Combining Language and App UI Analysis for the Automated Assessment of Bug Reproduction Steps

Junayed Mahmud*, Antu Saha†, Oscar Chaparro†, Kevin Moran*, Andrian Marcus‡
*University of Central Florida (USA), †William & Mary (USA), ‡George Mason University (USA)
junayed.mahmud@ucf.edu, asaha02@wm.edu, oscarch@wm.edu, kpmoran@ucf.edu, amarcus7@gmu.edu

***Abstract*—Bug reports are essential for developers to confirm software problems, investigate their causes, and validate fixes. Unfortunately, reports often miss important information or are written unclearly, which can cause delays, increased issue resolution effort, or even the inability to solve issues. One of the most common components of reports that are problematic is the steps to reproduce the bug(s) (S2Rs), which are essential to replicate the described program failures and reason about fixes. Given the proclivity for deficiencies in reported S2Rs, prior work has proposed techniques that assist reporters in writing or assessing the quality of S2Rs. However, automated understanding of S2Rs is challenging, and requires linking nuanced natural language phrases with specific, semantically related program information. Prior techniques often struggle to form such language ↔ program connections – due to issues in language variability and limitations of information gleaned from program analyses.**

To more effectively tackle the problem of S2R quality annotation, we propose a new technique called AstroBR, which leverages the language understanding capabilities of LLMs to identify and extract the S2Rs from bug reports and map them to GUI interactions in a program state model derived via dynamic analysis. We compared AstroBR to a related state-of-the-art approach and we found that AstroBR annotates S2Rs 25.2% better (in terms of F1 score) than the baseline. Additionally, AstroBR suggests more accurate missing S2Rs than the baseline (by 71.4% in terms of F1 score).

I. Introduction

End-users and developers frequently submit natural language bug descriptions through issue trackers in the form of bug reports. These reports are essential in helping developers reproduce and understand the bugs, which in turn help in fixing them. At the very least, a good bug report should describe the observed behavior (OB) of the app (*i.e.,* the buggy behavior), the expected behavior (EB) of the app (*i.e.,* the correct behavior), and the steps to reproduce the bug (S2Rs) [1, 2]. Among these, the S2Rs are arguably the most important in reproducing the reported bug, an essential step in confirming the presence of the bug.

In GUI-based applications, reproducing a bug requires exercising a series of interactions via the Graphical User Interface (GUI), as described by the S2Rs. A developer (or a tool) trying to replicate a bug needs to understand and extract from each S2R description the user action (a click, swipe, *etc.*) and the GUI component the action is applied to (a button, menu, check box, *etc.*). This is often challenging, as end-users often use their own language and understanding of the app when describing the S2Rs, which may differ from that of the developers. Incorrect or ambiguous S2R descriptions and missing S2Rs hinder developers' ability to understand the bug and lead to non-reproducible bugs [3], delays in bug fixes [4, 5], unresolved bugs [4], and even reopening bugs due to incorrect fixes [5].

The first two authors contributed equally to this work.

To address the problem of low-quality S2R descriptions in bug reports, previous research focused on generating missing S2Rs [6], providing quality feedback to bug reporters [7], automatically reproducing the bug reports [8], or facilitating interactive bug reporting [9–11]. A common issue shared by several of these approaches is related to difficulties in mapping low-quality S2R sentences to elements of the GUI, stemming from the limitations of traditional natural language processing techniques.

In this paper, we present AstroBR (languAge underStanding and assessmenT of the steps to ReprOduce in Bug Reports), a novel approach for improving bug reports at reporting time, by providing quality feedback on S2Rs to the reporter. To do this, AstroBR constructs an application execution model comprising the application interactions via dynamic analysis. Then, for each S2R, it identifies the corresponding application interactions via traversal of an app execution model, guided by GPT-4 [12]. During the traversal, it identifies the best path comprising interactions for the first to last S2R of the bug report. Leveraging the interaction path and the mapped interaction information for each S2R, AstroBR can assess the quality of the reported S2R as well as generate the potential steps that are not reported in the bug, but required to reproduce the bug (*i.e.,* missing steps).

Unlike previous work, AstroBR uses an LLM (GPT-4) for three different tasks, in three distinct ways. *First*, it automatically extracts S2R sentences in a natural language bug report, framing the task as a text classification problem. For this task, we evaluated three prompt templates, based on three prompting strategies (*i.e.,* zero-shot, few-shot, and chain-of-thought), using a development set of 54 bug reports. *Second*, it extracts individual user actions and the GUI components interacted with from the S2R sentences, framing the task as a phrase extraction problem. For this task, we evaluated three additional prompt templates, based on the three prompting strategies. *Third*, it maps the extracted actions and GUI components to elements of an app execution model, framing the task as a guided graph exploration problem. For this task, we evaluated six prompt templates, based on the three prompting strategies. GPT-4 is used to guide the systematic and efficient exploration of the execution model. During this mapping process, AstroBR identifies problems with the S2R sentences (*e.g.,*

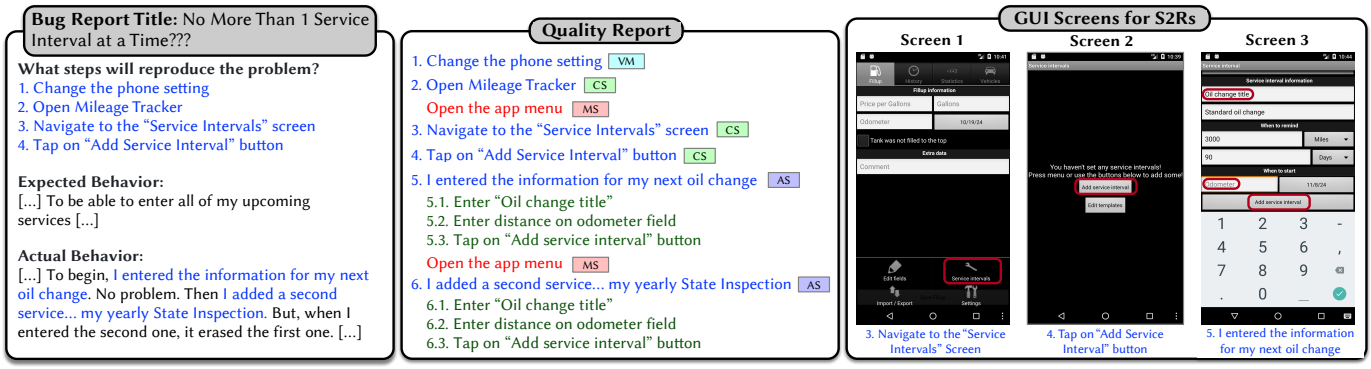

Fig. 1: Bug Report Quality Annotations

ambiguous descriptions, vocabulary mismatches, or missing steps) and generates a quality report with annotations reflecting these issues. When S2Rs are missing, AstroBR also generates the missing steps and includes them in a quality report.

We compared the performance of AstroBR with a recent state-of-the-art technique, Euler [7], utilizing a test dataset consisting of 21 bug reports having 73 S2R sentences, from five Android applications. AstroBR achieves better results in generating quality annotations (by 25.2% F1 score) and identifying missing S2Rs (by 71.4% F1 score).

In summary, this paper makes the following contributions:

- An approach that integrates graph-based dynamic analysis and LLMs (*i.e.,* GPT-4) to automatically identify and extract S2Rs from the bug report, assess their quality, and generate missing steps.
- A ground truth dataset containing the identified and extracted S2Rs, quality annotations, and missing steps to evaluate AstroBR with the existing baselines. This dataset contains 75 annotated bug reports.
- A replication package [13, 14] containing all the dataset and source code to replicate and validate our results.

II. Quality Model for Reproduction Steps

In this paper, we adopt the quality model proposed by Chaparro *et al.* [7], with the following quality categories for the steps to reproduce the bug (S2Rs) in a bug report:

- Correct step (**CS**): the step corresponds to a specific interaction and GUI component on the application.
- Ambiguous Step (**AS**): the step corresponds to multiple interactions on GUI components on the application.
- Vocabulary Mismatch (**VM**): the step does not correspond to any interactions or GUI components on the application due to misaligned terminology.
- Missing Steps (**MS**): interactions that are required to replicate the bug, but not reported in the bug report.

