Enhancing Exploratory Testing by Large Language Model and Knowledge Graph

Yanqi Su
Australian National University
Australia
Yanqi.Su@anu.edu.au

Dianshu Liao
Jiangxi Normal University
China
dianshu.liao@jxnu.edu.cn

Zhenchang Xing*
Data61, CSIRO
Australia
Zhenchang.Xing@data61.csiro.au

Qing Huang†
Jiangxi Normal University
China
qh@jxnu.edu.cn

Mulong Xie
Data61, CSIRO
Australia
Mulong.Xie@data61.csiro.au

Qinghua Lu
Data61, CSIRO
Australia
qinghua.lu@data61.csiro.au

Xiwei Xu
Data61, CSIRO
Australia
xiwei.xu@data61.csiro.au

ABSTRACT

Exploratory testing leverages the tester's knowledge and creativity to design test cases for effectively uncovering system-level bugs from the end user's perspective. Researchers have worked on test scenario generation to support exploratory testing based on a system knowledge graph, enriched with scenario and oracle knowledge from bug reports. Nevertheless, the adoption of this approach is hindered by difficulties in handling bug reports of inconsistent quality and varied expression styles, along with the infeasibility of the generated test scenarios. To overcome these limitations, we utilize the superior natural language understanding (NLU) capabilities of Large Language Models (LLMs) to construct a System KG of User Tasks and Failures (SysKG-UTF). Leveraging the system and bug knowledge from the KG, along with the logical reasoning capabilities of LLMs, we generate test scenarios with high feasibility and coherence. Particularly, we design chain-of-thought (CoT) reasoning to extract human-like knowledge and logical reasoning from LLMs, simulating a developer's process of validating test scenario feasibility. Our evaluation shows that our approach significantly enhances the KG construction, particularly for bug reports with low quality. Furthermore, our approach generates test scenarios with high feasibility and coherence. The user study further proves the effectiveness of our generated test scenarios in supporting exploratory testing. Specifically, 8 participants find 36 bugs from 8 seed bugs in two hours using our test scenarios, a significant improvement over the 21 bugs found by the state-of-the-art baseline.

CCS CONCEPTS

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

KEYWORDS

Exploratory testing, Knowledge graph, AI chain, Prompt engineering

ACM Reference Format:
Yanqi Su, Dianshu Liao, Zhenchang Xing, Qing Huang, Mulong Xie, Qinghua Lu, and Xiwei Xu. 2024. Enhancing Exploratory Testing by Large Language Model and Knowledge Graph. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3597503.3639157

1 INTRODUCTION

Exploratory testing (ET), a widely practiced testing approach in software industry, utilizes the knowledge, experience, and creativity of testers to create diverse and variable tests uncovering unexpected bugs [4, 17, 31, 37]. In contrast, scripted testing [13–15, 25] aims to identify known bugs through repetitive execution of pre-designed test cases in mostly a mechanical manner. Thus, despite the significant focus on scripted testing and test automation in research, exploratory testing, as a complementary testing to scripted testing, is still indispensable in the software industry [13–15, 25]. For conducting effective exploratory testing, concrete principles and guidelines have been established [4, 17, 37]. A fundamental principle emphasizes that not scripting does not mean not preparing [17]. Hence, it's crucial for exploratory testers to gather, comprehend, and analyze vast information related to application domains, software products, and users.

Some studies have proven the effectiveness and efficiency of ET for system-level testing of interactive systems through GUI from an end user's perspective [13–15, 31, 32]. This work is dedicated to facilitating scenario-based exploratory testing, which enables testers

*Also with Australian National University.
†Corresponding author.

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04…$15.00
https://doi.org/10.1145/3597503.3639157

to design complex test scenarios to provoke failures. ET leverages the tester's personal knowledge of domain, system and users [14]. To enrich tester's personal knowledge, we aim at distilling the crowd knowledge of realistic usage scenarios and past failures. Bug reports consist of steps to reproduce (**S2Rs**), expected behaviors (**EBs**) and observed behaviors (**OBs**), including abundant user tasks and failures. Thus, we use bug reports as the data source. To further improve the efficiency of ET, usage scenarios and failures extracted from bug reports can be utilized for exploratory test scenario generation. With these test scenarios, testers are better equipped to conduct more effective ET, unveiling failures unforeseen or even hidden within complex usage scenarios.

Recent work [31, 32] utilizes traditional open information extraction methods to construct a system KG outlining user tasks and system behaviors from bug reports. It then applies rule-based heuristics to generate test scenarios by combining scenarios of relevant bugs based on the KG. However, this existing approach has significant limitations that impede its practical application and widespread adoption. First, the traditional open information extraction-based approach used for the system KG construction has difficulty in efficiently processing bug reports with variable quality and diverse expression styles. Thus, the KG encompasses only a fraction of valuable system and bug knowledge. Second, the rule-based test scenario generation approach lacks the capability to discern and filter out invalid test scenarios, which not only impacts the testing experience negatively, leading to inefficiencies but also consumes valuable testing time.

To construct a KG with more comprehensive system and bug knowledge, we utilize the impressive NLU capabilities of LLMs to process bug reports. Our KG construction involves three processes: section extraction (i.e., extracting S2Rs, EBs, and OBs from bug reports in Section 3.1.1), step splitting (i.e., splitting S2Rs into atomic steps in Section 3.1.2), and step clustering (i.e., using the fast clustering algorithm to group identical steps in Section 3.1.3). For section extraction and step splitting, we utilize prompt engineering for generating the desired output. All we require are a few representative examples to help LLMs in recognizing these tasks and a corresponding task query to describe the task request.

The next challenge lies in generating feasible test scenarios. We utilize both the system and bug knowledge encapsulated in the KG and the logical reasoning capabilities of LLMs. Based on the KG, we identify relevant bug reports that share user task steps and use relevant bug reports to generate all potential test scenarios by concatenating the user task workflows of relevant bugs at both scenario and step levels. We then employ LLMs to validate the feasibility of each scenario. This validation is carried out based on representative examples with intermediate reasoning and the task query that explains the process of assessing the feasibility of each scenario. Furthermore, for each feasible test scenario, we identify if any redundant or unnecessary steps are included. If such steps are found, our approach removes these steps to enhance the coherence of test scenarios. Unlike the straightforward tasks of section extraction and step splitting, the task of identifying test scenario feasibility requires complex logical reasoning. We engage the CoT reasoning to instruct LLMs to simulate a developer's process of validating test scenario feasibility.

As a proof-of-concept, we construct a KG based on 32,772 bug reports from Mozilla Firefox. We choose Firefox due to its complexity and open-source nature, which results in a variety of available bug reports with inconsistent quality and diverse presentation styles. Furthermore, Firefox supports a rich set of user features and is a mature and stable product, which ensures the diversity of the testing scope. We then carry out a comprehensive evaluation to assess the effectiveness of our approach. Even though over half of the bug reports have low quality, our approach still demonstrates high performance for the KG construction, especially for section extraction (Table 1) and step splitting (Table 2). Moreover, the test scenarios generated by our approach exhibit high feasibility and coherence (Table 4). To assess whether our generated test scenarios can support testers in conducting more effective ET, we carry out a user study involving 16 participants and 8 seed bugs. Each participant uses test scenarios derived from two seed bugs generated by our approach or the baseline, SoapOperaTG [31, 32], a state-of-the-art approach generating test scenarios for ET. 8 participants using our approach find 36 bugs in 2 hours, a significant improvement compared to 21 bugs found by using SoapOperaTG. The results suggest that our approach enhances testers' ability to uncover hidden failures more efficiently. The contributions of paper are as follows:

- The first work exploits LLMs into exploratory testing, paving the way for new opportunities in supporting exploratory testing.
- We propose a lightweight and robust approach to construct a KG with comprehensive system and bug knowledge from bug reports with inconsistent quality and diverse expression styles.
- We generate test scenarios with high feasibility and coherence by leveraging NLU and logistical reasoning capabilities of LLMs.
- The user study, which identifies real-world bugs, further showcases the effectiveness of our approach in supporting exploratory testing. Our replication package can be found here.

2 MOTIVATION

Recent work [31, 32] utilizes traditional open information extraction methods to construct a system KG of user tasks and system behaviors from bug reports. Then, it uses rule-based heuristics to generate soap opera test scenarios by combining scenarios of relevant bugs based on the system KG. However, the state-of-the-art approach, SoapOperaTG, faces notable limitations, hindering its practicality and adoption: the complexity of processing inconsistent bug reports and the infeasibility of generated test scenarios.

2.1 Complexity of Processing Inconsistent Bugs

Despite bug writing guidelines and research work [5, 29] emphasizes the explicit inclusion of S2Rs, EBs and OBs in bug description, only 13,943 out of 32,772 bug reports from Mozilla Bugzilla clearly provide these sections. Since SoapOperaTG [31, 32] uses regular expressions to extract these sections, it falls short in processing bug reports that lack clear corresponding sections. Therefore, a considerable number of unstructured bug reports remain unprocessed, greatly affecting the completeness of the KG. As Bug 1578873 demonstrated in Figure 2a, extracting information involves more than just recognizing explicitly labeled S2Rs sections. It requires a comprehensive understanding of the text. For instance, the text within green box describes the issue experienced in this bug, namely

"Tabbing gets stuck on the 'Create New Login' button... The 'Lockwise Support' link can only be accessed via keyboard by tabbing backwards...". This description constitutes the OBs of the bug. The text enclosed in the blue box, "With no previously saved logins", specifies the conditions under which this bug can be reproduced. Hence, even though it's located in the S2Rs section, it should be classified as a precondition. Moreover, there are no explicit EBs stated in the bug description. However, an understanding of the bug, especially its OBs, allows us to infer the EBs as "Tabbing should proceed smoothly through all page elements".

