Constructing a System Knowledge Graph of User Tasks and Failures from Bug Reports to Support Soap Opera Testing

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

Zheming Han
Australian National University
Australia
u6865707@anu.edu.au

Zhenchang Xing†
Data61, CSIRO
Australia
Zhenchang.Xing@anu.edu.au

Xin Xia
Huawei
China
Xin.Xia@acm.org

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

Liming Zhu‡
Data61, CSIRO
Australia
liming.zhu@data61.csiro.au

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

ABSTRACT

Exploratory testing is an effective testing approach which leverages the tester's knowledge and creativity to design test cases to provoke and recognize failures at the system level from the end user's perspective. Although some principles and guidelines have been proposed to guide exploratory testing, there are no effective tools for automatic generation of exploratory test scenarios (a.k.a soap opera tests). Existing test generation techniques rely on specifications, program differences and fuzzing, which are not suitable for exploratory test generation. In this paper, we propose to leverage the scenario and oracle knowledge in bug reports to generate soap opera test scenarios. We develop open information extraction methods to construct a system knowledge graph (KG) of user tasks and failures from the steps to reproduce, expected results and observed results in bug reports. We construct a proof-of-concept KG from 25,939 bugs of the Firefox browser. Our evaluation shows the constructed KG is of high quality. Based on the KG, we create soap opera test scenarios by combining the scenarios of relevant bugs, and develop a web tool to present the created test scenarios and support exploratory testing. In our user study, 5 users find 18 bugs from 5 seed bugs in 2 hours using our tool, while the control group finds only 5 bugs based on the recommended similar bugs.

*Corresponding author.
†Also with Australian National University.
‡Also with University of New South Wales.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE '22, October 10–14, 2022, Rochester, MI, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3556967

CCS CONCEPTS

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

KEYWORDS

Knowledge Graph, Exploratory Testing, User Tasks and Failures

ACM Reference Format:
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 *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3551349.3556967

1 INTRODUCTION

Exploratory testing [12, 23, 56] is a complementary style of testing to scripted testing [19–21, 42]. In scripted testing, test cases are pre-designed and are then executed in a repetitive way in mostly a mechanical manner. In contrast, exploratory testing leverages the knowledge, creativity and experience of testers to create varying and diverse tests on the fly to provoke unexpected bugs and recognize them. Although scripted testing and test automation have received much attention in the research community, exploratory testing is widely practiced and appreciated in the software industry [19–21, 42], and some scientific studies [19–21] show that exploratory testing is an effective and efficient testing approach for system-level testing of interactive systems through the GUI and from the end user's viewpoint, and can find functional, usability, performance and security bugs [42].

Exploratory testing treats test-related learning, test design, test execution, and test result interpretation as mutually supportive activities that run in parallel [23]. Cem Kaner, James Bach and others [12, 23, 56] have developed concrete principles and guidelines for performing exploratory testing. The foremost important message is that *not scripting does not mean not preparing* [23]. In fact, exploratory testers need to collect, learn and analyze a wide range of information about application domain, software products,

and users. However, there are no tools to support this information analysis process and to inspire exploratory testing.

In this work, we focus particularly on *user tasks and failure history*. We would like to model the user interface of the product, the transactions that people try to complete (what are the steps, data items, outputs, displays, etc.), and the expected and actual outcomes. Based on this model of user tasks and failures, we would like to support scenario-based exploratory testing (also known as soap opera testing [10]) where the testers can design complex test scenarios to provoke failures and recognize them. Among various information sources of user tasks, we focus on bug reports as the primary source, because bug reports are more readily available than other sources like requirements. Furthermore, bug reports contain failure history that is generally absent in other sources. High-quality bug reports generally describe the steps to reproduce the bug (S2Rs), observed (incorrect) behavior (OB), and the expected (correct) behavior (EB) [8], from which we can derive a model of user tasks and failure history (see the examples in Figure 1).

Much work has been done to automatically reproduce the bugs from the S2Rs descriptions [15, 61]. Some research aims to detect duplicate or related bugs to facilitate bug triaging and bug localization [1, 58, 62]. A recent work [50] builds a bug-component triaging graph to boost the bug triaging performance. Tan and Li propose collaborative bug finding which uses a bug in one app to inspire the test creation in the other similar app [52]. None of existing work performs fine-grained analysis of the S2Rs, OB and EB content in the bug reports to create a model of user tasks and failures, and apply the model to support exploratory testing.

In this work, we represent the model of user tasks and failures in a system knowledge graph (KG), which consists of a static part and a dynamic part. The static part of the KG contains the knowledge of application concepts and individual system features and operations, while the dynamic part contains the knowledge of task workflows from the user's perspective, observed and expected behaviors. As bug reports are semi-structured text, we develop domain-specific open information extraction methods to extract application concepts, task workflows and oracles from the bug report contents to build the knowledge graph. Our knowledge graph transforms textual information about user tasks and failures in bug reports into structured knowledge, and this essentially crowd-contributed knowledge would go beyond the knowledge of individuals. The structured knowledge is more convenient to work with, and more likely to lead to new insights than working with bug reports.

Our next challenge is to map the knowledge in the KG to test ideas and test scenarios. In this work, we assume a tester obtains a bug report by some means (could be scripted testing or exploratory testing), and she would like to start the exploratory testing from this seed bug (see an exploratory testing example by the Mozilla tester inspired by a bug reported by us). First, we find past relevant bugs that share some common user task steps with the seed bug. Different from duplicate or similar bug detection [22, 41], our relevant bugs may not be overall similar to the seed bug, or may not even be about the same system features or components. This creates the opportunities to compose soap opera tests for simulating expert use of the system and revealing windfall failures [20]. In this work, we create soap opera tests by concatenating the user task workflows of the seed bug and a relevant bug at both scenario and step level. The created soap opera tests reuse past failures (EBs and OBs) at both scenario and step level. The steps' EBs and OBs help the tester attend to additional potential failures on the way of performing this soap opera test. This is similar to exploring side quests in a role-playing game while the player is completing the main mission.

We developed a proof-of-concept system knowledge graph from the Mozilla Firefox bugs. We choose Firefox because it is a complex web browser that supports rich user features and is stable, which makes it an ideal test bed for exploratory testing. By a statistical sampling method [47], we confirm the high accuracy of our information extraction methods. We developed a web interface (see Figure 4) to present the composed soap opera test scenarios. On this web interface, the tester can freely explore the seed bug, the relevant bugs and the created soap opera tests. Relying on the KG, the important application concepts are highlighted as information cues for test learning and result interpretation. Meanwhile, the S2Rs descriptions are chunked into atomic steps when constructing the KG to ease the workflow understanding and execution. We conduct a user study involving 10 users of 5 seed bugs. 5 users using our tool find 18 bugs in 2 hours, while 5 users relying on only recommended similar bugs find only 5 bugs. The study shows our exploratory testing tool supports users to learn the product, simulate expert users of the system, and expose hidden failures in a complex way.

In this paper, we make the following contributions:

- We propose a knowledge graph-based method for automatic generation of soap opera test scenarios.
- We design and construct a novel system knowledge graph to represent user tasks and failures in bug reports.
- We design a method to compose soap opera test scenarios from relevant bug scenarios in the knowledge graph.
- We develop a proof-of-concept tool of our approach and confirm the quality of the constructed knowledge graph and the usefulness of our exploratory testing tool. Our replication package can be found here [1].