We illustrate the definitions with an example in Figure 1. The bug report presented in the figure comprises six S2Rs, each annotated with the above categories. ① The first S2R is *"Change the phone setting"*, which does not represent any interactions in the app. Therefore, this S2R is annotated as **VM**. ② The second S2R contains only one individual S2R, *"Open Mileage Tracker"*, representing only one app interaction. Therefore, this S2R is annotated as **CS**. ③ The third S2R, *"Navigate to the 'Service Intervals' screen"*, does not immediately follow after the second step. There is a required intermediate step, *"Open the app menu"*, which must be performed by tapping the "three dots" button in the bottom left menu bar of Screen 1. Therefore, this missing step is included in the quality report and annotated as **MS**. With this missing step added, the third reported S2R requires a single interaction that can be reliably mapped to the GUI, *i.e.,* performing a *click* operation on the "Service Interval" button on Screen 1. Therefore, it is categorized as **CS**. ④ *"Tap on 'Add Service Interval'* requires only one interaction in the GUI, *i.e.,* performing a *click* operation on "Add Service Interval" component on Screen 2, and hence, is annotated as **CS**. ⑤ The fifth S2R, *"I entered the information for my next oil change"*, requires multiple operations. At first, a user has to enter the *"Oil change title"* by performing a *type* operation on the "title" text field at the top of Screen 3. The individual S2R for this interaction is *"Enter Oil change title"*. Secondly, s/he has to enter a value in the "Odometer" text field on Screen 3 by performing a *type* operation which implies an individual S2R: *"Enter distance on odometer field"*. Finally, s/he has to perform a *click* operation on the "Add Service Interval" button on Screen 3. This interaction represents the individual S2R: *"Tap on Add service interval button"*. As three interactions are required to complete the fifth step in the bug report, it is labeled as **AS**. ⑥ To execute the sixth S2R, *"I added a second service my yearly State Inspection"*, there is another step missing, *"Open the app menu"*, and it is labeled as **MS**. Moreover, the sixth step requires the same three individual S2Rs as the fifth step and annotated as **AS**. In the next section, we explain how this quality model can be automatically applied to bug reports.

III. AstroBR: Automated S2R Quality Assessment

This section presents AstroBR, an automated approach that leverages an LLM and a graph-based app execution model to assess the quality of the steps to reproduce (S2Rs) in textual bug reports. AstroBR identifies, extracts, and processes the S2Rs from a bug report to detect which ones are correct, ambiguous, missing, or phrased using language that does not correspond to a target app, according to the quality model described in Section II. AstroBR generates a quality report with annotations that provide feedback to the reporter about problematic S2Rs

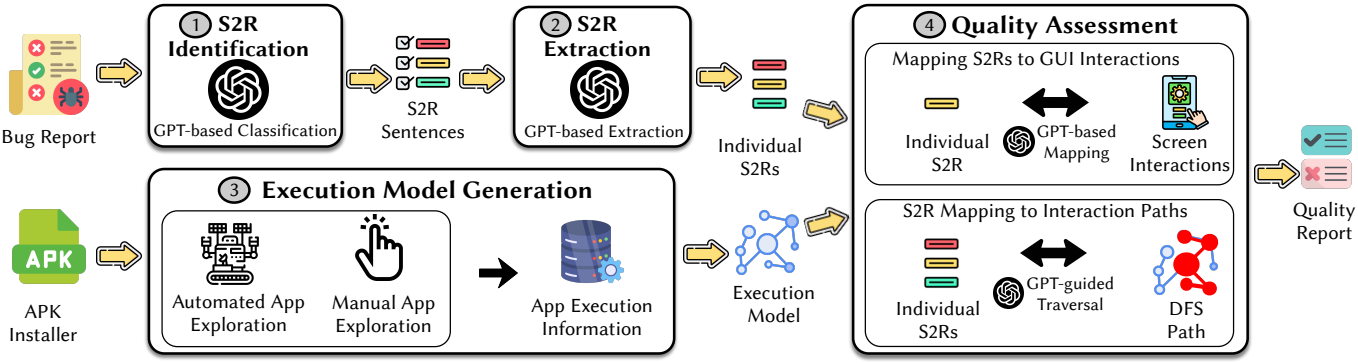

Fig. 2: The AstroBR Approach

and includes generated missing S2Rs. AstroBR has four main components, as illustrated in Figure 2:

1) **S2R sentence identification**: AstroBR identifies the sentences that describe any S2Rs (Section III-A).
2) **Individual S2R extraction**: AstroBR extracts phrases describing individual S2Rs from S2R sentences (Section III-B).
3) **App execution model generation**: AstroBR builds a graph-based model using automated and manual app execution (Section III-C).
4) **S2R quality assessment**: AstroBR maps individual S2Rs to GUI-level interactions captured in the app execution model, providing feedback about high- and low-quality S2Rs as well as missing steps in a quality report (Section III-D).

We leverage the language processing capabilities of LLMs (*i.e.,* GPT-4) across the three phases, integrating these with GUI-level dynamic app analysis to assess S2R quality. The selection of GPT-4 as the LLM was based on its demonstrated effectiveness in language and bug understanding tasks, including bug reproduction [15] and analysis [6]. In the remainder of this section, we detail AstroBR's components or phases.

A. S2R Sentence Identification Phase

AstroBR automatically identifies sentences that describe any steps to reproduce (S2R) in the bug report (see the blue sentences in Fig. 1). This is necessary as the bug report typically includes other content, notably the observed (OB) and expected app behaviors (EB). We formulate this task as a text classification task, using LLMs. AstroBR decomposes the bug report into a list of sentences and asks the LLM to identify which of these sentences describe any S2Rs. Sentence parsing is done using the Stanford CoreNLP toolkit [16] and heuristics. We experimented with three types of prompts, each one providing a different context to facilitate the task for the LLM (*e.g.,* the definition of S2Rs and guidelines on how to distinguish them from other content like the OB and EB). Section IV describes the process we followed to develop and evaluate these prompts.

B. Individual S2R Extraction Phase

After identifying S2R sentences, AstroBR asks the LLM to extract the individual S2Rs from these sentences in a particular format (described below). Individual S2Rs are phrases that describe a single, atomic interaction with the app. Individual S2R extraction is needed because S2R sentences may describe multiple interactions with the app together with content such as the OB (*e.g.,* "I opened the app and clicked on the Start button" or "The app crashes if the user checks the Angle Box"). In addition, different S2R sentences may describe the same interaction (*e.g.,* "... the user checks the Angle Box" and "give the Exercise a name and check the Angle Box"). AstroBR resolves this redundancy by asking the LLM to provide only one S2R among all extracted individual S2Rs that describe the same interaction.

The output of this phase is a list of individual S2Rs extracted in the order they appear in the sentences from left to right and top to bottom. AstroBR asks the LLM to represent the individual S2Rs in the following format: `[action][object][preposition][object2]`. The `[action]` is a verb associated with the app interaction (tap, long tap, enter, *etc.*). The `[object]` is the GUI component upon which the action is performed. The `[object2]` is additional information related to the object connected by a `[preposition]`. For example, the S2R "Click any button on this page" is formatted as `[Click] [any button] [on] [this page]`.

We designed and evaluated three prompt types to extract individual S2Rs via GPT-4. Each prompt implements a different approach, providing different contexts about the task (*e.g.,* examples that illustrate how to accomplish the task). Section IV details the prompts and the process we followed to design and evaluate them.

C. App Execution Model Generation Phase

AstroBR's quality assessment relies on mapping individual S2Rs to interactions that can be executed on the app to replicate the reported bug. This requires collecting and representing possible user GUI interactions, for which we adapt graph-based representations and dynamic app execution strategies from prior work [17, 18].