Beyond scenario extraction, it is also essential to split S2Rs section into individual steps. Especially for compound steps, we need to further split them into atomic steps, namely, each atomic step containing only a single User Interface (**UI**) operation. As shown in Figure 3a, Step 1 and 3 are compound steps. Given that previous work [31, 32] utilizes constituency parsing to generate parse trees, and then employs a rule-based heuristic algorithm to split compound sentences, it could not be flexibly and correctly applied to various step expressions. For instance, Step 1 includes the temporal conjunction 'after', which can be divided into two atomic steps: 'Restart the browser' and 'Set prerequisite preferences'. Considering the sequential order of these steps, 'Set prerequisite preferences' should precede. Step 3 uses '>' to express three consecutive atomic steps in one step, which can be divided into 'Open "System Settings".', 'Select "Ease of Access".' and 'Click on "Display" option.'. Such diverse step expressions are difficult to address using predefined patterns like previous work.

> The traditional open information extraction-based approach has difficulties in effectively processing bug reports due to their inconsistent quality and diverse expression styles. This results in the KG containing only a subset of valuable system and bug knowledge.

2.2 Infeasibility of Generated Test Scenarios

The previous work [31, 32] utilizes a very naive way to generate new test scenarios in both scenario and step levels, without any assurances for the feasibility of these scenarios.

Scenario-level test scenarios are generated by directly linking the steps of two relevant bug reports. As shown in Figure 4b, Bug 1678633 and Bug 1575516 result in two scenario-level test scenarios (S2Rs_0 and S2Rs_1) by linking steps from Bug 1678633 and Bug 1575516, and vice versa. In S2Rs_0, the steps for Bug 1678633 involve signing out from Firefox Sync, deleting all data, returning to the 'about:logins' page, and observing the Login Item area. The steps for Bug 1575516 involve editing a saved login on the 'about:logins' page and logging into the selected login's website. As Bug 1678633's steps delete all data and therefore leaves no saved login available for Bug 1575516's steps to edit, S2Rs_0 is unfeasible for execution. As for S2Rs_1, executing Bug 1575516's steps (edit a saved login and log into it) do not hinder the execution of Bug 1678633's steps (sign out and delete all data). Thus, S2Rs_1 is feasible. As depicted in the green box in Figure 4b, the final step of Bug 1575516 is 'Focus back to the "about:logins" page,' and the initial step of Bug 1678633 is 'Navigate to the "about:logins" page.' Given that both steps involve accessing 'about:logins' page, they are repetitive. Thus, S2Rs_1 could be further optimized by removing the latter step.

Step-level test scenarios are generated by the shared steps between relevant bug reports. As depicted in Figure 5b, the shared

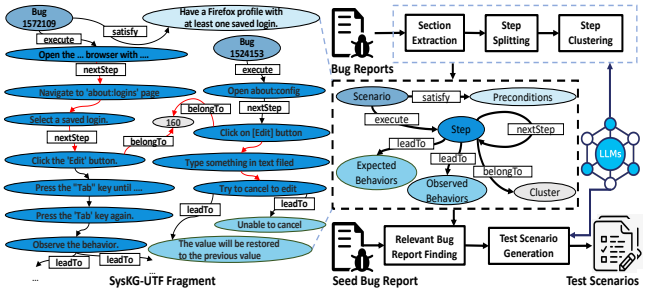

Figure 1: Approach Overview

step between Bug 1524153 and Bug 1572109 is 'Click on "Edit" button', because both bugs have this step. Subsequently, two step-level test scenarios are generated: S2Rs_0 and S2Rs_1. S2Rs_0 is formed by combining the steps before the shared step from Bug 1524153, the shared step, and the steps following the shared step from Bug 1572109. Conversely, S2Rs_1 is created by replacing Bug 1524153 and Bug 1572109. For S2Rs_0, "Create New Login" button from 'Press the Tab key until the "Create New Login" button is focused.' is specific on "about:logins" page. S2Rs_0 operating on the "about:config" page doesn't have this button, so infeasible. For S2Rs_1, after clicking the "Edit" button on "about:logins" page, it has text field for editing, so 'Type something in text filed.' and 'Try to cancel to edit.' can be executed. S2Rs_1 is feasible.

> The ability to identify invalid test scenarios not only requires text comprehension but logistic reasoning. The rule-based test scenario generation approach lacks the capability to discern and filter test scenarios, which negatively impacts the testing experience, causing inefficiencies and wasting valuable testing time.

3 APPROACH

As identified in Section 2, the existing approach [31, 32] has two major limitations: inability to handle diversified bug reports and infeasibility of generated test scenarios. To mitigate two limitations in the prior work, we propose a novel approach, which leverages the superior NLU and logistical reasoning capabilities of LLMs, derived from large-scale training corpora. As depicted in Figure 1, first, we construct a SysKG-UTF based on bug reports. Second, we generate test scenarios for supporting ET based on the KG. Unlike recent work [31, 32], which uses rule-based heuristics and traditional NLP for KG construction and test scenario generation, our approach simplifies this process by leveraging LLMs in both phases by novel prompt engineering.

3.1 KG Construction

Bug reports contain valuable information encapsulating system logic, features, interactions and failures, which is beneficial for test scenario generation to ET [31, 32]. To effectively use the information, we propose an approach to construct a KG.

3.1.1 Section Extraction. To leverage scenarios and oracles from bug reports for ET, we first extract specific sections (i.e., preconditions, S2Rs, EBs, and OBs) from bug reports. Specifically, we utilize the LLMs in recognizing and extracting these sections. To help LLMs extract specific sections efficiently, we provide representative examples for in-context learning. We take Bug 1578873 as an example in Figure 2. The input includes the bug report (bug summary and description in Figure 2a) and the query for the section

Bug Summary: Tabbing gets stuck on "Create New Login" when there are zero saved logins.
Bug Description: Created attachment 9090502 focusIsStuckOnButton.png Observed Behaviors
Tabbing gets stuck on the "Create New Login" button on the 9/4 build of Nightly when there are zero saved logins. The "Lockwise Support" link can only be accessed via keyboard by tabbing backwards through the page elements.

Steps to reproduce: Preconditions Steps To Reproduce
- With no previously saved logins, open `about:logins` in 9/4 build of Nightly and begin tabbing through page elements.

(a) To-Be-Processed Bug Report

Please split the bug description into specific sections and answer in the format of a JSON string:
{"PRECONDITIONS":["",],"STEPS_TO_REPRODUCE":["",],"EXPECTED_RESULTS":["",],"ACTUAL_RESULTS":["",]}

(b) Section Extraction Query

{"**PRECONDITIONS**": ["No previously saved logins"],
"**STEPS_TO_REPRODUCE**": ["Open about:logins",
"Start tabbing through page elements"],
"**EXPECTED_RESULTS**": ["Tabbing should proceed smoothly through all page elements"],
"**ACTUAL_RESULTS**": [
"Tabbing gets stuck on the 'Create New Login' button when there are zero saved logins",
"The 'Lockwise Support' link can only be accessed via keyboard by tabbing backwards"]}

(c) Section Extraction Output

Figure 2: Section Extraction Example

extraction (i.e., the specific request about the desired sections to be extracted in Figure 2b). The output in Figure 2c is the extracted sections. By providing such examples, the LLMs are better equipped to understand the task at hand and efficiently identify and extract the relevant sections in similar contexts.

In-context learning. Due to token limitations of LLMs for input context, it's crucial to select representative examples as input. The selection criterion aims to use as few examples as possible to cover the task complexities, thereby providing the LLMs a comprehensive understanding. The complexities of section extraction contain: a) not all sections having explicit tags, b) sections mistakenly listed under wrong tags, and c) the absence of certain sections. Based on this criterion, we have chosen three representative examples: Bug 1578873, Bug 1537640, and Bug 1552289. Both Bug 1578873 and Bug 1537640 have the explicit S2Rs tag. For Bug 1578873 in Figure 2a, the preconditions are erroneously listed under the S2Rs tag and thus need to be accurately extracted and identified. Any text outside the S2Rs tag needs to be properly identified into the corresponding sections. For Bug 1537640, the whole text is found under the S2Rs tag, requiring further identification and extraction. Bug 1552289 does not feature any tags, requiring careful extraction of the mixed sections. Notably, neither Bug 1578873 nor Bug 1537640 contain EBs in bug descriptions, but these can be inferred from the context, particularly from OBs. While Bug 1552289 does not include S2Rs, these also can be derived from the content detailed in the bug report. The derived S2Rs are 'Open DiscoveryStream.' and 'Check if the search-only view is shown.'.