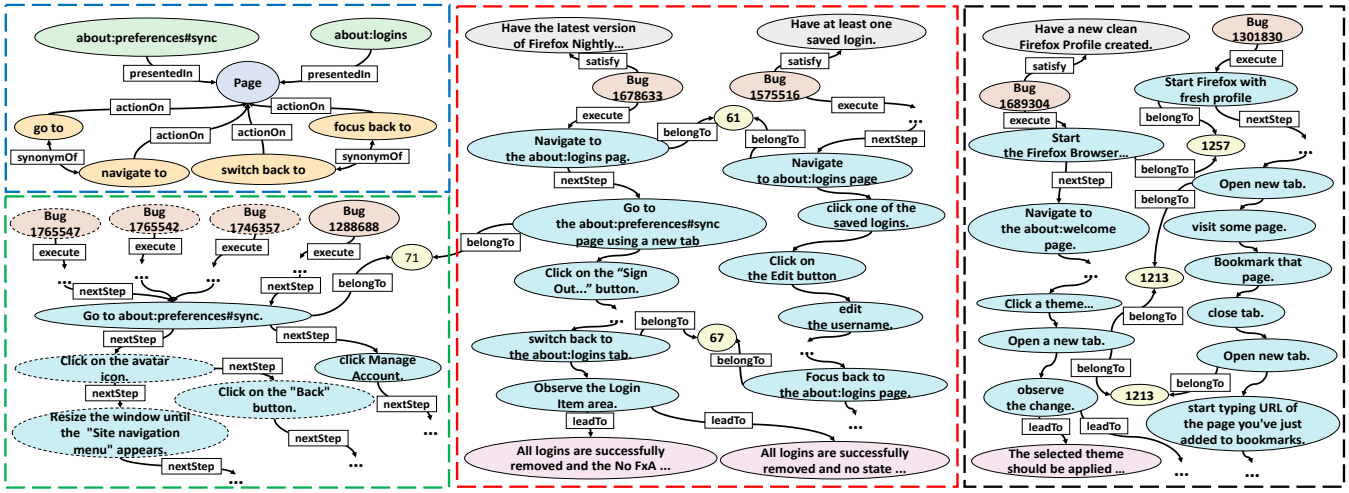
2 BUG REPORTS AS A KNOWLEDGE SOURCE OF USER TASKS AND FAILURES

Like a soap opera on television, soap opera tests should reflect motivating and credible system usage scenarios. Meanwhile, soap opera tests are condensed and exaggerated to depict dramatic system use in order to reveal complex and hidden problems. Soap opera tests should be easy to evaluate with hints to interpret the results and recognize failures. Bug reports are created by people with diverse roles and involvement in the system. Their S2Rs reflect real-life system use, and their OBs and EBs summarize what to attend to for result interpretation and past failures. Furthermore, system use from multiple bug reports can be combined into a hypothetical "drama" to test the system in a complex way.

2.1 Scenarios

When submitting a bug report, the reporter describes the scenario leading to the bug as steps to reproduce (S2Rs). The scenarios in bug reports are not only useful for reproducing and fixing bugs, but also provide abundant and variant real-life scenarios for software testing.

[1] https://github.com/SuYanqi/SYS-KG



Please refer to Figure 3 for entity types following the same color coding

Figure 1: KG Fragments for the Examples in Section 2

For example, when a Firefox browser user attempts to set the master password and right-clicks inside the password field, the context menu does not offer the "Fill password" option. Then, the user reports a bug Bug 1572445. The S2Rs in this bug report can be used for testing the master password setting further. One author followed this scenario and found another bug during exploratory testing. When right-clicking inside the password field, the context menu displays some blank options. This new functional Bug 1764683 was quickly confirmed and fixed by the Mozilla developer.

2.2 Oracles

In the above example, although the original bug "no fill-password option" is different from the new bug "display blank options", the OB and EB of the original bug provide the hints for what to attend to for recognizing errors. This is important for exploratory testing as humans are fallible, i.e., testers do not always recognize a failure even when a test reveals it [5]. This hinting power is also a source of creativity in exploratory testing. For example, the EB and OB of the bug report Bug 1570945 is about whether the avatar on "about:logins" page is displayed correctly. Inspired by this hint, when observing the avatar, the author found that there is an impurity on the avatar, and reported a new aesthetic Bug 1744772, which was confirmed and fixed by the Mozilla developers.

2.3 Hypothetical Dramas

After researching and analyzing a number of bug reports, we realize that steps in the S2Rs among many bug reports have the intersection, even though the bugs are not similar or not about the same features or system components. This allows the testers to compose the scenarios and oracles from different bug reports into hypothetical scenarios that inspire expert uses of the system and complex feature interactions, in addition to using the scenarios individually. Hypothetical dramas can be created at both scenario and step level.

2.3.1 Scenario Level Dramas. For example, as shown in the red box in Figure 1, both Bug 1678633 and Bug 1575516 have the steps ("Navigate to the about:logins page" and "Switch back to the about:logins tab"). Although they are both about "about:logins" page, except for the two common steps, the rest of operations are completely different. Bug 1678633 is about wrong display on "about:logins" page when signing out on "about:preferences#sync" page. Bug 1575516 is about editing a saved login without saving the changes.

By appending the scenario from Bug 1678633 to Bug 1575516, a soap opera is created, namely "Assume that a user using her classmate's computer to change the saved login in her Firefox account. She just changes the username and password without saving the changes. Her classmate rushes to use the computer. Then the user hurries to sign out on 'about:preferences#sync' page using a new tab and check the option to delete data from this device. But when she goes back to 'about:logins' page, the Edit mode is still open and can still reveal the password.". From this hypothetical soap opera generated from the two bugs, the author found a new security bug Bug 1764756 (i.e., Edit mode is still open after "Remove All Logins" on another tab), which involves two features (edit and sign out) on two different pages ("about:logins" and "about:preferences#sync"). The Mozilla developer commented on this bug "thanks for reporting this bug and attaching video, good catch!". As of the paper writing, the bug patch was committed and waiting for the further testing. We can imagine it could be hard for testers to think out this scenario from scratch due to the complex operations and dramatic interactions involving two features and two different pages.