AstroBR creates an app execution model represented as a directed graph, $G = (V, E)$, where V represents the set of unique GUI screens for an app, and E represents the set of unique interactions that users can perform on the GUI components of the screens. A GUI screen (*i.e.,* node) is represented as a hierarchy of the GUI components and layouts. Two GUI screens with different GUI component hierarchies are considered distinct graph nodes. Each interaction (*i.e.,* edge) in E is represented by a unique tuple in the form of (v_x, v_y, e, c),

where c is a GUI component of screen v_x and e in an action (tap, type, *etc.*) performed on c, resulting in a transition to another screen v_y. Each edge contains additional interaction metadata such as the interacted GUI component type, ID, text (*i.e.,* label), and description.

To build the execution model for an app, AstroBR parses GUI interaction traces collected from automated app exploration and manual app usage. AstroBR executes an adapted version of the CrashScope tool [19, 20], which implements multiple automated exploration strategies to interact with the UI components of app screens, trying to exercise as many app screens and GUI components as possible. In the process, CrashScope collects app screenshots and XML-based GUI hierarchies and metadata for the exercised app UI screens and components. As CrashScope may fail to interact with certain GUI screens and components that app users would normally interact with, AstroBR can also make use of interaction data collected from manual app usage and testing. In this paper, for the development set, we used the set of traces collected by Saha *et al.* [18] which consists of 10-12 manually recorded feature interaction traces for each of the 5 test applications. For the prompt development dataset, two authors collected the same number of traces for each of the 31 apps. These recordings include all the app GUI interactions starting from launching the application to the last step related to carrying out an application feature (more details of this process, used for prompt development and evaluation, are found in Section IV-A). In practical applications of AstroBR, manual executions can be collected in several ways. For example, developers can enable user monitoring features in the app and perform record-and-replay during in-house or crowd-sourced app testing [21]. AstroBR parses the interaction traces generated by CrashScope and the traces collected during app usage/testing to build the graph, according to the graph format we previously described in this section (details found in Section IV-A).

D. S2R Quality Assessment Phase

The app execution model captures possible interaction sequences that a user could perform when using or testing an app as paths in the graph. To assess the quality of the S2Rs, AstroBR attempts to map each individual S2R to interactions (*i.e.,* edges) along these paths. To do so, AstroBR implements an LLM-guided depth-first-search (DFS) graph traversal to establish the correspondence between an individual S2R and interactions on a given screen.

Any S2Rs that cannot be mapped to a graph interaction are labeled as having a Vocabulary Mismatch (**VM**). S2Rs that map to multiple interactions performed on a single screen (*i.e.,* a node) are labeled as Ambiguous Steps (**AS**). Those that map to single interactions within a sequence are labeled as Correct Steps (**CS**). Finally, for the mapped S2Rs that correspond to non-consecutive interactions spanning different screens in a path, additional interactions are required to connect them to form a complete path. These additional interactions are used to generate individual S2Rs that are labeled as Missing Steps (**MS**) and used to fill in the "gaps" between the existing S2Rs.

1) Mapping Individual S2Rs to Interactions on a Screen: Mapping an individual S2R (S2R, hereon) to interactions on a given screen is supported by GPT-4. For a graph node (*i.e.,* a screen), AstroBR asks GPT-4 to identify which of the outgoing edges (*i.e.,* interactions) from that node correspond to the S2R. Both the S2R and graph interactions are represented textually: the S2R is extracted from the bug report, while each interaction is represented as a tuple of textual information (*e.g.,* the event description and the label of the interacted GUI component). We designed and evaluated a set of prompts using different prompting strategies to accomplish this mapping in a 2-step manner: a first prompt asks GPT-4 to return a yes/no answer on whether an individual S2R maps to the interactions of a given screen and if the answer is yes, a second prompt asks GPT-4 to return the list of corresponding interactions. The methodology used to develop and evaluate the prompts is detailed in Section IV-B.

2) Graph Traversal and S2R Mapping to Interaction Paths: To map all the S2Rs from a bug report to app interaction sequences, AstroBR implements an algorithm that traverses the graph in a depth-first-search (DFS) manner, aiming to map the S2Rs to interactions along the DFS paths. When S2Rs map to non-consecutive interactions within a path, AstroBR connects these interactions by selecting the shortest path between the nodes where these interactions occur. Since multiple paths may map to the S2Rs, AstroBR selects the path with the most mapped S2Rs or the shortest path, if multiple paths have the same number of mapped S2Rs.

The DFS traversal of the graph is guided by the LLM-based mapping approach from Section III-D1, as only edges that map to S2Rs are traversed, avoiding the need to explore the entire graph. While none of the S2Rs can map to any interaction in the graph (in which case AstroBR would traverse the graph entirely), this scenario is expected to be rare, as we assume reporters would describe at least one S2R using the app's vocabulary and the graph is as complete as possible, covering a broad range of screens and interactions.

Algorithm Details. AstroBR's DFS-based graph traversal algorithm is recursive. It receives an S2R s and graph node n as input, where s is the first item in the S2R list L (the bug report S2Rs). The algorithm returns either: the best DFS path p (starting from n) that maps to a subset of S2Rs in L (possibly including s), or no path if no S2Rs can be mapped.

The traversal begins with the first S2R from the bug report and the starting node of the graph, which contains "*open app*" interactions that navigate to the screens users usually see upon launching the application.

The algorithm has two main logic branches:

1) If S2R s does not map to any of the outgoing interactions I from n, the algorithm recurses, attempting to map s on each node connected to n by I. If this traversal results in no DFS paths mapped to s or following S2Rs in L, s is labeled as having a Vocabulary Mismatch (**VM**), and the algorithm recurses with the next S2R in L at the current node n. This means that the S2R s cannot be mapped to any node in the

(sub)graph starting from n, then the algorithm attempts to map the next S2R.

2) Conversely, if s maps to interactions in I, the algorithm checks whether there are one or more mapped interactions. If there is a single interaction, s is labeled as a Correct Step (**CS**); if there are multiple, it is labeled as an Ambiguous Step (**AS**). The algorithm then recurses with the next S2R in L on each node connected to n by only the mapped interactions from I. Essentially, if the algorithm succeeds at mapping s to interactions from n, then it proceeds with attempting to map the next S2R to the resulting nodes after navigating to the mapped interactions.

It is possible that s maps to interactions in I (second branch above), but there are "gaps" between the previous mapped S2R and s: if their mapped interactions are not consecutive in the DFS path. If this is the case, the algorithm connects them by determining the shortest path between the involved nodes. The interactions used to connect the nodes are then labeled as Missing Steps (**MS**). Note that this shortest path may include interactions outside the DFS path, as we are not limiting the shortest path search to the DFS path alone. A shorter path may exist that bypasses parts of the DFS path.

After traversing a node with a given S2R (in either branch above), it is possible that when calling the algorithm recursively on a set of interactions (*i.e.,* when navigating down DFS paths), it returns multiple DFS paths mapped to the S2Rs. If this is the case, the algorithm selects the DFS path to return based on the following criteria: prioritizing the path with the most mapped S2Rs in L, or, if paths have the same number, choosing the shortest path.

The traversal continues until all S2Rs in L have been exhausted or until none of the S2Rs are mapped to any DFS paths. If all S2Rs have been mapped, but there are still nodes along a DFS path, the algorithm does not proceed to check additional nodes down the current DFS path. To prevent re-processing nodes and their interactions, the algorithm marks each (node, S2R) pair as visited before it processes the node and S2R.

3) Quality Report Generation: The returned DFS path contains interactions mapped to all or a subset of the S2Rs from the bug report. Each S2R is labeled as either a Correct Step (CS), Ambiguous Step (AS), or Vocabulary Mismatch Step (VM). In addition, interactions identified to fill in the "gaps" between S2Rs are labeled as Missing Steps (MS). For evaluation purposes, we also mark the corresponding S2Rs with missing steps as MS, so that we can perform a fine-grained analysis of results (more details found in Section V).

IV. AstroBR's Prompt Development and Evaluation

This section describes how we developed and evaluated the LLM prompts for three distinct tasks: (i) S2R sentence identification, (ii) individual S2R extraction, and (iii) individual S2R mapping to app interactions. We adopted a rigorous, comprehensive, and data-driven approach in which we designed an initial prompt that was iteratively evaluated and refined into new prompts. Prompt development and evaluation followed a quantitative and qualitative methodology based on a set of Android app bug reports. Overall, we designed and evaluated 12 prompt templates across all three tasks. To generate GPT-4 responses with the prompts for all tasks, we used a temperature of 0 to minimize randomness/non-determinism in the responses.