By these representative examples, LLMs can understand how to handle bug reports with chaotic or even no structure. Furthermore, for sections that need to be extracted but aren't explicitly stated in bug reports, LLMs can generate logical inferences based on the content of bug report itself. Owing to the potent NLU capabilities and logical reasoning of LLMs, there is no need for sophisticated prompt engineering for section extraction. LLMs can understand and carry out this section inference merely by being presented with examples that reflect this situation.

Prompt construction. The prompt is structured by <Examples>, <To-Be-Processed Bug Report>, and <Section Extraction Query>. The format of <Examples> is shown in Figure 2. <To-Be-Processed Bug Report> includes the bug summary and description. <Section Extraction Query> is shown in Figure 2b, namely 'Please split the bug description into the specific sections and answer in the format of a JSON string: {"PRECONDITIONS":["",], ...}'. Owing to the robustness of LLMs, the prompt's sentence construction doesn't have to strictly adhere to grammatical norms [3]. Benefiting from few-shot learning, the LLMs consistently generate the extracted sections in a JSON format, aligning with the format used in our example output.

STEPS_TO_REPRODUCE:
1. Restart the browser after setting prerequisite preferences. → Set prerequisite preferences. / Restart the browser.
2. Dismiss the Make Firefox default message if it is displayed to trigger the Suggestions modal.
3. Open System Settings > Ease of Access > Display. → Open 'System Settings'. / Select 'Ease of Access'. / Click on 'Display' option.
4. Set the "Make text bigger" slider to a larger value (e.g. 135%).
5. Observe the modal.

(a) To-Be-Processed S2Rs

Please split STEPS_TO_REPRODUCE into steps, especially splitting the step with multiple UI operations into steps with one UI operation.

(b) Step Splitting Query

[{"**STEP**": "Set prerequisite preferences.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Restart the browser.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Dismiss the Make Firefox ... the Suggestions modal.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Open 'System Settings'.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Select 'Ease of Access'.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Click on 'Display' option.", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Set the 'Make text bigger' ... larger value (e.g. 135%).", "**STEP_TYPE**": "OPERATION"},
{"**STEP**": "Observe the modal.", "**STEP_TYPE**": "NON_OPERATION"}]

(c) Step Splitting Output

Figure 3: Step Splitting Example

3.1.2 Step Splitting. Upon extracting specific sections from bug reports, we further split the S2Rs into atomic steps. Each atomic step should only contain one UI operation. Step splitting is particularly vital for compound steps, as demonstrated by Step 1 and 3 in Figure 3a. Meanwhile, we also categorize each step as operation or non-operation type. Steps not altering the state of software under test, e.g., Step 5: 'Observe the modal.', are deemed non-operation steps. Conversely, steps that do bring changes in the system are identified as operation steps. In detail, we employ LLMs for this task. Representative examples are provided for in-context learning. An example of step splitting from Bug 1730692 is presented in Figure 3. The input comprises two parts: the S2Rs of bug report, and a query explicitly requesting step splitting. The output is a list of atomic steps, each labeled with their respective step types.

In-context learning. The task complexities encompass: a) decomposing compound steps including temporal or coordinating conjunctions, or common abbreviations, and b) distinguishing step type as operation or non-operation. We select two representative examples, Bug 1730692 and Bug 1556965 for few-shot learning. In Figure 3, Bug 1730692 offers an example for splitting compound sentences with temporal conjunctions (Step 1) and provides insight on how to split steps with common abbreviations (Step 3). Bug 1556965 provides two step with coordinating conjunctions, e.g., splitting 'Launch Firefox and enable the prefs' into 'Launch Firefox' and 'enable the prefs'. These examples encompass most types of compound steps, offering clear guidance on step splitting. Additionally, both bug reports include a non-operation step, i.e., Step 5 in Bug 1730692 and 'observe the Header' in Bug 1556965, demonstrating to the LLMs how to differentiate step types.

Prompt construction. The prompt consists of <Examples>, <To-Be-Processed S2Rs>, and <Step Splitting Query>. <Step Splitting Query> requests: 'Please split STEPS_TO_REPRODUCE into steps, especially splitting the step with multiple UI operations into steps with one UI operation.', shown in Figure 3b.

3.1.3 Step Clustering. Different bug reports may contain identical steps, although expressed in varied ways, such as "Click the 'Edit' button." and "Click on [Edit] button" from the SysKG-UTF fragment in Figure 1. By clustering these steps, we can link steps in the KG by the relation, i.e., *belongTo* the same cluster. Such steps are termed shared steps. Consequently, the scenarios extracted from bug reports within the KG aren't stand-alone entities. Instead, they are connected by shared steps, facilitating a more comprehensive and tightly connected KG of user tasks and failures.

The KG not only takes advantage of relevant bug report finding but also enhances the likelihood of generating test scenarios that simulate expert system usage and uncover unexpected failures. In detail, we identify relevant bug reports through shared steps rather than relying solely on duplicate or similar bug detection (Section 3.2.1). It ensures that relevant bug reports exhibit a certain degree of similarity (shared steps) while also presenting differences. As shown in the SysKG-UTF fragment in Figure 1, two relevant bug reports share the step "Click the 'Edit' button.", indicating their relevance to editing. However, the differences between them, such as occurring on different pages ('about:logins' and 'about:config') and focusing on different operations ("using the 'Tab' key to navigate" and "cancel edit command"), create opportunities to generate new and unexpected test scenarios. Of particular significance is the utilization of shared step connections within the KG (Section 3.2.2). Transitioning from the steps of Bug 1572109 to Bug 1524153 by the shared step results in the creation of a new test scenario, as demonstrated in Figure 5d. This process is exemplified by the red lines within the SysKG-UTF fragment in Figure 1.

To achieve this, we utilize a fast clustering algorithm based on SBERT sentence similarities. The algorithm clusters steps if their cosine similarity is equal to or greater than a certain threshold (set at 0.85, as [31]), and if the minimum community size is satisfied (set at 1, as [31]). To better capture the relationships among various entities, especially the intricate connections among steps across scenarios, we choose the KG over a traditional database for storage.

3.2 Test Scenario Generation

Considering limited context input of LLMs, it is impractical to input all bug reports directly into LLMs for test scenario generation. Therefore, we need to effectively synthesize the essential information for test scenario generation before feeding it into LLMs. For this, we construct a system KG that encapsulates all information for test scenario generation. Then, based on the shared steps in the KG, we identify relevant bug reports for each seed bug. Subsequently, the seed bug report and its relevant bug report are combined to generate all potential test scenarios. As highlighted in Section 2.2, the validity of bug reports does not inherently assure the test scenario feasibility they derive. Thus, we utilize LLMs to assess the feasibility of potential test scenarios, eliminating infeasible ones. This marks a clear distinction from SoapOperaTG, which overlooks the infeasibility of generated test scenarios.

3.2.1 Relevant Bug Report Finding. To find relevant bug reports for a seed bug report, we utilize shared steps between bug reports within the KG, which ensures a certain level of similarity while also accommodating differences, thereby fostering the generation of new and unforeseen test scenarios.

We consider both step co-occurrences and bug frequencies, referred to in the previous work [31]. We enhance step co-occurrence metric by focusing solely on operation-type steps from Section 3.1.2. Thus, step co-occurrence is a measure that calculates the number of shared operation-type steps between the seed bug report and other bug reports, with higher step co-occurrence implying greater relevance due to more overlapping UI operations. Bug frequency assesses the occurrence of steps from the same cluster across different bug reports. Steps common to many bug reports, such as "Open any website", have high bug frequency and thus are less indicative of specific bugs. In contrast, steps that primarily appear within a particular context, such as "Open about:config", have a low bug frequency, making them more representative of specific issues. When ranking bug report relevance, we prioritize according to step co-occurrence, sorting in descending order. If multiple bug reports share the same step co-occurrence, we then rank by their cumulative bug frequency in ascending order. Lastly, for bug reports sharing both step co-occurrence and total bug frequency, we sort them in reverse chronological order, assuming that the recent bug reports provide more valuable information for testing.

3.2.2 Test Scenario Generation. Given a seed bug report bug_s, we generate test scenarios in combination with each relevant bug report bug_r from the KG at both scenario and step level. Scenario-level test scenarios are generated by linking steps from bug_s with bug_r. We generate step-level test scenarios by transitioning steps from bug_s to bug_r using the shared steps in the KG, as exemplified by the red lines in the SysKG-UTF Fragment shown in Figure 1. By swapping bug_s and bug_r, an expanded set of test scenarios are generated.

Scenario-Level Test Scenario. First, we generate all potential S2Rs by connecting the steps from bug_s with the steps from bug_r, and vice versa. Second, we input all potential S2Rs into LLMs to assess their feasibility. Third, for each feasible S2Rs, LLMs further check if the S2Rs contain redundant or unnecessary steps within the connecting part of steps from two bug reports. If such steps are detected, S2Rs are refined by eliminating these steps. Finally, for each viable S2Rs, a complete scenario is generated by determining preconditions, EBs and OBs based on the S2Rs, bug_s, and bug_r. Figure 4 illustrates this process based on potential S2Rs from Bug 1678633 and Bug 1575516, followed by a request for scenario-level test scenario generation. Before generating the test scenario, a CoT analyzing the potential S2Rs' feasibility is provided. This entire process is fully automated. The complex procedure involved in the test scenario generation is distilled into the query that LLMs can easily understand and execute in Figure 4c. To further aid LLMs, in-context learning and CoTs are utilized by providing examples including potential test scenarios that are infeasible or contain redundant steps in Figure 4b.

