[37] Andrew J. Ko, Brad A Myers, and Duen Horng Chau. 2006. A Linguistic Analysis of How People Describe Software Problems. In *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*. 127–134.
[38] Guillaume Lample, Miguel Ballesteros, Sandeep Subramanian, Kazuya Kawakami, and Chris Dyer. 2016. Neural Architectures for Named Entity Recognition. In *Proceedings of North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'16)*. 260–270.
[39] 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.
[40] Erik Linstead and Pierre Baldi. 2009. Mining the coherence of GNOME bug reports with statistical topic models. In *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR'09)*. 99–102.
[41] Xuezhe Ma and Eduard Hovy. 2016. End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL'16)*, Vol. 1. 1064–1074.
[42] 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.
[43] 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.
[44] 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.
[45] 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 IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*. 15–18.
[46] Abraham Naftali Oppenheim. 1992. *Questionnaire Design, Interviewing and Attitude Measurement*. Pinter Publishers.
[47] Lance A Ramshaw and Mitchell P Marcus. 1999. Text chunking using transformation-based learning. In *Natural language processing using very large corpora*. 157–176.
[48] Swarup Kumar Sahoo, John Criswell, and Vikram Adve. 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *Proceedings of the International Conference on Software Engineering (ICSE'10)*. 485–494.
[49] 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.
[50] Donna Spencer. 2009. *Card sorting: Designing usable categories*. Rosenfeld Media.
[51] Ashish Sureka and Pankaj Jalote. 2010. Detecting Duplicate Bug Report Using Character N-Gram-Based Features. In *Proceedings of the Asia Pacific Software Engineering Conference (APSEC'10)*. 366–374.
[52] Jie Yang, Shuailong Liang, and Yue Zhang. 2018. Design Challenges and Misconceptions in Neural Sequence Labeling. In *Proceedings of the 27th International Conference on Computational Linguistics (COLING'18)*. 3879–3889.
[53] Jie Yang and Yue Zhang. 2018. NCRF++: An Open-source Neural Sequence Labeling Toolkit. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL'18)*.
[54] Razieh Nokhbeh Zaeem, Mukul R. Prasad, and Sarfraz Khurshid. 2014. Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps. In *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST'14)*. 183–192.
[55] 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.
[56] Tao Zhang, Jiachi Chen, He Jiang, Xiapu Luo, and Xin Xia. 2017. Bug Report Enrichment with Application of Automated Fixer Recommendation. In *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17)*. 230–240.
[57] 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 ACM/IEEE International Conference on Software Engineering (ICSE'19)*. 128–139.
[58] 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.
[59] 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.
[60] 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.