A. Development Dataset Construction

We constructed a dataset of 54 bug reports and corresponding ground truth data, with manually identified S2R sentences, individual S2Rs, and interactions mapped to each S2R.

1) Bug Report Collection: We selected the 54 bug reports from the dataset released by Saha *et al.* [18], which contains reproducible mobile app bug reports from the AndroR2 dataset [22, 23]. These reports describe bugs for 31 Android apps of various domains (*e.g.,* web browsing, WiFi network diagnosis, and finance tracking). The reported bugs span different bug types, namely crashes (15 reports), output problems (19), UI cosmetic issues (13), and navigation problems (7).

2) S2R Sentence Labeling: Two authors annotated the 1,031 sentences present in the bug reports as either S2R or non-S2R, following the S2R criteria and methodology defined by Chaparro *et al.* [24]. One author annotated each sentence, while the second author validated the annotations, recording disagreements and their rationale. The authors agreed on the annotations for 1,002 sentences (97.2%, 0.91 Cohen's kappa [25]), which represents near-perfect agreement. Disagreements were resolved via discussion. The most common reasons for disagreements were content misinterpretations and mistakes (*e.g.,* a sentence describing the observed behavior, not S2Rs). In total, the 54 bug reports contain 189 S2R sentences (3.5 per report on average), while the remaining 842 sentences describe non-S2R content.

3) Individual S2Rs Extraction: Two authors manually inspected the 189 S2R sentences to extract individual S2Rs (phrases describing a single interaction with the app). One author read and extracted the individual S2Rs in the format defined in Section III-B. The extracted S2Rs were validated by a second author. They discussed disagreements to reach a consensus where needed. From the 189 S2R sentences, we extracted 246 individual S2Rs with an agreement rate of 97.6%.

4) S2Rs to GUI Interaction Mapping: To create ground truth mappings between individual S2Rs and GUI app interactions, we first built the execution models (*i.e.,* graphs) for the 31 apps corresponding to the bug reports. To do so, we executed the CrashScope tool [19] using the corresponding APKs (from the original dataset [18, 23]) and a Pixel 2 Android emulator. We also used the manual interaction traces collected as part of Saha *et al.*'s dataset [18]. Both the CrashScope and manual interaction traces consist of GUI-event execution traces and (video) screen captures showing the executed interactions. We used Song *et al.*'s toolkit [9] to parse the traces and build the execution graphs.

Two authors manually inspected the execution data, graphs, and reproduction screen captures to map each S2R to graph nodes and interactions. One author first inspected this data to identify the GUI screen and target GUI component for each S2R. Then, the author identified the graph node corresponding to such screen, and within it, the interaction corresponding to the

S2R. In the process, missing steps and the path that represented a minimal bug reproduction scenario were identified. A second author followed the same procedure to verify the interactions/nodes mapped to the S2Rs and the reproduction paths identified by the first author. Both authors discussed any disagreements, involving a third author where necessary.

We applied the above methodology on a sample of 10 bug reports, in such a way that they spanned different bugs types, apps of different domains (9 apps), and S2R types (taps, types, *etc.*). The two authors created the ground truth for 46 individual S2Rs among 49 individual S2Rs for the 10 bug reports, agreeing on 43 S2Rs (agreement rate of 93.5%). The excluded three individual S2Rs did not have corresponding app interactions in the execution model because they are performed outside the app (*e.g., "install the app"*), and hence, are not included in the graph. Common reasons for disagreements were unclear individual S2Rs and misinterpretation of graph nodes/interactions. During the data creation process, we realized that it would take the two authors a prohibitive amount of effort to create the data for the remaining 44 bug reports. Therefore, we decided to focus on the S2R mapping prompt development using only the 10 bug reports and redirect our effort to curating the test data used for AstroBR's evaluation (see Section V).

B. Prompt Development Methodology

For each of three tasks where AstroBR uses GPT-4, our overall data-driven methodology used three prompting strategies, commonly used in software engineering research [26]:

- Zero Shot (ZS) prompting: starting from a base prompt template that includes the task description, input, and response format, we iteratively executed, evaluated, and refined the template until the performance plateaued. This involved computing performance metrics (precision, recall, and F1 score) against the ground truth, qualitatively analyzing false positives (FP) and negatives (FN), and adjusting the prompt to address those cases. For example, as S2R sentence identification is a classification task, two authors investigated the FP and FN of the GPT-4 responses to derive the classification criteria (Figure 3a) to better guide GPT-4 in the S2R sentence classification task. This process resulted in four versions of each type of prompt template. To determine if performance plateaued, we monitored the F1 score. For example, from version 3 to version 4 the F1 score decreased by 0.001 for the S2R identification task. Based on this minimal change, we selected version 3 as the optimal prompt for this phase.
- Few Shot (FS) prompting: starting from the obtained ZS template, we created a base FS template containing positive and negative examples selected from the remaining bugs of Saha *et al.*'s dataset [18] and the expected output. The example bug reports are representative of each task and selected based on certain criteria, *e.g.,* various bug types (crash, output, *etc.*), and bug reports with different wordings and structures. We iteratively executed, evaluated, and refined the template until the performance no longer improved, in the same way we did it in ZS prompting.

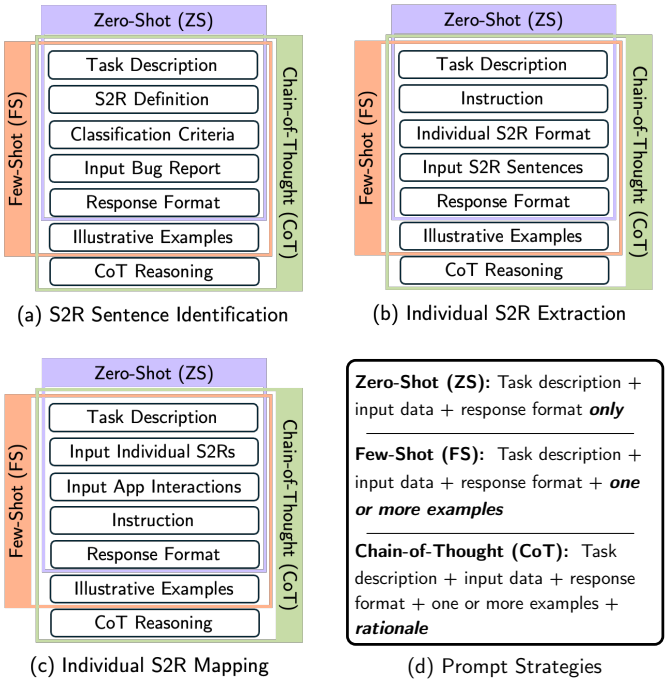

Fig. 3: Structure of the Developed Prompts

- Chain of Thought (CoT) prompting: starting from the obtained FS template, we created a base CoT template containing explanations for the outcome of the positive and negative examples. The explanation for the outcome was designed by two authors after discussion and reaching a consensus. We iteratively executed, evaluated, and refined the template until the performance plateaued, in the same way we did it in ZS and FS prompting.

This methodology resulted in three prompt templates (one from each prompting strategy) for S2R identification and three templates for individual S2R extraction. For S2R mapping, since we defined mapping as a 2-step task, we designed two prompts for each strategy, resulting in six prompts. The 2-step task consisted of first asking GPT-4 to return a yes/no answer on whether an individual S2R maps to the interactions of a given screen and if the answer is yes, asking GPT-4 to return the list of interactions that the S2R maps to. Our tests revealed that this approach led to less noisy answers from GPT-4, compared to executing only the second step. In total, we developed 12 prompt templates. To help visualize our prompt templates, Figure 3 illustrates the various components associated with the prompts for each task—our detailed templates are found in our replication package [13, 14].