In-context learning. Two bug pairs are selected as representative examples for this task, namely <Bug 1678633, Bug 1575516> and <Bug 1238444, Bug 1313079>. For <Bug 1678633, Bug 1575516>

Bug1678633_SCENARIO:
{**"SUMMARY":** "A blank page is wrongly displayed ... after signing out ... and ... delete all data",
"PRECONDITIONS": ["Signed in with a valid FxA account that contains at least one login saved", ...],
"STEPS_TO_REPRODUCE": ["Navigate to the 'about:logins' page.", ..., "Observe the Login Item area."],
"EXPECTED_RESULTS": ["All logins are successfully removed and the 'No FxA sync' state ... is displayed"],
"ACTUAL_RESULTS": ["All logins are successfully removed and no state is displayed ..."]}

Bug1575516_SCENARIO:
{**"SUMMARY":** "'Discard Unsaved Changes?' pop-up not displayed if 'Edit' mode of a saved login opened ...",
"PRECONDITIONS": ["Have at least one saved login"],
"STEPS_TO_REPRODUCE": ["Open the latest Nightly browser.", ... ,"Focus back to the 'about:logins' page."],
"EXPECTED_RESULTS": ["'Discard Unsaved Changes?' pop-up ... displayed", "The 'Edit' mode ... opened"],
"ACTUAL_RESULTS": ["The 'Edit' mode is dismissed", ...]}

(a) Bug Scenario Pair

Bug1678633 and Bug1575516 can generate 2 potential new S2Rs by linking steps from Bug1678633 and Bug1575516, as follows:
S2Rs_0:
GENERATED_METHOD: Steps from Bug1678633 + Steps from Bug1575516
STEPS_TO_REPRODUCE: [
"Navigate to the 'about:logins' page.",
"Go to the 'about:preferences#sync' page using a new tab.",
"Click on the 'Sign Out...' button.",
"Check the 'Delete data from this device' checkbox from the pop-up.",
"Click on the 'Sign Out' button from the pop-up.",
"Switch back to the 'about:logins' tab.",
"Observe the Login Item area.",
"Open the latest Nightly browser.",
"Navigate to 'about:logins' page.",
"Click on a saved login.",
"Click on the 'Edit' button.",
"Edit the username and password.",
"Click on the website address.",
"Log in.",
"Focus back to the 'about:logins' page."]

For S2Rs_0, the 1st highlighted step deletes data from the device. No saved login is available for the 2nd and 3rd highlighted step to click and edit, thus S2Rs_0 is infeasible

S2Rs_1:
GENERATED_METHOD: Steps from Bug1575516 + Steps from Bug1678633
STEPS_TO_REPRODUCE: [
"Open the latest Nightly browser.",
"Navigate to 'about:logins' page.",
"Click on a saved login.",
...
"Focus back to the 'about:logins' page.",
"Navigate to the 'about:logins' page.",
"Go to the 'about:preferences#sync' page using a new tab.",
...
"Observe the Login Item area."]

Repetitive Steps

(b) Potential S2Rs

Validate the feasibility of the potential STEPS_TO_REPRODUCE, details as follows:
For each potential set of steps to reproduce:
if StepsToReproduce is feasible:
if StepsInConnectingPart are redundant:
Remove one of the redundant steps
else:
if StepsInConnectingPart involve opening Firefox:
if No ClosingFirefoxStep before the OpeningFirefoxStep:
Remove OpeningFirefoxStep
Generate a complete Scenario for feasible StepsToReproduce
else:
Remove the infeasible StepsToReproduce

(c) Scenario-level Test Generation Query

{**"CHAINS_OF_THOUGHT"**: "Bug 1678633 is to sign out from Firefox Sync, delete all data, return to the 'about:logins' page, and observe the Login Item area. Bug1575516 is to edit a saved login on 'about:logins' page and log into the website of the selected login.
For S2Rs_0, since Bug 1678633 deletes all data, no saved login is available for Bug 1575516 to edit. Therefore, S2R_0 is infeasible.
For S2R_1, the execution of Bug 1575516 (edit a saved login and log into it) will not prevent Bug 1678633 (sign out and delete all data) from being executed. Hence, S2Rs_1 is feasible.
Then, we consider **whether the steps in the connecting part are redundant or unnecessary**. The last step of Bug 1575516 is [Focus back to the 'about:logins' page]. The first step of Bug 1678633 is [Navigate to the 'about:logins' page.]. Both of them are to about:logins page, so remove the first step of Bug 1678633.",
"SCENARIOS": [{
"PRECONDITIONS": ["Have at least one saved login"],
"STEPS_TO_REPRODUCE": ["Open the latest Nightly browser.",
"Navigate to 'about:logins' page.",
...
"Focus back to the 'about:logins' page.",
"Go to the 'about:preferences#sync' using a new tab.",
...
"Observe the Login Item area."],
"EXPECTED_RESULTS": ["The 'Discard Unsaved Changes?' pop-up is displayed.",
"The password and the username are not updated.",
"All logins are successfully removed and the "No FxA sync" state ... is displayed."],
"ACTUAL_RESULTS": ["All logins are successfully removed and no state is displayed for the 'about:logins' page."]}]}

Redundant Step is removed

(d) Scenario-level Test Generation Output

Figure 4: Scenario-Level Test Generation Example

displayed in Figure 4, two potential S2Rs are generated. One S2Rs is deemed infeasible, while the other is feasible but contains a redundant step, which is removed for refinement. For <Bug 1238444, Bug 1313079>, two feasible scenario-level test scenarios are generated. However, the S2Rs bridging steps from Bug 1313079 and Bug 1238444 include an unnecessary 'Open Firefox' step in the connection between the bug reports. This step is superfluous because, after executing the steps of Bug 1313079, Firefox remains open as the last action involves scrolling up the webpage, not closing Firefox.

Chain-of-thought reasoning. Unlike straightforward tasks such as section extraction and step splitting, the task of generating test scenarios involves validating their feasibility, which requires logical reasoning. Therefore, before providing an outcome. LLMs first engage in a CoT reasoning to enhance the reliability of the final results. Figure 4d illustrates an example of this CoT reasoning in scenario-level test scenario generation. First, LLMs analyze what these two bug reports aim to achieve individually for understanding the input bug pair. Second, for each potential S2Rs, LLMs determine whether there are conflicting steps that could impact the feasibility of the S2Rs. Finally, for the feasible S2Rs, LLMs further assess the coherence of steps, ensuring a streamlined and succinct process.

Prompt construction. The prompt incorporates <Examples>, <Bug Scenario Pair>, <Potential S2Rs>, and <Scenario-level Test Generation Query>. Notably, <Examples> involve a CoT reasoning process designed to guide the LLMs in executing logical reasoning before giving an outcome. <Bug Scenario Pair> refers to the scenarios of bug_s and bug_r. Each one is comprised of the bug summary and four sections extracted in Section 3.1.1. <Bug Report Pair> can guide the LLMs in generating a comprehensive scenario for the feasible S2Rs based on the preconditions, EBs, and OBs from the bug pair. Given the complexity of the task, we present the <Scenario-level Test Generation Query> in a pseudocode format, as illustrated in Figure 4c. This structured format provides a straightforward and intuitive explanation of our task requirements, simplifying the comprehension process for the LLMs.

Step-Level Test Scenario. First, we identify all potential S2Rs at the step-level based on bug_s and bug_r. This involves finding all the shared steps between two bugs. For each shared step, denoted as $step_{share}$, we generate step-level test scenarios by combining the steps preceding $step_{share}$ from bug_s, $step_{share}$ itself, and the steps subsequent to $step_{share}$ from bug_r, and vice versa. Second, we eliminate redundant S2Rs by referring to the cluster list associated with the steps. If multiple S2Rs have identical cluster lists, we retain only one of them. Moreover, any S2Rs that replicate the cluster list of either bug_s or bug_r are dismissed. Third, we input filtered potential S2Rs into LLMs for feasibility assessment. Finally, for each feasible S2Rs, we generate a comprehensive scenario by determining preconditions, EBs, and OBs based on S2Rs, bug_s, and bug_r. Figure 5 provides an example of step-level test generation using Bug 1524153 and Bug 1572109. Similar to scenario-level test generation, a CoT analysis for potential S2Rs' feasibility is provided before the generation of the test scenario.

In-context learning. Two bug pairs are selected as representative examples for this task, namely <Bug 1524153, Bug 1572109> and <Bug 1483786, Bug 1571414>. Each of these pairs generates one feasible and one unfeasible test scenario. Take <Bug 1524153, Bug 1572109> as an example, shown in Figure 5.