2.3.2 Step Level Dramas. Bug 1689304 and Bug 1301830 are from different Product::Components, i.e., Firefox::Messaging System and Firefox::Address Bar respectively. These two components have no relevance. However, as seen in the black box in Figure 1, these two bugs have two common steps "Start the Firefox Browser with the profile from prerequisites." and "Open a new tab.". By combining the steps before the second common step from Bug 1689304, the second common step and the steps after the second common step from Bug 1301830, we obtain a new test scenario: a) select a theme on "about:welcome" page; b) open a new tab; c) visit some page, bookmark that page, close the tab and visit the page again in a new tab. In Step c, we visit "https://www.google.com/", then we find that the first time we visit the website, the theme is still the previous one

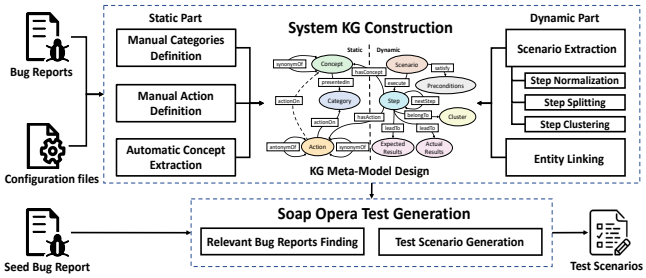


Figure 2: Approach Overview

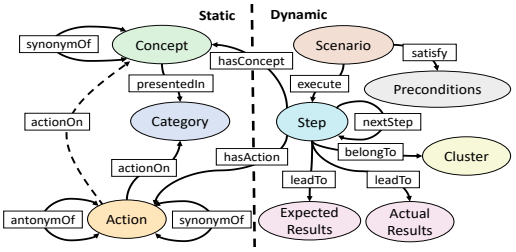


Figure 3: Meta-model of Our System Knowledge Graph

on this website. The second time opening the website, the theme becomes the one selected in Step a. Then, after simplifying the steps, we reported this usability bug in Bug 1766384, which has been already confirmed.

2.3.3 Drama after Drama. Exploratory testing is not a one-off activity. Instead, experiences and bugs from exploratory testing may lead to more exploratory testing which reveals more bugs. For example, the S2Rs in Bug 1288688 says "Go to about:preferences#sync, and click "Manage Account"" in Step 2. From previous exploratory testing (Section 2.2), the author learns that the account and the avatar on "about:logins" page navigate to the same page. She assumes the "Manage Account" button and the avatar on "about:preferences#sync" page would have the same effect. Thus, she clicks the avatar icon instead, not "Manage Account" mentioned in Step 2 of Bug 1288688. Then, she found a new bug, reported in Bug 1746357, namely a blank page shown after clicking the avatar icon on "about:preferences#sync" page. Inspired by this Bug 1746357, the Mozilla developer found a new bug in Bug 1762634 (i.e., Avatar change URL is out of date). After our reported Bug 1746357 is fixed, the author checked how it is fixed. Now, clicking the avatar icon navigates to the "Change profile picture" page. Then, she wanted to go back by using the "back" button, but no response. Another new bug was found and reported in Bug 1765542. Since the "back" button has no response, she tried to return to the previous page by using the "Site navigation menu", which does not work either. This new bug is reported in Bug 1765547. Then, the Mozilla tester did exploratory testing based on Bug 1765542 and Bug 1765547 we reported, which leads to important notes on the bugs and possible resolutions, details in Issue 12695. Furthermore, these new bugs provide additional rich test scenarios and can be merged and reused for testing, as shown in the green box in Figure 1.

3 KG-EMPOWERED SOAP OPERA TESTING

Inspired by our positive experience in exploratory testing based on the scenarios and oracles in bug reports, we construct a novel system knowledge graph (KG) of user tasks and failures from bug reports, and use this knowledge graph to support system-level soap opera testing of interactive systems through the GUI. The approach overview is shown in Figure 2.

3.1 System KG Construction

Although bug reports contain rich knowledge of system's logic, features, interactions and failures, this knowledge is implicitly embedded in text as the S2Rs, OB and EB descriptions. To utilize the scenario and oracle knowledge in bug reports in a systematic and effective way for exploratory testing, we construct a system knowledge graph by mining application concepts, system features, task workflows and failures included in bug reports.

3.1.1 KG Meta-Model. Figure 3 shows the meta-model of our system knowledge graph. We follow the separation of concern principle to model static and dynamic knowledge of user tasks in two interconnected parts, namely static part and dynamic part.

The **static part** captures application concepts and how these concepts are represented and operated in user-system interaction. It contains three kinds of entities (*Concept*, *Category* and *Action*) and four kinds of relations (*presentedIn*, *actionOn*, *synonymOf* and *antonymOf*). A *concept* is an application concept involved in a user task, which may come from application domain and system features. A concept can be *presented in* some GUI elements through which the user interacts with certain concept. We abstract GUI elements into *categories* to simplify the modeling of the relationships between actions and concepts. An *action* is a type of user operation applicable to certain GUI element category. The <*action*, *actionOn*, *concept*> relationship can be derived from the <*action*, *actionOn*, *category*> and <*concept*, *presentedIn*, *category*> relationship. In bug reports, the same application concept or user operation may be described differently, for example, "Browsing & download history" and "Browsing & Download History", "check" and "tick". The *synonymOf* relations model the different ways of expressing the same concept or action. Actions also have *antonymOf* relations which represent user operations with opposite effects (e.g., check and uncheck, enable and disable). Note that action *synonymOf* and *antonymOf* depends on an action-category pair. For instance, if "check" acts on "checkbox", it is the synonym of "tick". But if "check" acts on "text", it is the synonym of "observe". The construction of the static part involves two steps: the manual definition of commonly used GUI element categories and user actions (Section 3.1.2), and the automatic extraction of application concepts from software resource files and bug reports (Section 3.1.3).

The **dynamic part** captures user task workflows and outcomes. A user task is represented in a *scenario*. A scenario may optionally have some *preconditions* that the execution of the scenario must satisfy. A scenario contains a sequence of *steps*, chained by the *nextStep* relations. Any step (not just the ending step of the scenario) may lead to certain expected or observed outcomes, which are represented by the <*step*, *leadTo*, *expected results*> and <*step*, *leadTo*, *actual results*> relationships. A step may involve certain concept and action which are represented by the <*step*, *hasConcept*, *concept*> and <*step*, *hasAction*, *action*> relationships. These two relationships connect the static and dynamic part of the KG. For example, the step "Navigate to the about:logins page." has "navigate to" action and

"about:logins" concept. The construction of dynamic part includes two steps: scenario extraction from bug reports (Section 3.1.4), and entity linking between steps and concepts/actions (Section 3.1.5).

3.1.2 Constructing Static Part - Manual Definition of GUI Element Categories and User Actions. Although application concepts are open-ended, the GUI elements used to present them for user interaction are limited. In this work, we include eight categories of commonly used GUI elements: *radio button* and *drop down menu* for multiple choice, *check box* for option, *button* for taking action, *text field* for text input, *text* for displaying information, *panel* for organizing information, and *key* for keyboard operation. As our proof-of-concept KG is for a web browser, we include three categories of web-browser specific GUI elements: *hyperlink* reference, web *page*, and *app* for the browser itself. We include an *others* category for the concepts not included in the 11 main categories.

In GUI applications, standard user operations are limited, including click, switch, drag, hold, release, etc. However, these user operations may be mentioned in different terms in bug reports. Based on the frequent action verbs from the step extraction (see Section 3.1.4), we summarize 29 types of common user actions and their expression variants, e.g., click (click on), go to (navigate to, access, visit), open (start, launch), type (enter, fill). When linking actions to categories, we determine *synonymOf* and *antonymOf* actions which have equivalent or opposite effects when applied to certain category of GUI elements. For example, the action *check*, *tick* and *enable* are equivalent for enabling an option in a *check box*, while *uncheck*, *untick*, *disable* have the opposite effect.

3.1.3 Constructing Static Part - Automatic Concept Extraction. We extract *concepts* and determine their *categories* from two sources: GUI configuration files and bug reports.