C. Prompt Evaluation Results

We evaluated the prompt templates for S2R identification and extraction in terms of precision, recall, and F1 score, by executing these two phases in isolation. The F1 score was used to rank the templates. The S2R mapping prompt templates were evaluated by executing AstroBR's S2R quality assessment phase and evaluating the resulting S2R-interaction mappings. Since S2R mapping is a 2-step task, we evaluated each of the prompts based on the # and % of *hits*, defined as follows.

TABLE I: Prompt Template Performance for S2R Identification

Template	Precision	Recall	F1	#TP	#FP	#FN
ZS	0.929	0.968	0.948	183	14	6
FS	0.897	0.963	0.929	182	21	7
CoT	0.915	0.963	0.938	182	17	7

TABLE II: Prompt Template Performance for S2R Extraction

Template	Precision	Recall	F1	#TP	#FP	#FN
ZS	0.918	0.951	0.934	234	21	12
FS	0.897	0.951	0.923	234	27	12
CoT	0.810	0.951	0.875	234	55	12

TABLE III: Prompt Performance for S2R-Interaction Mapping

Template	1st-step template			2nd-step template	
	# Predictions	# Hits	Hit Rate	# Hits	Hit Rate
ZS	939	887	94.5%	30	76.9%
FS	970	912	94.0%	26	66.7%
CoT	1214	1152	94.9%	18	46.2%

For the first prompt, it is the number (and proportion) of correct predictions for the presence or absence of an S2R-interaction mapping in a given screen (out of the total number of predictions). For the second prompt, it is the number (and proportion) of correctly identified interactions for each individual S2R (out of the total number of individual S2Rs).

Tables I to III show the performance of the designed prompt templates for the three tasks: S2R identification, individual S2R extraction, and S2R mapping. Among the three templates for the S2R identification task, *ZS* achieved the best performance across the three metrics having the lowest # of FP (14) and FN (6). Likewise, for the individual S2R extraction task, the *ZS* template achieved the highest precision (0.918) with the lowest # of FP (21), sharing the same # of FN (12) with the other two prompts. Regarding the S2R mapping task, AstroBR with all three templates for the 1st-step prompt achieved a similar hit rate (94.0% to 94.9%) and with *ZS* template for the 2nd-step prompt achieved the best hit rate of 76.9%.

Interestingly, although prior research has shown the superiority of *CoT* prompts over *ZS* and *FS* prompts [15, 26], this is not the case for our tasks. Via qualitative analysis of GPT-4 responses, we observed that GPT-4 with *FS* and *CoT* prompts tends to include more unintended text in the responses compared to *ZS* prompt which results in more false positives, *e.g., CoT* template for S2R extraction generated 55 FPs while *ZS* template generated 21 FPs only. We conjecture that the long and complicated input (*e.g.,* bug reports can be long, and interaction information can be complicated) made the task difficult for GPT-4. Moreover, having three or four examples with reasoning made the prompts even longer.

As for all three tasks, *ZS* templates outperformed the other two, we utilized the *ZS* templates for implementing AstroBR.

V. AstroBR's Evaluation Design

AstroBR's evaluation has two main goals: (i) to evaluate AstroBR's ability to provide correct quality annotations for real bug reports, and (ii) to examine how well AstroBR can infer missing S2R information in bug reports. We apply AstroBR to a test dataset (see Section V-A) comprising 21 bug reports, in order to provide a comparison with prior work. We aim to answer the following research questions (RQs):

- **RQ$_1$:** How effective is AstroBR in generating correct S2R quality annotations?
- **RQ$_2$:** How accurately can AstroBR infer missing S2Rs?

A. Evaluation Dataset

We used the bug reports (*i.e., test set*) used by Chaparro *et al.* [7], which allow us to provide a direct comparison with their approach, Euler. This dataset contains 24 bug reports of various kinds (crashes, UI problems, and navigation problems) from six Android applications of different domains (web browsing, WiFi network diagnosis, finance tracking, *etc.*). The diverse evaluation set, separate from the development set, enabled us to assess the generalizability of the developed prompts across different bug reports. We discarded three bug reports, as follows: (1) two bug reports [27, 28] from the Aard Dictionary App [29], because the app version 1.4.1 is unable to load its dictionary database, and (2) one bug from Time Tracker app [30], because we could not generate the execution model for this app as the bug report requires a rotation action which AstroBR does not support. Hence, our test set contains 21 bug reports from the original Euler dataset.

Since this dataset does not contain any ground truth information for evaluating AstroBR, we constructed the ground truth manually. We used the same methodology discussed in Sections IV-A2 and IV-A3 to do so for identifying S2R sentences and extracting individual S2Rs.

To construct the quality assessment ground truth, the first two authors mapped the extracted individual S2Rs to GUI interactions manually following the methodology discussed in Section IV-A4. App execution models for the bug reports were built by parsing execution traces collected via CrashScope's app exploration and manual app usage. One author identified the reproduction interactions on the generated data and mapped such interactions with the extracted individual S2Rs from the bug report. They collected the mapped interactions for each individual S2R, as well as the interactions that are required to reproduce the bug, but not reported in the bug report, *i.e.,* ground truth for missing steps. Each individual S2R was mapped with one or more interactions in the execution model path, as needed. Using the mapped interactions and the quality assessment model (discussed in Section II), they assigned quality labels to each individual S2R. A second author performed the same steps and validated the interactions in the reproduction scenario as well as the quality annotations. Disagreements were resolved via discussion.

In summary, we identified 73 S2R sentences out of the 275 sentences present in the 21 bug reports with a near-perfect agreement between the two authors (98.2% agreement rate and 0.88 Cohen's kappa [25]). From the 73 S2R sentences, we extracted 82 individual S2Rs with an agreement rate of 93.9% between the two authors. We discarded four individual S2Rs as they represent rotation operation and the current version of AstroBR does not support this operation. We assigned the

remaining 78 individual S2Rs quality annotations (*i.e.,* 70 S2Rs as CS, 7 S2Rs as AS, 1 S2R as VM, and 38 S2Rs as MS). We identified 158 missing interactions, *i.e.,* missing steps for the 38 MS positions (*i.e.,* S2Rs with filled-in missing interactions). For constructing the annotations ground truth, the two authors agreed on 90% of the cases. Cohen's kappa for individual S2R extraction and mapping is inapplicable since the labeling is not based on a discrete set of labels.

B. Baseline Approach

We considered EULER [7] as the baseline approach, which also aims to assess the quality of S2Rs in a bug report. It identifies the S2R sentences from a bug report using deep learning techniques (*e.g.,* CNN [31], Bi-LSTM [32]). It identifies individual S2Rs via analysis of discourse patterns and assigns quality annotations by employing keyword-based mapping to app UI information. EULER and ASTROBR generate similar quality reports, therefore we can directly compare the ASTROBR reports to the original EULER reports provided by EULER's replication package [7], to answer the RQs.

C. Evaluation Methodology

We executed ASTROBR with the 21 bug reports on the test set, producing the quality report for each bug report, including the quality annotations and missing steps. To answer $\mathbf{RQ}_1$, we compared the ASTROBR assigned quality annotations with the ground truth quality annotations. To answer $\mathbf{RQ}_2$, we evaluated the generated missing steps by ASTROBR against the ground truth missing steps. For both RQs, we computed precision, recall, and F1 score. We applied the same process for EULER and qualitatively analyzed the false positives (FP) and negatives (FN) to understand the limitations of both approaches.

VI. RESULTS

A. RQ_1: Quality Annotation Results

We compared all the quality annotations in the ground truth with those produced by both ASTROBR and EULER. EULER fails to identify four (among the 78) S2Rs from the ground truth, whereas ASTROBR fails to identify two S2Rs. We considered these in our analysis as they represent false negatives.

Table IV compares the performance of ASTROBR and EULER for each quality annotation and in aggregate across all annotations. Overall performance metrics were computed by summing the TPs, FPs, and FNs. Overall, ASTROBR outperforms EULER in S2R annotation by a relative improvement of 25.2% in terms of F1 score. The overall performance difference between ASTROBR and EULER are statistically significant according to the Wilcoxon test (p-values = 0.03, 0.004, and 0.005 for precision, recall, and F1 scores, respectively).