Bug1524153_SCENARIO:
{"SUMMARY": "cannot cancel edit command in new preferences",
"PRECONDITIONS": [],
"STEPS_TO_REPRODUCE": ["Open about:config", ... , "Try to cancel the edit"],
"EXPECTED_RESULTS": ["The value should be restored to the previous value"],
"ACTUAL_RESULTS": ["Unable to cancel the edit"]}

Bug1572109_SCENARIO:
{"SUMMARY": "The focus remains blocked on the 'Create New Login' button when using the 'Tab' key to navigate if the 'Edit/Create' mode is opened",
"PRECONDITIONS": ["Have a Firefox profile with at least one saved login."],
"STEPS_TO_REPRODUCE": ["Open the latest Nightly browser ...", ..., "Observe the behavior."],
"EXPECTED_RESULTS": ["The next element from the page should be focused."],
"ACTUAL_RESULTS": ["The focus remains blocked on the 'Create New Login' button."]}

(a) Bug Scenario Pair

Bug1524153 and Bug1572109 can generate 2 potential new S2Rs based on the shared step:
S2Rs_0:
GENERATED_METHOD: "The shared STEP is 'Click on [Edit] button'.
New S2Rs: STEP 1 from Bug1524153 + shared STEP + STEP 5 – 7 from Bug1572109."
STEPS_TO_REPRODUCE: ["Open about:config.",
"Click on [Edit] button.",
"Press the 'Tab' key until the 'Create New Login' button is focused.",
"Press the 'Tab' key again.",
"Observe the behavior."]
S2Rs_1:
GENERATED_METHOD: "The shared STEP is 'Click the 'Edit' button'.
New S2Rs: STEP 1 - 3 from Bug1572109 + shared STEP + STEP 3 - 4 from Bug1524153."
STEPS_TO_REPRODUCE: ["Open the latest Nightly browser with the profile from ...",
"Navigate to 'about:logins' page.",
"Select a saved login.",
"Click the 'Edit' button.",
"Type something in text field.",
"Try to cancel the edit."]

For S2Rs_0, 'Create New Login' button from the 2nd highlighted step is specific on 'about:logins' page. S2Rs_0 happening on 'about:config' page from the 1st highlighted step doesn't have this button, so infeasible.

(b) Potential S2Rs

Validate the feasibility of each potential S2Rs. For each feasible S2Rs, generate a complete SCENARIO.

(c) Step-level Test Generation Query

{"CHAINS_OF_THOUGHT": ["For S2Rs_0, 'Create New Login' button from STEP 'Press the Tab key until the Create New Login button is focused.' is specific on 'about:logins' page. S2Rs_0 happening on 'about:config' page doesn't have this button, so S2Rs_0 is not feasible.",
"For S2Rs_1, after clicking the 'Edit' button on 'about:logins' page, it has text field for editing, so 'Type something in text filed.' and 'Try to cancel to edit.' can be executed. So, S2Rs_1 is feasible"],
"SCENARIOS": [{
"PRECONDITIONS": ["Have a Firefox profile with at least one saved login."],
"STEPS_TO_REPRODUCE": ["Open the latest Nightly browser ... from prerequisites.",
"Navigate to 'about:logins' page.",
"Select a saved login.",
"Click the 'Edit' button.",
"Type something in text field",
"Try to cancel the edit"],
"EXPECTED_RESULTS": ["The value will be restored to the previous value."],
"ACTUAL_RESULTS": ["Unable to cancel."]}]}

(d) Step-level Test Generation Output

Figure 5: Step-Level Test Generation Example

Chain-of-thought reasoning. Similar to scenario-level test scenario generation, step-level test scenario generation also needs to assess the feasibility of potential S2Rs. Thus, CoT reasoning needs to be processed before giving the outcome. In particular, for each S2Rs, LLMs need to determine whether certain steps are executable based on the preceding steps. Take S2Rs_0 in Figure 5 as an example. The step to focus on the 'Create New Login' button is deemed unfeasible because the current page, as established by previous steps, is 'about:config', which doesn't have this button.

Prompt construction. The prompt is comprised of <Examples>, <Bug Scenario Pair>, <Potential S2Rs>, and <Step-level Test Generation Query>. Significantly, <Examples> incorporate a CoT reasoning process, guiding LLMs to perform logical reasoning prior to delivering an outcome. <Bug Scenario Pair> guides LLMs in generating a complete scenario for each feasible S2Rs based on the bug pair. <Step-level Test Generation Query> instructs the model to 'Validate the feasibility of each potential S2R. For each feasible S2R, generate a complete SCENARIO.'.

4 EVALUATION

This section evaluates the quality of the KG and generated test scenarios, and their effectiveness in supporting exploratory testing by answering the following research questions:

RQ1: What is the quality of SysKG-UTF?
RQ2: What is the quality of generated test scenarios?
RQ3: How useful are test scenarios for exploratory testing?

4.1 Experimental Setup

4.1.1 Dataset. We crawl 53,022 bug reports created from January, 1st, 2016 to June, 10th, 2023, from Mozilla Bugzilla. We filter these bug reports based on their status and bug description. First, we only retain bug reports with the status marked as closed, resolved, or verified, ensuring the feasibility of S2Rs within these reports. We then discard bug reports that either lack bug descriptions or descriptions where more than half of the lines are logs. We set the threshold at 0.5, guided by extensive data observations revealing that the majority of such bug reports primarily consist of failure logs. Such reports typically lack sufficient UI operations, which are essential for ET. Following this filtering process, 32,772 bug reports are left for the KG construction. Note that 18,829 bug reports of them don't clearly present the preconditions, S2Rs, EBs, and OBs by the criteria set in [31] , making them difficult or even impossible to process using the previous system KG construction method [31].

4.1.2 Baseline and ablations. We compare our approach with SoapOperaTG [31, 32], the state-of-the-art approach for test scenario generation , which has been proven to effectively support testers in finding bugs during exploratory testing. SoapOperaTG utilizes traditional open information extraction methods to construct a system KG of user tasks and system behaviors, derived from clearly expressed S2Rs, EBs, and OBs in bug reports. It generates soap opera test scenarios using rule-based heuristics by combining scenarios of relevant bug reports, using the system KG (see Section 2 for illustrative examples). Additionally, we set up the ablation study to assess the impact of in-context learning (i.e., few-shot vs zero-shot) and the effectiveness of reasoning by comparing the performance with and without chain-of-thoughts (CoTs). Our approach without in-context learning is denoted as *Our app w/o Few*, and one excluding CoTs as *Our app w/o CoTs*.

4.2 Quality of SysKG-UTF (RQ1)

We assess the quality of the KG from three perspectives: section extraction, step splitting, and step clustering, covering the key steps in the KG construction and providing a comprehensive evaluation.

Given the vast number of bug reports for the KG construction, we adopt a statistical sampling method [28]. This method involves sampling the minimal number (MIN) of bug reports in the KG to ensure the estimated accuracy within the samples maintains a 0.05 error margin at a 95% confidence level. Two authors independently evaluate the quality of the KG. They are blinded to the source of the results, meaning they do not know whether the results come from our approach or SoapOperaTG. After completing their evaluation, they discuss any discrepancies in their assessments and finally reach a consensus. For assessing the agreement between raters, we employ Cohen's Kappa coefficient [19]. The computed Cohen's

Table 1: Section Extraction Accuracy

Tool	Prec	S2Rs	EBs	OBs
SoapOperaTG	0.903	0.708	0.705	0.655
Our app w/o Few	0.482	0.824	0.821	0.845
Our approach	0.926	0.976	0.921	0.892

kappa scores range between 86.6% and 96.5% for the various evaluations, indicating a strong level of agreement. Based on these final decisions, we compute the accuracy of section extraction.

The binary metrics like accuracy, classifying outcomes merely as correct or incorrect, are not insufficient to evaluate step splitting and clustering. For example, our approach splits "Turn on interaction recording and logging and restart Firefox" into "Turn on interaction recording and logging" and "Restart Firefox". This step splitting is correct but not thorough. "Turn on interaction recording and logging" can be further splitted into two atomic steps. Therefore, the performance of step splitting and step clustering is assessed by the Likert Scale [21], with 1 representing the worst performance and 5 signifying the best. This metric ensures a comprehensive assessment, adeptly capturing the subtleties in performance that binary metrics overlook. Furthermore, we also utilize the Davis-Bouldin index (DBI), a widely adopted metric for evaluating the quality of clustering algorithms [7], to assess the step clustering.

4.2.1 Section Extraction. We evaluate the quality of section extraction by randomly sampling 380 bug reports. The results are illustrated in Table 1. Our approach achieves accuracy of 0.926, 0.976, 0.921, and 0.892 for extracting preconditions, S2Rs, EBs, and OBs, which notably exceed 0.903, 0.708, 0.705, and 0.655 by SoapOperaTG. The results demonstrate that our section extraction method significantly outperforms the baseline across all sections.