GUI applications generally use GUI configuration files to store the information displayed on the GUI in key-value pairs. The key usually describes the GUI element category, and the value generally includes relevant application concept. For example, the Mozilla Firefox's GUI configuration file has a key-value pair: "performance-use-recommended-settings-checkbox" and "Use recommended performance settings". From this key-value pair, we can extract a *check box* for the concept *use recommended performance settings*. We use the extracted value string as-is as the concept label. We split the key string into tokens by white space, hyphen, underscore, camel case, etc. We summarize a dictionary that maps the tokens generally used to describe certain categories to the standard category labels. We use the mapped standard category label as the concept category.

GUI configuration files provide standard forms of application concepts. Next, we extract variant forms of application concepts from bug reports. For example, the concept "Saved Logins…" are often written as "Saved logins" in bug reports. By convention, people often use quotes (e.g., "Saved Logins..."), brackets (e.g., [Saved Logins...]), or capitalized words (e.g., "TAB") to indicate some words with special meanings. We define regular expressions to extract application concepts in the descriptions of preconditions, S2Rs, OBs and EBs of a bug report. Then, we merge application concepts extracted from bug reports with those from GUI configuration files. In particular, we use Sentence-BERT (SBERT) [44] to embed the concepts from the two sources and compute the cosine similarity between the two concepts. If the cosine similarity between the concept from bug reports and the top-1 most similar concept from GUI configuration file is ≥ 0.85, the concept from the bug report is considered as a synonym of the concept from GUI configuration file. Through experiments, we found that when the threshold is ≥ 0.85, the two texts can guarantee a strong semantic correlation. If the top-1 similarity is below 0.85, we treat the concept from bug reports as a new application concept. For example, for the login management, users use "about:logins" to quickly navigate to "Logins & Passwords" page. This "about:logins" page is a concept not included in the configuration file.

If the concept from the bug report is a new concept, we determine the category of the concept from bug reports by analyzing the atomic steps obtained in section 3.1.4. First, we replace the concept in the atomic step by its placeholder (e.g., "Navigate to the "about:logins" page" → "Navigate to the CONCEPT_0 page"), detailed in Section 3.1.4. Then, we do the dependency parsing for the atomic step using spaCy (disable "merge_noun_chunks" pipe). From the dependency tree, we get the ancestors of the concept with the dependency relation "compound (noun compounds)", "amod (adjectival modifier)", "nmod (nominal modifier)" or "nummod (numeric modifier)" as the concept's candidate categories. For example, for the step "Navigate to the "about:logins" page", we obtain <about:logins, page>, and for the step "switch back to the "about:logins" tab", we obtain <about:logins, tab>. Then, for each concept, we aggregate how many candidate categories this concept has and sort the candidate categories by their frequencies. We select the candidate category with the highest frequency as the category for the concept from bug reports. If more than one candidate categories with the same highest frequency, we choose the one with the higher occurrence in all bug reports. Once we obtain the concept category for the concept from bug reports, we match this concept category with the standard category label using SBERT in the same way as described above. If a match is found, the matched standard category label is used as the concept category. Otherwise, the concept category is *others*.

3.1.4 Constructing Dynamic Part - Scenario Extraction. This step extracts *preconditions*, *steps to reproduce*, *expected behavior* and *observed behavior* from bug reports to construct user task workflow and outcomes. We split the bug description into sections by regular expressions and obtain the four needed sections. Specifically, we summarize a list of section titles for each section, which are used frequently in bug reports, such as "Expected Results", "Expected Outcome", "Expected Behavior" for expected behavior section. In this work, we use the extracted precondition, EB and OB texts as-is in the knowledge graph. But we further process the S2Rs text to obtain detailed steps to facilitate the creation of soap opera tests. The extracted EB and OB will be attached to the last step.

Step Description Normalization. We split the S2Rs text into steps by line changes and sentence splitting using spaCy, and obtain a list of step sentences. The serial number or bullets of steps will be removed. Some steps include more than one user operations connected by coordinating conjunction (CCONJ), such as "Navigate to "about:logins" page and click the "Create New Login" button.". We further split such compound steps into atomic steps.

The concepts (e.g., "about:logins", "Create New Login") in the step sentences increase the complexity of step splitting. Therefore, we normalize these concepts before step splitting. To that end, we construct a dictionary mapping the concepts obtained in constructing static part (Section 3.1.3) into placeholders (i.e., CONCEPT_INDEX, INDEX is the index of concept). For example, if "about:logins" is the first concept in the concept list, its placeholder is CONCEPT_0. Besides concepts, URLs also interfere with sentence parsing. We use the regular expression to identify URLs, and add them into the placeholder dictionary. Based on the placeholder dictionary, we replace concepts and URLs in the step sentence with their placeholders. For example, "Navigate to "about:logins" page and click the "Create New Login" button." will be transformed into "Navigate to CONCEPT_0 page and click the CONCEPT_1 button.". We also discard content within parenthesis to simplify the sentence parsing. After the step splitting, we change placeholders back to corresponding concepts and URLs.

Splitting Compound Steps into Atomic Steps. We employ the Berkeley Neural Parser (benepar) [25, 26] to do the constituency parsing and generate the constituency parse tree for the compound step. We then traverse the leaf nodes of the constituency parse tree to determine whether the step contains coordinating conjunctions. If so, we do the breadth-first search on the constituency parse tree to identify the layer where the coordinating conjunctions are located. From the layer of the coordinating conjunction, we get the sibling nodes of the coordinating conjunction. Then, the parts connected by the coordinating conjunction can be obtained, which can be phrases or clauses. Based on the parts connected by the coordinating conjunction, we split the step into atomic steps. For example, "Navigate to "about:logins" page and click the "Create New Login" button." will be split into "Navigate to "about:logins" page.", "click the "Create New Login" button."; 'Click on the "Edit" button and edit the username and password.' will be split into "Click on the Edit button.", "edit the username.", "edit password.".

Step Clustering. Different bug reports may have the same steps expressed in different ways. For example, "switch back to the about:logins tab." and "Focus back to the about:logins page.", "Open a new tab." and "Open new tab". As shown in Figure 1, clustering such steps together will create a more connected user task workflow graph, which enables more accurate analysis of relevant bug reports (Section 3.2.1) and increases the possibility of creating hypothetical dramas in soap opera testing (Section 3.2.2).

We combine two clustering methods. For the steps having both linked concepts and actions (Section 3.1.5), we cluster them if a) the steps have the same concept or synonym concepts; and b) they have the same action or synonym actions. For example, "switch back to the about:logins tab." in Bug 1678633 and "Focus back to the about:logins page." in Bug 1575516 shown in the red box in Figure 1 are clustered into *Cluster 67*, because these two steps have the *concept* "about:logins" and their *actions* have the *synonymOf* relation (<"switch back to", *synonymOf*, "focus back to">) as shown in the blue box in Figure 1. For other steps, we cluster them by the fast clustering algorithm based on the SBERT sentence similarities, which clusters steps with cosine similarity $\geq$ threshold (i.e., 0.85 in this work) and minimum community size (i.e., 1 in this work). For the steps "Open a new tab" and "Open new tab", which do not have both linked *concepts* and linked *actions*, we cluster them by using the sentence similarity. Since the cosine similarity between them is 0.985, higher than the threshold (i.e., 0.85), they are clustered into *Cluster 1213*, as shown in the black box in Figure 1.