Both ASTROBR and EULER incorrectly labeled two S2Rs as CS. For example, the S2R *"Add a split"* from report #699 from GnuCash App [33] requires a user to perform *tap add split button* and *type split amount* interactions. However, GPT-4 identified only one mapped interaction for this S2R. EULER also incorrectly labeled it as CS because the matching algorithm used by EULER is too restrictive: it maps an S2R to one interaction even if multiple interactions exist on the screen. Moreover, EULER failed to annotate 16 S2Rs as CS, while ASTROBR failed to annotate four S2Rs as CS.

TABLE IV: Quality Annotation Results (ASTROBR vs. EULER)

QAs	Approach	Precision	Recall	F1	#TP	#FP	#FN
CS	EULER	0.964	0.771	0.857	54	2	16
	ASTROBR	0.971	0.943	0.957	66	2	4
AS	EULER	0.600	0.429	0.500	3	2	4
	ASTROBR	1.000	0.714	0.833	5	0	2
MS	EULER	0.600	0.553	0.575	21	14	17
	ASTROBR	0.750	0.789	0.769	30	10	8
VM	EULER	0.077	1.000	0.143	1	12	0
	ASTROBR	0.333	1.000	0.500	1	2	0
Overall	EULER	**0.725**	**0.681**	**0.702**	**79**	**30**	**37**
	ASTROBR	**0.879**	**0.879**	**0.879**	**102**	**14**	**14**

Furthermore, EULER incorrectly labeled S2Rs as AS for two S2Rs where only one mapped interaction exist, whereas ASTROBR never made such errors. For example, the individual S2R, *"Select an event"* from bug report #154 in schedule-campfahrplan [34] is annotated as AS by EULER because it incorrectly mapped to multiple actions (*e.g.,* long click or click). However, ASTROBR accurately annotated the S2R as CS. Overall, we observed this is because (i) GPT-4 is able to correctly identify whether the S2Rs refers to single or multiple interactions, and (ii) ASTROBR can map an S2R to multiple interactions on the current GUI screen when reporters combine steps or use generic verbs. Moreover, EULER failed to annotate four S2Rs as AS while ASTROBR failed to annotate two S2Rs as AS.

Additionally, the vocabulary mismatch (VM) annotation is typically assigned when interactions are not present in the GUIs. EULER produced 12 false positive VMs due to low graph coverage and its inability to infer the actual step in the app, even with the capability of adding screens/interactions to the graph while executing S2Rs (see EULER's paper for details [7]). In contrast, ASTROBR incorrectly annotated two S2Rs only as VM. The reasons for EULER failing more than ASTROBR can be attributed to the restrictive matching algorithm EULER uses, whereas ASTROBR leverages GPT-4 to map interactions to S2Rs. It is also possible that EULER could not cover the necessary screens even after random exploration, whereas ASTROBR utilizes a more complete graph in mapping interactions for S2Rs that traverse many GUI app screens.

There can be one or multiple missing steps between two consecutive mapped interactions for two S2Rs. In such cases, we annotated the latter S2R as MS for analysis purposes. In EULER's 14 misclassified MS annotations, the suggested missing steps were not necessary for bug reproduction. In contrast, out of the ten individual S2Rs misclassified as MS by ASTROBR, four involved completely unnecessary steps for bug reproduction, and the remaining six were due to the incorrectly mapped interactions. For example, the S2R, *"Select Units"* from bug report #12 from DroidWeight [35] could not map to a GUI interaction because the corresponding GUI component description is "back modal" and that GUI interaction was considered a missing step. This leads to the necessity for developers to use meaningful GUI descriptions in the underlying code. A possible strategy to address this

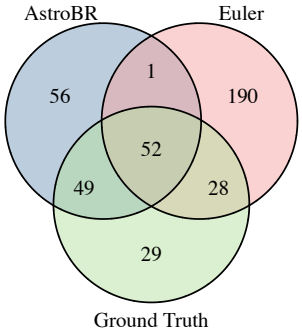

Fig. 4: # of Missing Steps Generated by AstroBR

issue is to use GUI interaction images in the prompt to GPT-4 to provide additional context. Moreover, AstroBR failed to identify MS annotations for 8 S2Rs whereas Euler failed in 17 S2Rs, meaning AstroBR is more successful in identifying intermediate paths between two S2Rs.

B. RQ_2: Missing S2R Results

In addition to evaluating Euler and AstroBR's ability to assign MS annotations, we also calculated the number of intermediate steps missing between two consecutive S2Rs. The ground truth dataset consists of 158 missing steps (7.5 per bug report on average) across 21 bug reports. Euler suggested 271 missing steps (12.9 missing steps per bug report on average), and 80 of those steps are present in the ground truth, achieving a precision of 0.295, a recall of 0.506, and an F1 score of 0.373. In contrast, AstroBR suggested 158 missing steps (7.5 per bug report on average), 101 of which are in the ground truth, resulting in a precision, recall, and F1 score of 0.639. This means that AstroBR outperforms Euler by 71.4% in terms of F1 score; AstroBR not only produces fewer incorrect, missing steps but also identifies more accurate missing steps.

We also analyzed if the missing steps provided by AstroBR are different compared to Euler, presented in Figure 4.

Both Euler and AstroBR successfully identified 52 correct missing steps, which represents 33% of the missing steps in the ground truth. These steps are primarily the setup steps commonly performed in the apps; however, reporters often forgo describing such steps in bug reports. For example, nine bug reports from the Gnucash app [36] require *"click the next button"* as the initial setup steps. In addition, both approaches produced one unnecessary missing step because of app traversal in an incorrect path. Moreover, both AstroBR and Euler failed to identify 29 crucial steps that reporters often ignore. These types of steps include: (i) actions in a popup dialog (*e.g., "Click OK button"*), and (ii) implicit steps that are easy to understand for humans but challenging for an automated tool to infer. For example, the S2R, *"Click on existing transaction"* from Gnucash's bug report #699 [33] requires the existence of a transaction in the app before clicking on the GUI component corresponding to the transaction, but that S2R is not explicitly described in the bug report.

Euler identified 28 ground truth missing steps that AstroBR failed to detect. This occurred for two main reasons. First, AstroBR would sometimes perform incorrect mapping between S2Rs and GUI interactions due to choosing an alternate path to reach from the current screen to a mapped screen, thus skipping crucial steps. For example, the S2R *"Navigate to Service Intervals screen"* from android-mileage's bug report #65 [37] was incorrectly mapped to *"click the ok button"*, which lead to omitting the necessary step *"open the app menu"*. Second, Euler identifies more initial setup steps required on a screen to proceed from the current GUI screen to the next due to its random exploration; however, AstroBR often ignores such steps, as it identifies the minimal steps necessary to perform on the current screen. To resolve this problem, future versions of AstroBR may incorporate knowledge of specific app functionalities to suggest missing steps in bug reports.

Across 21 bug reports, Euler produces 190 unnecessary steps that are not detected by AstroBR, whereas AstroBR only produces 56 such steps, excluding the ones detected by Euler. Although Euler prioritizes recall over precision to ensure the presence of necessary missing steps from which reporters can choose, the large number of missing steps may confuse developers when reproducing the bug. The reason for the large number of unnecessary steps is attributed to the incorrect S2R interaction matches in the GUIs. However, Euler's random exploration strategy in the graph exacerbates this problem by producing an excessive number of unnecessary steps.

AstroBR correctly identified 49 missing steps in the ground truth test dataset that Euler failed to detect. There are primarily two reasons: (i) Euler includes *"Open App"* step only if explicitly described in the bug report, yet only two out of 21 bug reports contain this step—AstroBR includes this step in all quality reports; and (ii) Euler's graph prevents the identification of GUI information for navigation components such as the navigation drawer and input widgets such as the spinner. For example, in Gnucash's bug report #615 [38], Euler identifies the step *"Tap the 'Navigation drawer opened' image button"* but fails to identify the interaction for the *"Manage Books"* that becomes visible when the navigation drawer is opened. The graph used in AstroBR correctly identifies such interaction, exhibiting its ability to handle complicated GUI structures.