SoapOperaTG uses rule-based heuristics to extract the corresponding sections, thus it fails when bug reports do not explicitly tag the related sections. However, our approach, benefiting from the superior NLU capabilities of LLMs, isn't constrained by the structure of bug reports and can even fill in missing parts based on the context described. For example, for Bug 1806518, SoapOperaTG doesn't extract any sections. In contrast, our approach correctly identifies ["Open https://www...", "Click 'Share Your Screen"', "Select one of the screens ..."] for the S2Rs section, [The inline-end (left) side of the preview video is cut off in the popup notification] for OBs. It also infers [The preview video should be fully visible in the popup notification] for the EBs. Despite the significant improvement in extraction by our approach, there is still room for further enhancement. For example, our approach often incorrectly classifies Firefox's version information as preconditions (e.g., 'Nightly version: 48...') or mistakenly includes the first few steps of the S2Rs into preconditions (e.g., 'Open a tweet'). To resolve this, we can enhance our approach by offering explicit definitions for these sections, thereby allowing LLMs to better understand these sections. We assess the effectiveness of in-context learning by comparing our approach with *Our app w/o Few*. Results in Table 1 demonstrate that in-context learning elevates the accuracy of section extraction from 0.482, 0.824, 0.821, and 0.845 to 0.926, 0.976, 0.921, and 0.892.

4.2.2 Step Splitting. We further evaluate the step splitting performance using the extracted S2Rs by the corresponding approach in Section 4.2.1. The results are presented in Table 2. In the Likert Scale results, the values correspond to the percentage of S2Rs that

Table 2: Step Splitting Likert

Tool	5-point Likert (1 worst, 5 best)				
	1	2	3	4	5
SoapOperaTG	0.120	0.143	0.114	0.137	0.486
Our app w/o Few	0.091	0.064	0.053	0.121	0.672
Our approach	0.027	0.034	0.068	0.061	0.811

Table 3: Step Clustering Result

Tool	5-point Likert (1 worst, 5 best)					DBI
	1	2	3	4	5	
SoapOperaTG	0.013	0.037	0.089	0.202	0.660	0.960
Our approach	0.008	0.026	0.094	0.192	0.680	0.754

received a specific Likert score out of the total number of S2Rs. Our approach significantly outperforms SoapOperaTG in step splitting. We additionally assess the accuracy of our approach in identifying the type of step as operation or non-operation. The accuracy is 0.969. Comparing our approach with *Our app w/o Few* in Table 2, it is evident that in-context learning can substantially enhance the performance of step splitting. Moreover, it leads to an increase in the accuracy of step type identification from 0.926 to 0.969.

Our approach can handle situations not previously encountered during in-context learning. For example, it successfully splits compound steps linked by '->', such as 'Open Devtools -> Performance', into 'Open Devtools' and 'Switch to Performance panel', demonstrating its ability to generalize from in-context learning examples. However, SoapOperaTG fails to break down this step into atomic steps. Additionally, our approach is not simply a mechanical splitter. It decides whether to split by the specific semantics of the step. For instance, the step 'Go to workato.com or any site that uses Intercom...' contains the conjunction 'or', yet our approach does not split it as 'or' suggests a choice between two objects, not a sequence of operations. In contrast, SoapOperaTG incorrectly splits it into 'Go to workato.com' and 'Go to any site...'. While our approach demonstrates generalizability and the capacity to address specific problems based on semantics, its ability to handle compound sentences involving temporal sequences requires improvement. For instance, it incorrectly splits 'Open a tab toolbox from about:debugging' into 'Open a tab toolbox' and 'Navigate to about:debugging', with incorrect temporal order wherein 'Navigate to about:debugging' should precede. We can enhance step splitting by more representative examples involving complex temporal order.

4.2.3 Step Clustering. Identical steps with various expression variations across different bug reports are clustered to construct a more coherent KG. This section is dedicated to evaluating the effectiveness of step clustering. Our approach groups steps into 41,467 clusters, while SoapOperaTG groups steps into 48,143 clusters. By the statistical sampling method [28], we sample 381 and 382 clusters with at least two steps for our approach and SoapOperaTG, respectively. The effectiveness of step clustering is assessed in two ways: subjectively using the Likert Scale [21], where 1 is the worst and 5 is the best, and objectively using the DBI [7]. The results are shown in Table 3. For Likert Scale results, the values represent the proportion of clusters that receive a particular Likert score out of the total number of clusters. Importantly, a lower DBI value (i.e., the better the separation of the clusters and the tightness inside the clusters) indicates better clustering results. From Table 3, our approach performs slightly better than SoapOperaTG.

Table 4: Quality of Test Scenarios

Tool	Feasibility							Coherence
	TP	FP	TN	FN	P	R	F1-score	Accuracy
SoapOperaTG	256	92	0	0	0.736	1.000	0.848	0.727
Our app w/o Few	287	47	14	36	0.859	0.889	0.874	0.819
Our app w/o CoTs	277	38	23	46	0.879	0.858	0.868	0.803
Our approach	301	19	42	22	0.941	0.932	0.936	0.966

The sub-optimal step splitting affect the clustering performance of SoapOperaTG. For example, SoapOperaTG incorrectly groups 'Open DevTools > Performance.' and 'Open Devtools.' into a cluster. For steps with identifiable UI elements and actions, SoapOperaTG performs better than our approach as it clusters these steps based on shared UI elements and actions [31]. It groups steps involving 'Create New Login' button and 'click' action into a cluster. However, our approach incorrectly clusters 'Click on the Create New Login button.' and 'Create a new login' together. A potential improvement for our approach is to identify UI elements and actions during step splitting, and then using these for better clustering.

4.3 Quality of Test Scenarios (RQ2)

For evaluating the quality of test scenarios, we randomly select 8 bugs from the dataset as seed bugs. For each seed bug, we use its top-10 relevant bugs, thus obtaining 80 bug pairs. Note that the top-10 relevant bugs for each seed bug, as identified by our approach, are not identical to those found by SoapOperaTG. This difference can be attributed to the bug knowledge difference in our KG and SoapOperaTG's KG, namely our KG including more bug reports by the superior approach for KG construction. Additionally, the algorithms used to find relevant bugs also differ. Specifically, our approach focuses only on shared steps classified as operations. Owing to this difference in bug pairs, we calculate the metrics for our approach and SoapOperaTG separately, using the test scenarios that each approach independently generates.

Our approach applies the 80 bug pairs to generate 384 potential test scenarios. When forming these scenarios, our approach employs a filtering process that omits test scenarios that replicate the S2Rs of other test scenarios or the bugs within the bug pair. This filtering process, detailed in Section 3.2.2, ensures a low level of redundancy among 384 test scenarios. In contrast, SoapOperaTG generates 438 potential test scenarios based on its bug pairs. SoapOperaTG doesn't integrate a filtering process for redundancy. When we apply our redundancy filtering process to SoapOperaTG's output, the number of potential test scenarios reduces to 348. One author verifies if these 90 test scenarios have identical S2Rs to the remaining 348. The outcome confirms that the excluded test scenarios are indeed duplicates of at least one of the left scenarios. Thus, to avoid repeated evaluations of test scenarios with the same S2Rs, our evaluation in this section uses the remaining 348 test scenarios.

4.3.1 Feasibility of Test Scenarios. We utilize precision, recall, and the F1-score as metrics to gauge the feasibility of test scenarios. Precision, in this context, represents the ratio of truly feasible test scenarios correctly identified by the approach to all the test scenarios predicted as feasible by the approach. Recall signifies the ratio of truly feasible test scenarios correctly identified by the approach to all actually feasible test scenarios in the potential test scenarios. Here, precision acts as an indicator of the accuracy in identifying feasible test scenarios, whereas recall reflects the comprehensiveness in detecting feasible test scenarios. The F1-score is introduced as a means to achieve a balance between precision and recall.

In this section, we utilize 384 potential test scenarios generated by our approach. Our approach identifies 320 as feasible and 64 as infeasible. In contrast, we evaluate 348 potential test scenarios generated by SoapOperaTG. Given that SoapOperaTG does not assess test scenario feasibility, all 348 potential scenarios are deemed feasible. Thus, the true negative (TN) and false negative (FN) values for SoapOperaTG are zero, as shown in Table 4, leading to the recall for SoapOperaTG is 1.000, higher than our approach (0.932). In contrast, the precision for test scenario feasibility is higher for our approach (0.941) compared to SoapOperaTG (0.736), thereby demonstrating the superior performance of our approach in identifying feasible test scenarios. For the F1-score, which is 0.936 and 0.848 for our approach and SoapOperaTG, it becomes apparent that despite our approach overlooking a few truly feasible test scenarios, the overall benefit of discarding infeasible test scenarios outweighs this minor disadvantage. By comparing our approach with *Our app w/o Few* and *Our app w/o CoTs* in Table 4, we clearly observe that both in-context learning and CoT reasoning contribute positively to the feasibility of test scenarios.

The effectiveness of our approach in test scenario feasibility could be further improved by incorporating user interface information into the process. For example, our approach wrongly identifies a test scenario as feasible since its CoTs show that *going to 'about:preferences#privacy' and selecting 'Never remember history' from the 'History' section should not affect observing the highlights on the 'Remove' or 'Edit' buttons. Therefore, this test scenario is feasible.* However, these buttons do not exist on this page, rendering the scenario infeasible. As illustrated by this example, although LLMs inherently hold some knowledge about Firefox, this knowledge is not always effectively utilized. To address this limitation, we propose augmenting LLMs with detailed user interface (UI) information as described in this work [9].