Table 1: Syntax Patterns for Entity Linking

No.	Syntax Pattern	Example	Event
1	verb + nsubj/nsubjpass[1]	Have session store enabled.	<enable,session store, none>
2	verb + dobj[2]	Click the Copy button.	<click, the Copy button, Copy>
3	verb + prt[3] + dobj	Scroll down the Feedback page.	<scroll down, the Feedback page, Feedback>
4	advmod[4] + verb + dobj	Right click Address bar.	<right click, Address bar, Address bar>
5	verb + prep[5] + pobj	Go to about:logins	<go to, about:logins, about:logins>
6	verb + prt + prep + pobj[6]	Scroll down to Applications.	<scroll down to, Applications, Applications>
7	intj[7] + verb + prep + pobj	Right click on the tab header;	<right click on, the tab header, none>
8	verb + advmod + prep + pobj	Focus back to the "about:logins" page.	<focus back to, the "about:logins" page, about:logins>

[1]nominal subject/nominal subject (passive) [2]direct object [3]particle [4]adverbial modifier [5]prepositional modifier [6]object of preposition [7]interjection

3.1.5 Constructing Dynamic Part - Entity Linking. Given an atomic step, we try to identify the concept and action involved in the step and link the step with the concept and action in the static part. To that end, we extract the event tuple <action, target, concept> from the step by eight grammar patterns, as shown in Table 1. These grammar patterns are summarized from a large number of atomic steps, which broadly cover the syntax patterns of steps.

Take "Focus back to the "about:logins" page" as an example. First, we replace the concept with the placeholder (e.g., "about:logins" as CONCEPT_0) for simplifying the analysis of dependency parsing. Then, we parse the step using the dependency parser of spaCy (enable "merge_noun_chunks" to merge noun chunks into a single token, e.g., "the CONCEPT_0 page" will be regarded as a whole). We get the root word as the verb (i.e., "Focus"). By traversing the children of the root word, we identify whether the root word has the direct object (the relation between the root word and children is "dobj (direct object)", "nsubjpass (nominal subject 'passive')" or "nsubj (nominal subject)"). If so, the direct object is the target, e.g., the section 1 (No.1-No.4 examples) in Table 1. Otherwise, we identify if the children of root word contain a preposition (prep). For the root word having a preposition, the preposition object (pobj) is the target, referred to section 2 (No.5-No.7 examples) in Table 1.

We further recognize the children of the root word with the relation "intj" (interjection) or "advmod" (adverbial modifier). We take the children with these relations as the adverbial modifier for the root word, for example, "back" is the right adverbial modifier for "Focus". If there is the right adverbial modifier, we further investigate if the right adverbial modifier has a preposition ("to" is the preposition for "back"). For the root word not having the direct object and preposition, the preposition object (pobj) of the right adverbial modifier is the target (i.e., "the CONCEPT_0 page"). We also check if the root word has the particle (the root word's children with the relation "prt", e.g. "down" to "scroll" in No. 3 and No. 6 examples in Table 1). Then, by merging the root word, adverbial modifier, particle and preposition based on their relative positions, we can get the verb_phrase (i.e., "Focus back to"). For the target extracted from the above, we replace the placeholder by concept, then obtain the target (i.e., "the about:logins page"), and further get the concept (i.e., "about:logins") in the target.

After obtaining the concept and action involved in the step, we link them to the corresponding concept and action in the static part of the KG. The same concept (or action) may be expressed in different ways in the step and in the KG. Similar to Section 3.1.3, we use the SBERT-based synonym finding algorithm to link the concept (or action) in the step to the synonym concept (or action) in the KG. In this link, we also require that the action should be applicable to the category of the matched concept.

3.2 Soap Opera Test Generation

Based on the system knowledge graph of user tasks and failures, we find relevant bug reports to a seed bug and generate soap opera test scenarios for supporting scenario-based exploratory testing.

3.2.1 Finding Relevant Bug Reports. For a seed bug report, we obtain its relevant bug reports based on the number of common steps as a bag between the two bug reports (referred to as "step co-occurrence"). The two bug reports with higher step co-occurrence are more relevant, since they have more overlapping operations. Furthermore, the number of times the steps in the same cluster occurring in different bug reports, referred to as "bug frequency" is another metric to assess the bug relevance. The cluster with the smaller bug frequency is more representative. For example, the steps in the cluster of "Open the Firefox browser" occur in lots of bug reports, and thus have a very high bug frequency. The steps in the cluster of "Navigate to about:logins page" mainly occur in the bugs for the Firefox::about:logins component with a lower bug frequency. We add up the bug frequency of common steps in the two bug reports as bug frequency sum. We first sort the bug reports by the descending order of step co-occurrence. For the bug reports having the same step co-occurrence, we sort them by the bug frequency sum in ascending order; Finally, for bug reports with the same step co-occurrence and bug frequency sum, we sort them in reverse chronological order, assuming that more recent bug reports have higher reference value for testing.

3.2.2 Test Scenario Generation. Next, we generate soap opera test scenarios based on the seed bug report bug_s and one of the relevant bug reports bug_r. We construct test scenarios at two levels, namely step level and scenario level.

For generating the test scenario at the step level, we first obtain the common steps between bug_s and bug_r. Then, for each shared step (i.e., $Step_{share}$), we construct a test scenario as follows: a) get the steps before $Step_{share}$ in bug_s, namely $Step_s$; b) get the steps after $Step_{share}$ in bug_r, namely $Step_r$; c) concatenate the steps $Step_s$, $Step_{share}$ and $Step_r$ as the steps of the created test scenario; d) use the preconditions (if any) from bug_s as the preconditions of the created test scenario and use the EB and OB of bug_r as the EB and OB of the created test scenario. Constructing the test scenario at the scenario level is done as follows: a) create the steps of the test scenario by concatenating the steps from bug_s with the steps from bug_r; b) use the preconditions of bug_s as the preconditions of the created test scenario and use the EB and OB from bug_r as the EB and OB of the created test scenario. By swapping the position of bug_s and bug_r, we get another test scenario.

For each step of the created soap opera test scenario, if this step has been the ending step of some other bugs, the OBs and EBs of these bugs will be attached to the soap opera test step which help the tester attend to additional failures on the way of performing this soap opera test, similar to side quest of a main mission.