VII. Related Work

Researchers have investigated bug reports for a variety of purposes including bug report management [18, 39–41], understanding bug resolution [42], predicting bug priority and severity [43–45], categorizing bug types [46, 47], identifying duplicate bugs [48–53], reproducing bugs [8, 15, 54–58], and localizing buggy code [59–62]. We discuss the most closely related work in this section.

Assessing Bug Report Quality. Past research in assessing the quality of bug reports is primarily focused on the readability, coherence, and inclusion of the necessary components within bug reports. Zimmermann [1] proposed an approach to assess the quality of bug reports by classifying them as bad, neutral, or good, considering various features such as keyword completeness, patches, screenshots, and readability. Dit *et al.* [63] evaluates the quality of bug reports based on the

coherence of comments in bug report discussions. Linstead *et al.* [64] later proposed a different textual coherence calculation technique, utilizing an information-theory-based approach by measuring the entropy of the distribution of latent topics in bug reports. Very recently, Bo *et al.* introduced ChatBR [6], which assesses and generates S2Rs if absent but does not evaluate generated S2R quality. AstroBR advances upon ChatBR by assessing S2Rs using annotations by determining whether S2Rs can be mapped to application UI interactions.

Chaparro *et al.* [7] introduced Euler, an approach that provides quality annotation for the S2Rs in bug reports. Euler is the closest related work, in as much as they produce the same type of quality reports, given a bug description. Unlike Euler, AstroBR uses LLMs to generate the quality reports. In consequence, its internals are fundamentally different, particularly in how the app model is explored. These improvements lead to more effective quality annotations as illustrated by the results of AstroBR's evaluation.

Automated Bug Reproduction. Researchers introduced various techniques to generate test cases for automated bug reproduction to diagnose, validate, and understand bugs. Fazzini *et al.* [8] developed Yakusu, which combines program analysis and text processing techniques to create test cases for bug reproduction. Zhao *et al.* [54] proposed ReCDroid, to reproduce crashes. ReCDroid formulates a dynamic ordered event tree (DOET) leveraging GUI components and event transitions, which aids in traversing GUIs for a given app and prioritizes relevant GUI components for exploration. Feng and Chen [15] introduced AdbGPT, which focuses on automatically reproducing bugs using LLMs. AdbGPT extracts actions and objects from S2Rs through prompt engineering and later leverages GUI encoding and LLMs to replay bugs within app screens. Wang *et al.* [55] proposed ReBL that mitigates different limitations of AdbGPT and utilizes the entire bug report instead of only using S2Rs to improve the contextual reasoning of the LLMs in automatically reproducing bugs.

Interactive Bug Reporting Systems. Researchers have proposed systems for interactive bug reporting, which typically aim also at improving the quality of the bug reports. Moran *et al.* proposed Fusion [10] that allows reporters to choose available actions and GUI components from dropdown lists, resulting in more structured and comprehensive bug reports. Fazzini *et al.* proposed EBug [65] that extends Fusion and suggests potential S2Rs to the reporters alongside the dropdown lists available to Fusion. Song *et al.* proposed Burt [9, 17], a chatbot that guides the reported and verifies the quality of bug information in real-time, providing suggestions to the reporters.

VIII. Threats to Validity and Limitations

Construct Validity. The main threats to construct validity stem from manually verifying the matching of the interactions extracted from the S2R sentences to the information on the execution model and constructing a ground truth dataset. To mitigate this threat, two authors independently carried out the manual verification tasks and ground truth creation, following well-defined and replicable methodologies. More so, we computed and reported agreement levels, which are very high in all cases.

Internal Validity. Selecting the optimal prompt can be challenging for any use of GPT-4, let alone for multiple distinct tasks, and this process of finding the best prompt impacts the performance of our approach. We selected the best prompt by evaluating 14 prompt templates, using three prompting strategies (*i.e.,* zero-shot, few-shot, and chain-of-thought) on a rich development set of bug reports from multiple applications.

External Validity. Our results are compared with the state-of-the-art, Euler, where we used 21 bug reports from their original dataset across six applications. We could not increase the dataset size for comparison due to difficulties in running the Euler tool. However, our approach is built by analyzing a dataset with bug reports from nine different applications consisting of four types of bugs. Therefore, AstroBR can be generalized to diverse types of bug reports. Moreover, AstroBR currently supports the most frequently used GUI interactions in Android applications (tap, long tap, *etc.*). While the lack of support for certain types of interactions (*e.g.,* rotation) is a limitation, this is not due to the inherent design of the approach, and the support of these features can be added through additional engineering effort in future work.

Limitations. AstroBR's performance depends on the completeness of the app execution model. The automated execution information collected with CrashScope may result in an incomplete execution model. To overcome this issue, we collected information from manual app executions.

IX. Conclusions and Future Work

Providing quality feedback to bug reporters at reporting time promises to result in bug reports that are easier to understand and reproduce by developers. We found that using LLMs (*i.e.,* GPT-4) for automatically extracting and analyzing S2Rs from natural language bug reports, and matching them to GUI interactions, is very effective, resulting in better quality annotations than state-of-the-art approaches. As with any other applications of LLMs, their performance is highly dependent on the prompting quality. By investigating 12 prompt templates, using different prompting strategies, we observed that the use of GPT-4 in this context is quite robust with respect to the type of prompt used. That is, different prompt templates lead to similar performance levels.

Future work will focus on expanding the evaluation to larger data sets and tackle the quality of other elements of bug reports, such as, the observed and expected behavior. We will also add support for additional types of interactions (*e.g.,* rotation) and perform a user study to assess AstroBR's usability by engineers in real-world scenarios. In addition, we will compare multiple LLMs in a follow-up extension of this work.

Acknowledgments

This work is supported by U.S. NSF grants CCF-1955853, CCF-2343057, and CCF-2441355. The opinions, findings, and conclusions expressed in this paper are those of the authors and do not necessarily reflect the sponsors' opinions.

References

[1] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schroter, and C. Weiss, "What makes a good bug report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[2] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, p. 308–318.

[3] M. Erfani Joorabchi, M. Mirzaaghaei, and A. Mesbah, "Works for Me! Characterizing Non-reproducible Bug Reports," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'14)*, 2014, pp. 62–71.

[4] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 495–504.

[5] T. Zimmermann, N. Nagappan, P. J. Guo, and B. Murphy, "Characterizing and predicting which bugs get reopened," in *Proceedings of the International Conference on Software Engineering (ICSE'12)*, 2012, pp. 1074–1083.

[6] L. Bo, W. Ji, X. Sun, T. Zhang, X. Wu, and Y. Wei, "Chatbr: Automated assessment and improvement of bug report quality using chatgpt," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, p. 1472–1483.

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

[8] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA'18)*, 2018, pp. 141–152.

[9] Y. Song, J. Mahmud, N. De Silva, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "Burt: A chatbot for interactive bug reporting," in *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering (ICSE'23)*, 2023, pp. 170–174.

[10] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing Bug Reports for Android Applications," in *Proceedings of the Joint Meeting on Foundations of Software Engineering (FSE'15)*, 2015, pp. 673–686.

[11] Y. Song and O. Chaparro, "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*, 2020, pp. 1551–1555.

[12] "Chatgpt - https://chat.openai.com," 2024.

[13] "Online replication package," https://github.com/sea-lab-wm/AstroBR-Bug-Reproduction-Steps-Assessment, 2025.

[14] "Doi," https://doi.org/10.5281/zenodo.14822745, 2025.

[15] S. Feng and C. Chen, "Prompting is all you need: Automated android bug replay with large language models," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[16] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The stanford corenlp natural language processing toolkit," in *Proceedings of 52nd Annual Mmeeting of the Association for Computational Linguistics: system demonstrations*, 2014, pp. 55–60.

[17] Y. Song, J. Mahmud, Y. Zhou, O. Chaparro, K. Moran, A. Marcus, and D. Poshyvanyk, "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*, 2022, pp. 344–356.

[18] A. Saha, Y. Song, J. Mahmud, Y. Zhou, K. Moran, and O. Chaparro, "Toward the automated localization of buggy mobile app uis from bug descriptions," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1249–1261.