4.3.2 Coherence of Test Scenarios. We further assess the coherence of all feasible test scenarios identified by our approach and SoapOperaTG, which are 320 and 348 test scenarios respectively. Coherence here refers to whether the test scenarios contain any redundant or unnecessary steps. Test scenarios without such steps are deemed coherent. We use accuracy as our metric, which is defined as the ratio of coherent test scenarios to all test scenarios. Our approach demonstrates superior accuracy with a score of 0.966, compared with 0.727 of SoapOperaTG. Our approach effectively identifies and properly removes redundant or unnecessary steps in test scenarios, thereby generating more coherent test scenarios. For instance, our approach eliminates one of the redundant steps in a test scenario, the consecutive and repetitive steps 'Access the "about:preferences#privacy" page.' and 'Go to "about:preferences#privacy".' We compare our approach with *Our app w/o Few* and *Our app w/o CoTs*, which indicates in-context learning and CoTs can significantly improves LLMs' performance from 0.819, 0.803 to 0.966.

4.4 Usefulness for Exploratory Testing (RQ3)

We perform a user study for evaluating the usefulness of test scenarios generated by our approach for supporting exploratory testing.

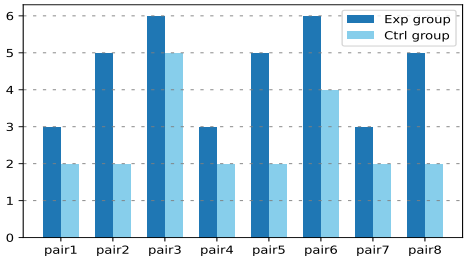

Figure 6: Bug Distribution in User Study

4.4.1 Pre-survey. We recruit 16 participants, comprising 6 Ph.D., 1 MS, 3 Undergraduates and 6 professionals with expertise in testing. Prior to the user study, we conduct a survey to gain insights into the participants' programming experience, familiarity with Firefox Desktop browser, testing experience, and history of reporting bugs to open-source communities. We then pair two participants with similar backgrounds, one in the experimental group and the other in the control group. The pairing is guided by a clear set of priorities. The foremost consideration is participants' testing experience and history of reporting bugs, aligning with the testing-centric nature of user study. Furthermore, as our user study involves user operations, familiarity with the software being tested is our second priority, considering its direct impact on the testing process. Finally, as ET is a manual testing simulating user operations to find bugs and does not involve code-level activities, programming experience and educational background are considered as tertiary factors.

4.4.2 Tasks. Participants are tasked with doing ET on Firefox Desktop browser, with the objective of finding as many new bugs as possible within 2 hours. The participants in experimental group are assisted by test scenarios generated by our approach, the control group using ones from SoapOperaTG. The 8 seed bugs that we utilize for this user study are the ones that are sampled in Section 4.3. Each participant is assigned two of these seed bugs, with participants in a pair receiving the same seed bugs. Each seed bug is used by two participants in a group. Furthermore, participants utilize the top-10 relevant bugs associated with each seed bug during ET.

4.4.3 Results. As shown in Figure 6, participants in the experimental group find 36 new bugs, with 19 confirmed, 1 assigned an assignee, 3 assigned priority and severity, 7 as duplicate, 2 as invalid, 3 as wontfix and 1 as worksforme. Participants in the control group find 21 bugs, with 1 fixed, 8 confirmed, 1 assigned an assignee, 7 as duplicate, 3 as invalid and 1 as wontfix. The user study results demonstrate that the test scenarios generated by our approach can more effectively support participants in conducting ET and finding new bugs. To evaluate the statistical significance of the differences between experimental and control groups, we conduct Welch's t−test [35], using a significance threshold of $p < 0.05$. The obtained p−value is 0.010, which is well below the 0.05 threshold, indicating that the differences are statistically significant.

The user study indicates that certain bugs are difficult to detect by automated execution of test scenarios. For instance, the test scenario generated by our approach involves dragging 'Zoom Controls' and 'Synced tabs' icons to the toolbar by linking steps from Bug 1317686 and Bug 1245844. During the manual ET, an unexpected Bug 1863719 is discovered and has been set priority and severity: *'Zoom Controls' on Toolbar shows unequal spacing between adjacent items, wider on the right*, a discrepancy that is challenging to identify by automated execution of this test scenario.

A confirmed Bug 1844778 is found using the test scenario from our approach by linking steps from Bug 1764787 and Bug 1678986. It involves selecting a saved login on the 'about:logins' page, clicking the 'Remove' button without confirming the removal, and then opening the 'about:logins' page in a new tab and removing all logins. Execution of this test scenario finds Bug 1844778: *'Remove this login?' pop-up remains on the previous 'about:logins' page with clickable buttons even after all logins have been removed.* The discovery of this bug inspires us to vary the test scenario. The test scenario includes two 'removal' operations: remove one login and remove all logins. New test scenarios can be generated by interchanging these two operations or swapping their execution order. For example, one test scenario variation is to execute the 'remove one login' operation on both pages. This test scenario variant leads to uncover a new Bug 1844898: *unintentional removal of the next login occurs after removing the selected login in another "about:logins" page.* Ultimately, we derive 3 new test scenario variants, uncovering Bug 1844898, Bug 1844899, and Bug 1844903.

The underlying principle of these bug discoveries lies in conducting correlated 'removal' operations across two pages. An interesting thought arises: How about replacing these 'removal' with 'edit' operations? Consequently, this modification results in four new test scenario variants. These variants uncover Bug 1844932, Bug 1844940, Bug 1844969, and Bug 1844971. All these 7 bugs found based on the test scenario variants have been assigned priority and severity by the Mozilla developers. Taking inspiration from our positive experiences with test scenario variation, we aim to enrich our approach by incorporating test scenario variation into the process of generating test scenarios instead of merely generating new test scenarios based on step recombination.

5 DISCUSSION

Bugs found inspire us that test scenario variant generation could further enhance our approach's ability by reducing the demand for creative thinking from testers, details as follows:

Specific scenario generalization involves transforming a particular test scenario into a generalized one, which can then be used to test other similar functionalities within the same or even different software. For instance, Bug 1578873 in Figure 2 details a scenario involving tabbing through the 'about:logins' page. The expected behavior is that tabbing should proceed smoothly through all page elements. The generalized scenario for this bug can be generated by replacing the specific UI element 'about:login' with its category, resulting in *'start tabbing through a page, then tabbing should proceed smoothly through all page elements*'. This generalized scenario enables testing of tab navigation on any page in Firefox or even in other desktop applications. Applying this scenario on Firefox's 'about:policies' and 'about:policies#documentation' pages, we discover two new and confirmed tab navigation-related bugs: Bug 1846182 and Bug 1846184. Similarly, we identify a tab navigation issue on Google Chrome's homepage, specifically 'The Gmail button sometimes cannot be focused using the Tab key', reported to Google via 'Send Feedback to Google'. Another tab navigation issue is found on Safari's main page: 'After focusing on the "Show

tab overview" button, pressing the Tab key fails to focus on the next element.', reported to Apple through their Feedback Assistant.

Boundary value generation means creating test scenario variants by negating the preconditions of the original test scenario to test a system's robustness and error-handling capabilities. For instance, we consider the scenario presented by Bug 1571444, performing a search that yields no results on 'about:logins' page. The precondition for this scenario is 'Have a Firefox profile with multiple saved logins.'. By negating this precondition (i.e., 'Have a Firefox profile with no saved logins.') and then executing the same operations, we find a new Bug 1844997: the content 'Looking...' disappears after performing a search on 'about:logins' with no saved logins.

Equivalence class alternation is to generate a test scenario variant by replacing the original operations with equivalent ones to expand the scope of testing. For example, we can adapt the test scenario by searching on the 'about:logins' page via different steps, namely by navigating to 'about:logins?filter=[searchContent]' page. By searching on 'about:logins' in this alternative way, we discover that the same issue still persists in this mode of search, finding a new confirmed Bug 1845136 (the content 'Looking...' disappears after navigating to 'about:logins' page with a prepopulated filter).

6 RELATED WORK

Test cases have been generated based on a variety of data sources and methods, including program differences [2, 6], specifications [1, 27, 34, 38], GUI runtime models [8, 30, 36], and fuzzing technique [10, 22]. Li et al. [20, 33] indicate that bug reports can effectively guide testers to find bugs in similar applications and develop Bugine [20] to recommend bug reports from similar applications to aid in bug finding. Rather than merely recommending bug reports from similar applications to the one being tested, Su et al. [31, 32] construct a KG from bug reports and generate test scenarios based on the KG to support ET. SoapOperaTG [32] is developed to optimize the usage of test scenarios. Despite SoapOperaTG demonstrating effectiveness in bug finding, difficulty in processing bug reports with inconsistent quality and expression and infeasibility in test scenarios impede its practical application and widespread adoption.