3.3 Proof-of-Concept Tool Implementation

We build a proof-of-concept system knowledge graph based on the bug reports of the Mozilla Firefox web browser. Firefox has been actively maintained over 20 years and is a very stable software. This makes exploratory testing an intriguing technique to explore

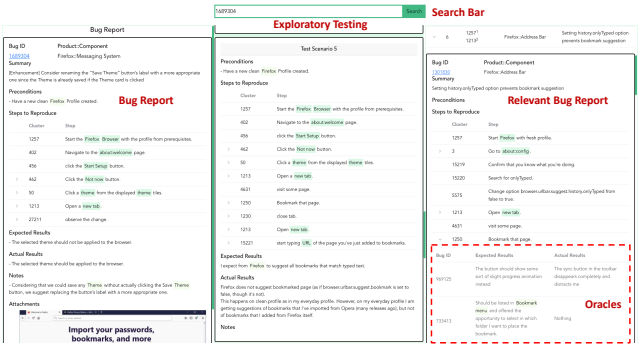


Figure 4: User Interface of Our Exploratory Testing Tool

complex feature interactions and find new bugs that have been overlooked by scripted testing. Firefox has accumulated a large number of bug reports related to various user tasks, system logic and interactions. This makes it feasible to apply our knowledge graph method. We crawl 62,226 Firefox bugs from the Mozilla Bugzilla. We retain the bugs having the S2Rs, EB and OB sections required by our approach. Then, we filter bugs by steps to reproduce as follows: a) remove the bug having steps with more than 64 tokens; b) remove the bug having more than 20 steps. From data observation, we find that if one step has too many tokens, the step tends to describe too many operations in one sentence. Such sentences are too complex to be reliably parsed using existing NLP tools. Moreover, for bugs with more than 20 steps, the S2Rs section is more likely to be a stack trace or log rather than user steps for some tasks. After data filtering, we get 25,939 bug reports to construct our system knowledge graph as described in Section 3.1.

We deploy the system knowledge graph in an online exploratory testing tool, which can be found in our replication package. This tool implements our soap opera testing generation approach. As shown in Figure 4, the tool consists of four panels, namely *Search Bar*, *Bug Report*, *Relevant Bug Reports* and *Exploratory Testing*. After searching the bug id of the seed bug in *Search Bar*, the details of the seed bug are shown in *Bug Report*, e.g., Product::Component, Bug Summary, Steps to Reproduce, Expected Results, Actual Results. The steps in the Steps to Reproduce section contain three fields, i.e., Cluster, Step and Oracles (i.e., EBs and OBs). The Cluster field shows the cluster index from the Step Clustering analysis. For example, the second step in the seed bug shown in Figure 4 belongs to the *Cluster 402*. If a step has oracles, oracles field is indicated by the ⟩ symbol left to the Cluster field, which can be expanded to see the oracles table. The Step field shows the text of atomic step in the KG. The concepts involved in the step are highlighted to give testers some hints. The concepts mentioned in preconditions, EBs and OBs are also highlighted.

Relevant Bug Report displays the seed bug's relevant bug reports ranked by the bug relevance. Each relevant bug is initially folded, but shows four pieces of information. The first column shows the rank position. The second column shows the step cluster(s) shared in both the seed bug and the relevant bug. The superscript of the cluster index is the times of the shared step appearing in the two bugs, for example, there are two common steps in *Cluster 1213* between the seed bug and the relevant bug. The third and fourth columns show the bug's Product::Component and summary. The testers can expand a relevant bug item to see its details. When

selecting a relevant bug report, soap opera test scenarios generated from the seed bug and the relevant bug are shown in *Exploratory Testing* as a list of cards. Each card shows a generated scenario with preconditions, steps to reproduce, and expected and actual results. The details of relevant bugs and soap opera test scenarios are presented in the same format as the seed bug.

4 EVALUATION

This section reports the evaluation of our system knowledge graph and soap opera testing tool to answer three research questions:

RQ1: What is the intrinsic quality of the system knowledge graph constructed from bug reports?

RQ2: How useful is the soap opera testing tool in helping the testers do soap opera testing?

RQ3: How is the quality of soap opera test scenarios generated from the system knowledge graph?

4.1 Intrinsic Quality of KG (RQ1)

The static part of our proof-of-concept system KG contains 12 categories, 50 actions and 3,109 concepts (1,669 from GUI configuration files and the rest from bug reports). The dynamic part contains 25,939 scenarios, which include 112,497 steps (merged into 69,795 clusters), 1,170 Preconditions and 25,939 Expected Behaviors and Observed Behaviors. Relations include 25,939 execute, 1170 satisfy, 86,558 nextStep, 112,497 belongTo, 51,878 leadTo, 3,109 presentedIn, 50 actionOn, 6,176 hasAction, 37,979 hasConcept, 328 synonymOf (concept), 39 synonymOf (action) and 25 antonymOf. We evaluate the intrinsic quality of this system knowledge graph from three aspects, namely automatic concept extraction, scenario extraction and entity linking. As the categories and actions are manually developed, we assume they are correct by construct.

As the system knowledge graph involves a large number of entities and relations, we use a statistical sampling method [47], namely sampling the minimal number *MIN* of entities or relations in the knowledge graph to ensure the estimated accuracy in the samples has the error margin 0.05 at 95% confidence level. Two of the authors first independently evaluate the intrinsic quality of the system knowledge graph. After finishing all the evaluation, they discuss the inconsistencies in the results and finally reach an agreement. Based on the final decisions, we compute the accuracy of sampled entities and relations or use the Likert Scale [33] to evaluate the effectiveness of step clustering. We have inter-rater agreements Cohen's Kappa 71.9%-92.2% for different evaluations.

4.1.1 Automatic Concept Extraction. We randomly sample 384 concepts and their presentedIn relations from the static part to evaluate the performance of concept extraction and category identification. The accuracy of concept extraction and category identification is 0.979 and 0.906 respectively. For the category identification, we find that some concepts are linked to more than one category, e.g., <"Create New Login", presentedIn, text> and <"Create New Login", presentedIn, button>. Both of the relations are right since there is a subtitle called "Create New Login" and a button with the same name on "about:logins" page. For this phenomenon, we plan to add an attribute to further distinguish concepts with the same name but different categories. Furthermore, we get all 328 <concept, synonymOf, concept> pairs to evaluate the quality of synonymOf relations among concepts. The accuracy is 0.835. Based on cosine similarities, some concepts with minor differences but are not the same things are linked. For example, "Tools" and "More tools" are linked by synonymOf but they are two different concepts in the GUI. Context information from bug reports could help to reduce such errors.

4.1.2 Scenario Extraction. As for the scenario extraction, first, we randomly sample 384 compound steps to validity the quality of the compound step splitting by accuracy. The accuracy of compound step splitting is 0.773. Two reasons lead to some wrong splitting. One is that some poorly written sentences in bug reports cannot be parsed correctly. Another is the limitations of compound sentence splitting patterns for complex sentences. For example, we sometimes fail to recognize the coordinating relation represented by commas, such as when splitting "The file can be opened in xpdf, evince and okular under the same operating system.", our pattern erroneously regards "xpdf, evince" as a whole phrase. Second, we extract 384 clusters with at least two steps for evaluating the effectiveness of step clustering by the Likert Scale [33], which 1 is the worst and 5 is the best. 75.5% step clusters are rated 5 and 12.5% are rated 4, and only 5.5% clusters are rated 1 or 2. The low ratings are primarily because clustering based on cosine similarities may cluster some steps with many overlapping words but semantic differences into one cluster. For instance, "Share the Firefox window" and "Share a Firefox profile on a Windows computer" have high cosine similarity (i.e., 0.853) but they are completely different steps. Moreover, erroneously linked concepts or actions in some steps may result in irrelevant steps in a cluster. For example, "Click the login button", "Click on any login entry" and "Click any login" are grouped into a cluster due to the concept misidentification.