[19] K. Moran, M. Linares-Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2016, pp. 33–44.

[20] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 15–18.

[21] M. Du, S. Yu, C. Fang, T. Li, H. Zhang, and Z. Chen, "Semcluster: a semi-supervised clustering tool for crowdsourced test reports with deep image understanding," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 1756–1759.

[22] T. Wendland, J. Sun, J. Mahmud, S. Mansur, S. Huang, K. Moran, J. Rubin, and M. Fazzini, "Andror2: A dataset of manually-reproduced bug reports for android apps," *Proceedings of the 18th IEEE/ACM International Conference on Mining Software Repositories (MSR'21)*, pp. 600–604, 2021.

[23] J. Johnson, J. Mahmud, T. Wendland, K. Moran, J. Rubin, and M. Fazzini, "An empirical investigation into the reproduction of bug reports for android apps," *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'22)*, 2022.

[24] O. Chaparro, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting missing information in bug descriptions," in *Proceedings of the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*, 2017, pp. 396–407.

[25] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and Psychological Measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[26] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, 2024.

[27] "Aard dictionary bug id 81 - https://github.com/aarddict/android/issues/81," 2024.

[28] "Aard dictionary bug id 104 - https://github.com/aarddict/android/issues/104," 2024.

[29] "Aard dictionary app - https://play.google.com/store/apps/details?id=itkach.aard2&hl=en_US," 2024.

[30] "Atimetracker bug id 1 - https://github.com/netmackan/ATimeTracker/issues/1," 2024.

[31] K. O'Shea, "An introduction to convolutional neural networks," *arXiv preprint arXiv:1511.08458*, 2015.

[32] P. Zhou, W. Shi, J. Tian, Z. Qi, B. Li, H. Hao, and B. Xu, "Attention-based bidirectional long short-term memory networks for relation classification," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (volume 2: Short papers)*, 2016, pp. 207–212.

[33] "Gnucash bug id 699 - https://github.com/codinguser/gnucash-android/issues/699," 2024.

[34] "Schedule bug id 154 - https://github.com/tuxmobil/CampFahrplan/issues/154," 2024.

[35] "Droidweight bug id 12 - https://github.com/sspieser/droidweight/issues/12," 2024.

[36] "Gnucash mobile app - https://play.google.com/store/apps/details?id=com.nicktylah.gnucash_mobile&hl=en_US," 2024.

[37] "Mileage bug id 65 - https://github.com/evancharlton/android-mileage/issues//65," 2024.

[38] "Gnucash bug id 615 - https://github.com/codinguser/gnucash-android/issues/615," 2024.

[39] A. Adnan, A. Saha, and O. Chaparro, "Sprint: An assistant for issue report management," *Proceedings of the 22nd IEEE/ACM International Conference on Mining Software Repositories (MSR'25)*, 2025.

[40] W. Zou, D. Lo, Z. Chen, X. Xia, Y. Feng, and B. Xu, "How practitioners perceive automated bug report management techniques," *IEEE Transactions on Software Engineering*, vol. 46, no. 8, pp. 836–862, 2018.

[41] J. Mahmud, N. De Silva, S. A. Khan, S. H. Mostafavi, S. M. H. Mansur, O. Chaparro, A. A. Marcus, and K. Moran, "On using gui interaction data to improve text retrieval-based bug localization," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2024.

[42] A. Saha and O. Chaparro, "Decoding the issue resolution process in practice via issue report analysis: A case study of firefox," in *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering (ICSE'25)*, 2025.

[43] Q. Umer, H. Liu, and I. Illahi, "Cnn-based automatic prioritization of bug reports," *IEEE Transactions on Reliability*, vol. 69, no. 4, pp. 1341–1354, 2019.

[44] Y. Tian, D. Lo, X. Xia, and C. Sun, "Automated prediction of bug report priority using multi-factor analysis," *Empirical Software Engineering*, vol. 20, pp. 1354–1383, 2015.

[45] Z. Huang, Z. Shao, G. Fan, H. Yu, K. Yang, and Z. Zhou, "Bug report priority prediction using developer-oriented socio-technical features," in *Proceedings of the 13th Asia-Pacific symposium on internetware*, 2022, pp. 202–211.

[46] K. Somasundaram and G. C. Murphy, "Automatic categorization of bug reports using latent dirichlet allocation," in *Proceedings of the 5th India software engineering conference*, 2012, pp. 125–130.

[47] G. Catolino, F. Palomba, A. Zaidman, and F. Ferrucci, "Not all bugs are the same: Understanding, characterizing, and classifying bug types," *Journal of Systems and Software*, vol. 152, pp. 165–181, 2019.

[48] Y. Yan, N. Cooper, O. Chaparro, K. Moran, and D. Poshyvanyk, "Semantic gui scene learning and video alignment for detecting duplicate video-based bug reports," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[49] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *CIKM'12*, 2012, pp. 852–861.

[50] J. He, L. Xu, M. Yan, X. Xia, and Y. Lei, "Duplicate bug report detection using dual-channel convolutional neural networks," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 117–127.

[51] N. Cooper, C. Bernal-Cárdenas, O. Chaparro, K. Moran, and D. Poshyvanyk, "It takes two to tango: Combining visual and textual information for detecting duplicate video-based bug reports," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 957–969.

[52] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating queries for duplicate bug report detection," in *2019 IEEE 26th international conference on software analysis, evolution and reengineering (SANER)*. IEEE, 2019, pp. 218–229.

[53] O. Chaparro, J. M. Florez, and A. Marcus, "On the vocabulary agreement in software issue descriptions," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2016, pp. 448–452.

[54] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G.J. Halfond, "Recdroid: Automatically reproducing android application crashes from bug reports," in *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, 2019, pp. 128–139.

[55] D. Wang, Y. Zhao, S. Feng, Z. Zhang, W. G. J. Halfond, C. Chen, X. Sun, J. Shi, and T. Yu, "Feedback-driven automated whole bug report reproduction for android apps," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, p. 1048–1060.

[56] C. Bernal-Cárdenas, N. Cooper, K. Moran, O. Chaparro, A. Marcus, and D. Poshyvanyk, "Translating video recordings of mobile app usages into replayable scenarios," in *Proceedings of the ACM/IEEE 42nd international conference on software engineering*, 2020, pp. 309–321.

[57] C. Bernal-Cárdenas, N. Cooper, M. Havranek, K. Moran, O. Chaparro, D. Poshyvanyk, and A. Marcus, "Translating video recordings of complex mobile app ui gestures into replayable scenarios," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 1782–1803, 2023.

[58] M. Havranek, C. Bernal-Cárdenas, N. Cooper, O. Chaparro, D. Poshyvanyk, and K. Moran, "V2s: A tool for translating video recordings of mobile app usages into replayable scenarios," *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pp. 65–68, 2021.

[59] J. M. Florez, O. Chaparro, C. Treude, and A. Marcus, "Combining query reduction and expansion for text-retrieval-based bug localization," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 166–176.

[60] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *Empirical Software Engineering*, vol. 24, pp. 2947–3007, 2019.

[61] ——, "Using observed behavior to reformulate queries during text retrieval-based bug localization," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 376–387.

[62] O. Chaparro and A. Marcus, "On the reduction of verbose queries in text retrieval based software maintenance," in *Proceedings of the 38th International Conference on Software Engineering Companion*, 2016, pp. 716–718.

[63] B. Dit and A. Marcus, "Improving the readability of defect reports," in *Proceedings of the 2008 International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pp. 47–49.

[64] E. Linstead and P. Baldi, "Mining the coherence of gnome bug reports with statistical topic models," in *Proceedings of the 6th IEEE International Working Conference on Mining Software Repositories*, 2009, pp. 99–102.

[65] M. Fazzini, K. Moran, C. Bernal-Cárdenas, T. Wendland, A. Orso, and D. Poshyvanyk, "Enhancing mobile app bug reporting via real-time understanding of reproduction steps," *IEEE Transactions on Software Engineering*, vol. 49, no. 3, p. 1246–1272, Mar. 2023.