Oscar et al. [5] detect the absence or presence of EBs and S2Rs in bug descriptions by using three methods based on the summarized discourse patterns. The ML-based approach DeMIBuD-ML has the best performance, but needs to be trained using a labeled set of bug reports explicitly reporting the ones containing or not EB and S2R. Yang et al. [29] introduce BEE, which identifies the structure of bug descriptions by automatically labeling sentences that correspond to OBs, EBs or S2Rs. Rather than relying on one multi-class SVM for classifying the sentences, BEE implements three binary SVMs, for OBs, EBs, S2Rs. BEE's application is limited as it requires an amount of data to individually train three SVM models. Furthermore, its ability to process bug descriptions is restricted by merely classifying existing sentences into corresponding sections, lacking the capacity to infer sections that aren't explicitly stated in bugs.

Given the remarkable performance of LLMs in NLP, researchers have begun to explore their applicability or impact on software engineering, such as automated code completion [16, 23], software development [12, 26], and security vulnerabilities [11, 24]. Kang et al. [18] propose an approach for bug reproduction by engaging LLMs in a report-to-test generation task. They prompt LLMs to generate executable test scripts from bug reports to reproduce the described issues. Our approach utilizes the LLMs to identify and extract specific sections, namely S2Rs, EBs and OBs from bug reports to construct a SysKG-UTF for doing ET. AdpGPT [9] aims for bug reproduction by extracting the S2R entities from bug reports and reproducing each step based on the current GUI state to trigger the bugs. AdbGPT utilizes LLMs to extract the S2R entities defining each step to reproduce the bug report, including action types, target components, input values, or scroll directions. However, our approach focuses on splitting S2Rs into atomic steps and do step clustering for a more coherent SysKG-UTF construction.

7 THREATS TO VALIDITY

To mitigate potential human bias during the KG evaluation, two authors perform independent assessments, remaining blind to the source of results (whether from our approach or the baseline). In the user study, to control the variation in participants' backgrounds during the user study, we conduct a pre-survey to understand participants' background. Participants with similar backgrounds are paired and assigned to either the experimental or control group, thus further reducing potential bias. A significant external validity threat is its evaluation conducted solely on Firefox. However, Firefox, with more than 20 years of history as a large open-source system, presents a wide array of bug reports of varying quality and presentation styles, thereby ensuring the diversity of our test dataset. The diversity of Firefox's functionalities help mitigate the single-system bias. Given our approach's lightweight nature, which utilizes prompt engineering with few representative examples, it is easily transferred to other systems. We plan to deploy our approach on a wider range of software systems to further verify its effectiveness. The KG construction relies on the numerous bug reports. For the new or unpopular software lacking such data, we can utilize specific scenario generalization method, outlined in Section 5. It generalizes test scenarios from analogous software and then applies the scenarios to the target software, mitigating cold-start.

8 CONCLUSIONS AND FUTURE WORK

We propose a lightweight approach that leverages robust NLU capabilities of LLMs to construct a SysKG-UTF, effectively processing bug reports with inconsistent quality and diverse expression styles. Then, utilizing system and bug knowledge encapsulated in the KG, along with logistical reasoning capabilities of LLMs, we generate test scenarios with high feasibility and coherence. Our evaluation demonstrates the high performance of KG construction and test scenario generation. The user study shows the effectiveness of test scenarios in supporting ET. Furthermore, our experience with 88 bugs found during the development and user study prompts us to consider potential benefits of test scenario variant generation, which could enhance our approach's ability to support ET by alleviating the demand for creative thinking from testers. In the future, we will integrate test scenario variant generation to our approach.

9 ACKNOWLEDGEMENTS

This work is supported by the National Nature Science Foundation of China under Grant (Nos. 62262031, 62367003).

REFERENCES

[1] Yusuke Aoyama, Takeru Kuroiwa, and Noriyuki Kushiro. 2020. Test case generation algorithms and tools for specifications in natural language. In *2020 IEEE International Conference on Consumer Electronics (ICCE)*. IEEE, 1–6.

[2] Marcel Böhme, Bruno C d S Oliveira, and Abhik Roychoudhury. 2013. Regression tests to expose change interaction errors. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 334–344.

[3] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[4] JD Cem Kaner and James Bach. 2006. The nature of exploratory testing. (2006).

[5] Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. 2017. Detecting missing information in bug descriptions. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 396–407.

[6] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Lu Zhang. 2019. History-guided configuration diversification for compiler test-program generation. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 305–316.

[7] David L Davies and Donald W Bouldin. 1979. A cluster separation measure. *IEEE transactions on pattern analysis and machine intelligence* 2 (1979), 224–227.

[8] Christian Degott, Nataniel P Borges Jr, and Andreas Zeller. 2019. Learning user interface element interactions. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 296–306.

[9] Sidong Feng and Chunyang Chen. 2023. Prompting Is All Your Need: Automated Android Bug Replay with Large Language Models. *arXiv preprint arXiv:2306.01987* (2023).

[10] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. 2008. Automated whitebox fuzz testing.. In *NDSS*, Vol. 8. 151–166.

[11] Anastasiia Grishina. 2022. Enabling automatic repair of source code vulnerabilities using data-driven methods. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 275–277.

[12] Saki Imai. 2022. Is github copilot a substitute for human pair-programming? an empirical study. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*. 319–321.

[13] Juha Itkonen and Mika V Mäntylä. 2014. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering* 19 (2014), 303–342.

[14] Juha Itkonen, Mika V Mäntylä, and Casper Lassenius. 2012. The role of the tester's knowledge in exploratory software testing. *IEEE Transactions on Software Engineering* 39, 5 (2012), 707–724.

[15] Juha Itkonen and Kristian Rautiainen. 2005. Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering, 2005.* IEEE, 10–pp.

[16] Naman Jain, Skanda Vaidyanath, Arun Iyer, Nagarajan Natarajan, Suresh Parthasarathy, Sriram Rajamani, and Rahul Sharma. 2022. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*. 1219–1231.

[17] Cem Kaner. 2008. A tutorial in exploratory testing. *QUEST* (2008), 6.

[18] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[19] J Richard Landis and Gary G Koch. 1977. An application of hierarchical kappa-type statistics in the assessment of majority agreement among multiple observers. *Biometrics* (1977), 363–374.

[20] Ziqiang Li and Shin Hwei Tan. 2020. Bugine: a bug report recommendation system for Android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Companion Proceedings*. 278–279.

[21] Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of psychology* (1932).

[22] David Molnar, Xue Cong Li, and David A Wagner. 2009. Dynamic Test Generation to Find Integer Bugs in x86 Binary Linux Programs.. In *USENIX Security Symposium*, Vol. 9. 67–82.

[23] Noor Nashid, Mifta Sintaha, and Ali Mesbah. 2023. Retrieval-based prompt selection for code-related few-shot learning. In *Proceedings of the 45th International Conference on Software Engineering (ICSE'23)*.

[24] Hammond Pearce, Benjamin Tan, Baleegh Ahmad, Ramesh Karri, and Brendan Dolan-Gavitt. 2023. Examining zero-shot vulnerability repair with large language models. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2339–2356.

[25] Dietmar Pfahl, Huishi Yin, Mika V Mäntylä, and Jürgen Münch. 2014. How is exploratory testing used? a state-of-the-practice survey. In *Proceedings of the 8th ACM/IEEE international symposium on empirical software engineering and measurement*. 1–10.

[26] Advait Sarkar, Andrew D Gordon, Carina Negreanu, Christian Poelitz, Sruti Srinivasa Ragavan, and Ben Zorn. 2022. What is it like to program with artificial intelligence? *arXiv preprint arXiv:2208.06213* (2022).

[27] Yuji Sato. 2020. Specification-based test case generation with constrained genetic programming. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 98–103.

[28] Ravindra Singh and Naurang Singh Mangat. 2013. *Elements of survey sampling*. Vol. 15. Springer Science & Business Media.

[29] Yang Song and Oscar Chaparro. 2020. Bee: A tool for structuring and analyzing bug reports. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1551–1555.

[30] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[31] Yanqi Su, Zheming Han, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2022. Constructing a System Knowledge Graph of User Tasks and Failures from Bug Reports to Support Soap Opera Testing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–13.

[32] Yanqi Su, Zheming Han, Zhenchang Xing, Xiwei Xu, Liming Zhu, and Qinghua Lu. 2023. SoapOperaTG: A Tool for System Knowledge Graph Based Soap Opera Test Generation. In *2023 IEEE/ACM 45th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 51–54.

[33] Shin Hwei Tan and Ziqiang Li. 2020. Collaborative bug finding for android apps. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 1335–1347.

[34] Rong Wang, Yuji Sato, and Shaoying Liu. 2019. Specification-based Test Case Generation with Genetic Algorithm. In *2019 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 1382–1389.

[35] Bernard L Welch. 1947. The generalization of 'STUDENT'S'problem when several different population varlances are involved. *Biometrika* 34, 1-2 (1947), 28–35.

[36] Thomas D White, Gordon Fraser, and Guy J Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 307–317.

[37] James A Whittaker. 2009. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education.

[38] Arvin Zakeriyan, Ramtin Khosravi, Hadi Safari, and Ehsan Khamespanah. 2021. Towards automatic test case generation for industrial software systems based on functional specifications. In *Fundamentals of Software Engineering: 9th International Conference, FSEN 2021, Virtual Event, May 19–21, 2021, Revised Selected Papers 9*. Springer, 199–214.