4.1.3 Entity Linking. For the entity linking, 384 steps with links to concepts and actions are sampled for estimating the accuracy of hasConcept and hasAction relations. The accuracy of hasConcept relation and hasAction relation are 0.786 and 0.914 respectively. We further analyze the reasons for the relatively low accuracy of hasConcept relation, the main reason is the bug reports we use are written by bug reporters with diverse writing habits, which makes the concept identification and linking a tough task. We will add more syntax patterns to expand the concept expressions that can be handled. Moreover, we misidentify some common expressions as concepts. For example, we erroneously link the "performance" in the step "observe the performance." of Bug 1588767 to the concept <performance, presentedIn, text>. This step is to observe the poor performance when monitoring websocket data, not some displayed text. Therefore, we will take the context of step into consideration to further improve the accuracy of concept linking.

4.2 Usefulness for Soap Opera Testing

We perform a user study for evaluating the usefulness of our soap opera testing tool for supporting exploratory testing.

4.2.1 Baseline. The baseline tool ranks relevant bug reports according to the SBERT-based cosine similarity between the seed bug and other bugs. For ease of use and fair comparison, we make a website interface for the baseline tool, shown in our replication

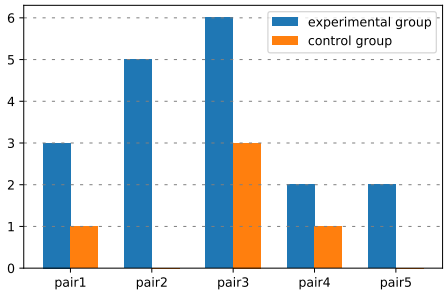


Figure 5: Bug Findings in Our User Study

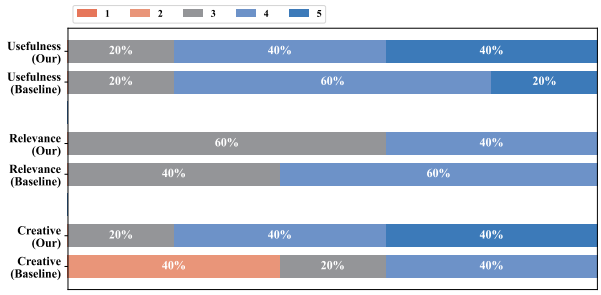


Figure 6: Tool Comparison (5-point Likert: 1 worst, 5 best)

package. The baseline tool uses the bug id as the input, then displays the seed bug at left and recommends relevant bug reports at right in the interface.

4.2.2 Pre-survey. We recruit 10 students (6 Ph.D. students, 2 MS students and 2 Undergraduate students). We conduct a pre-study survey to learn about the participant's programming experience, if having used Firefox (Desktop) browser, how familiar with Firefox (Desktop) browser, testing experience, whether reporting bugs for open source community and how many. The participant's programming experience is from 1 year to 15 years. All of them have used the Firefox (Desktop) browser with different levels of familiarity. Only two of them have testing experience and three of them have reported bugs to open source community (3 bugs, 8 bugs and more than 10 bugs respectively). Based on these information, we make two of them as a pair with the similar background (one is in the experimental group, the other is in the control group).

4.2.3 Tasks. Participants perform soap opera testing on the Firefox (Desktop) browser aiming at finding as many new bugs as possible in 2 hours, for participants in the experimental group supported by our tool and for participants in the control group supported by the baseline. We randomly select 5 bugs from our dataset as seed bugs. Each participant is assigned two seed bugs, and participants in a pair get the same seed bugs. For each seed bug, it is used by two participants in a group. Moreover, for each seed bug, participants may use top-10 relevant bugs for soap opera testing. Participants in the experimental group can use the test scenarios generated from the seed bug and the relevant bug by our approach directly.

4.2.4 Results. As shown in Figure 5, participants in the experimental group find 18 new bugs. After reporting these bugs, 10 bugs have been confirmed, and 1 bug has been fixed. 1 bug has been marked as duplicate and 2 as invalid bugs. For the rest 4 unconfirmed bugs, 2 bugs have been set the priority and severity, 1 bug has been assigned, and for 1 bug causing crash, some developers commented for reproducing the bug and analyzing the crash reasons. For participants in the control group, 5 bugs have been found. Among them, 2 bugs have been confirmed, 1 bug is an invalid bug, 1 bug is still unconfirmed but set the priority and severity, and 1 bug is duplicated with the crash bug found by the experimental group. From the experimental results, our approach having explicit exploratory test scenario support achieves the superiority in bug findings, helping participants (most of them have little testing experience) find 18 bugs in 2 hours on a mature and stable software Firefox web browser. For participants with the same seed bugs, they found different bugs (only 2 bugs are duplicate among the 23 bugs found by the two groups). This shows an excellent advantage of exploratory testing for exploring new scenarios and finding diverse bugs.

4.2.5 Post-survey. To understand the perceptions of participants on our tool and the baseline, we conduct a post-survey, which involves three aspects, namely the usefulness of tools, relevance about seed bug and relevant bugs, whether tools inspire creative thinking or exploring more steps during testing. We use a 5-point Likert Scale with 1 being the worst and 5 being the best. As shown in Figure 6, for the usefulness, both our approach (3, 4, 4, 5, 5) and the baseline (3, 4, 4, 4, 5) are rated useful for participants to do exploratory testing. Even though the baseline only shows similar bug reports without any further processing. This further confirms that bug reports contain useful scenarios and oracles for exploratory testing, even in their raw form. Moreover, all participants in both groups think that the tools can help them do testing based on scenarios instead of random testing from just the seed bugs.

For the relevance between the seed bug and relevant bugs, the Likert scope in our approach (3, 3, 3, 4, 4) is slightly lower than the baseline (3, 3, 4, 4, 4). The baseline is based on the cosine similarities of bug summary and description as a whole, while our approach considers step overlapping between scenarios. Therefore, it is normal for our approach having lower relevance ratings. Based on this phenomenon, we add a follow-up question (Do you think the tool need to improve the relevance between seed bug report and relevant bug reports? why?). 60% participants from both groups think that there is no need to improve the relevance. The reason from the baseline group is that the relevance is good enough. However, the reason from the experimental group is that the seed bug and relevant bugs have some relevance (depth) but still keep some difference (breadth). This difference is important for finding new bugs. As for the other 40% participants, the control group participants who want to improve the relevance comment "It would be nice if there were more similar patterns between the seed bug report and relevant bug reports (like reproducing steps, or all focusing on a certain component)", and "the relevance mostly focuses on the same component, but the steps to reproduce may lack the relevance". For participants in the experimental group, they want to improve the relevance to reduce the duplication among test scenarios to make the test scenarios more concise and easy to read.

For creativity, we see our approach (3, 4, 4, 5, 5) is much better to inspire creative thinking and step exploration than the baseline (2, 2, 3, 4, 4). Participants in the control group state that due to the difficulty in connecting the seed bug with relevant bugs by

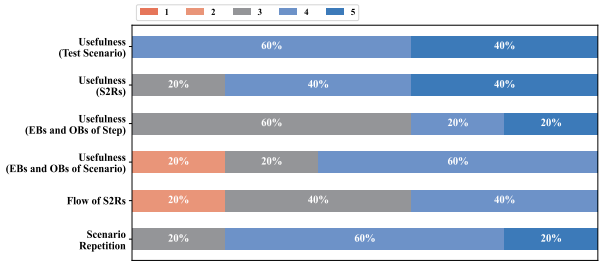


Figure 7: Test Scenario Quality (5-point Likert Scale)

themselves, they tend to test by using bugs separately, such that their thinking will be imprisoned by the limited operations of a single bug report, which makes it hard to think beyond the present scenario in a bug report in a more creative way.

4.3 Quality of Generated Test Scenarios

To assess the quality of test scenarios generated by our approach and their effectiveness for supporting soap opera testing, we conduct a follow-up survey for the experimental group. The participants were asked to evaluate the test scenarios by using the 5-point Likert Scale [33] from the three aspects: usefulness, flow of S2Rs and scenario repetition. After the survey, we interview each participant on the experiment results to reveal the reasons behind their ratings.

4.3.1 Usefulness. As shown in Figure 7, all of the participants think that the generated test scenarios are useful and helpful for their exploratory testing, namely three of them rated 4 (moderately useful) and two rated 5 (useful). We also survey the usefulness of specific parts in the generated test scenarios, namely S2Rs, EBs and OBs of step, as well as EBs and OBs of scenario.

S2Rs. The ratings of S2Rs in test scenarios by the 5 participants are 3, 4, 4, 5, 5. Participants think the S2Rs tell them what and how to test, which plays an essential guiding role during exploratory testing. Besides, the generated scenarios have already combined the scenario from the seed bug and the scenarios from relevant bugs, which can save the time for creative thinking and inspiring more exploratory paths. The soap opera embedded in test scenarios can help to reveal hidden bugs in depth, not superficial ones.

The participant who gives 3 ratings agrees that the test scenarios are useful for testing, but some S2Rs in test scenarios are too detailed. She thinks that it is better to summarize the key information. For example, one seed bug of the participant Bug 1679131 is about "icon and close button icons can be dragged on Remove all confirmation modal". By combining this bug with its relevant bug Bug 1243333, a test scenario is generated, namely drag elements on "Remove this login?" confirmation modal. Then, the participant finds a new bug Bug 1764787, "icon and close button icon can be dragged on Remove this login? confirmation modal". Although the detailed S2Rs are helpful, she proposes it would be better to give her the key information, e.g., "drag elements" instead of detailed S2Rs. During the interview, we find that Firefox (Desktop) browser is her usual browser. She is familiar with the operation on Firefox. Therefore, for the detail level of test scenarios generation, we need to take the experience of tester into consideration, namely the familiarity with the software under test and testing expertise.

EBs and OBs of Step. The ratings of the EBs and OBs attached to the step are 3, 3, 3, 4, 5. Two participants say that this part plays a great role in finding bugs, as they provide important hints for recognizing bugs. However, almost all participants mentioned that the information overload problem in this part affects its effectiveness, namely some steps having too many EBs and OBs attached. This is also our concern when designing the tool. To alleviate the problem, we highlight the concepts in the EBs and OBs to emphasize the important concepts involved. Although participants agree these concept highlights are useful, more concise and informative EBs and OBs presentation is needed to further address this problem.

EBs and OBs of Scenario. The ratings of EBs and OBs for the test scenario are 2, 3, 4, 4, 4. For participants with the 4 ratings, they think that EBs and OBs of scenario tell them where they should focus on instead of testing without aims, which keeps their desire to explore and test. However, the other two participants think there is no necessary connection between the S2Rs and the EBs and OBs of scenario in some test scenarios. Inspired by this feedback, we would generate EBs and OBs of scenario based on the S2Rs of test scenario by training a text generation model with the knowledge in our KG instead of directly copying the EBs and OBs from original bug reports to improve the connection between S2Rs and EBs/OBs.

4.3.2 Flow of S2Rs. For the logical flow among steps in the generated test scenarios, the ratings are 2, 3, 3, 4, 4. The participant giving 2 ratings states that S2Rs (especially joint by some general steps) in some test scenarios are less logical and even meaningless. For example, the "Click on the [...] button" step in one of his seed bugs Bug 1679131 is a general step, as many parts of the software have "..." button, for example the Top-1 relevant Bug 1441196 of the seed bug. But, the "..." menu of these two bugs are irrelevant. As a result, the steps chained by clicking the "..." button are irrelevant, which leads to the meaningless scenario connected by this common step. However, the other participant giving 4 ratings has different opinion. He thinks that the general step can make the generated test scenarios more drama so that he may find unexpected bugs. For example, Bug 1689304 is one of his seed bugs. By the test scenario generated by the general step "Open a new tab.", he finds the new bug Bug 1766384, detailed in Section 2.3.2. We believe both options make sense. It is an important future work to fully utilize advantages and avoid disadvantages of general steps.

The two participants giving 3 ratings think some steps are redundant and unnecessary in the generated test scenarios, especially those generated at the scenario level. For example, "Launch Firefox" in a test scenario generated at the scenario level always occurs twice, which is meaningless and affects the coherence of steps. As our scenario-level test scenario generation method simply connects the scenarios from the seed bug and the relevant bug, the generated test scenarios often have redundant or unnecessary steps affecting the testing experience. Based on this feedback, strengthening the coherence of the S2Rs of the generated test scenarios and simplifying the test scenarios is an essential part of our future work.

4.3.3 Scenario Repetition. For the degree of repetition among the generated test scenarios, the ratings are 3, 4, 4, 4, 5 (1 is no repetition and 5 is high repetition). Almost all of the participants think the generated test scenarios have repetitions and the repetitions need to be reduced. This is a potential issue we were aware of when

designing the tool. Nonetheless, surprisingly two participants consider a certain degree of repetitions can help them familiarize with the software under test, inspire creative thinking for finding bugs. When repeating similar test scenarios, as the participants gradually deepen their understanding of the system under test, they often generate some new thinking, stimulate some new exploration, and thus discover new bugs. Our current soap opera test scenarios correspond to workflow paths in our knowledge graph, which inevitably have many repetitions. In addition to presenting workflow paths individually, we could present an interactive workflow graph of relevant bugs (e.g., Figure 1). This would allow testers to get a holistic view of commonalities and relationships among test scenarios and effectively explore the large scenario space.

5 RELATED WORK

Much research has been focused on test case generation from specifications [2, 46, 54, 59], program differences [9, 13], or using fuzzing techniques [16, 40], focusing mainly on exposing data and logic failures. Test cases can also be generated from GUI runtime models [14, 49, 55], focusing mainly on functional testing and crash bugs. In spite of a great deal of research focusing on test automation, manual software testing is unlikely to be replaced by automated testing in the foreseeable future [7, 19, 20, 27–29, 34, 39, 43]. Manual exploratory testing has been shown to be effective at system testing level [3, 4, 12, 20, 24, 37, 53, 57]. Principles and guidelines [3, 10, 11, 37, 56] have been proposed for performing and managing exploratory testing. Our tool is the first to automatically generate exploratory test scenarios based on the knowledge of user tasks and failures in bug reports. The generated scenarios help to find functional, usability and security bugs (see Section 2).

Exploratory testing leverages the tester's personal knowledge of domain, system and users [20]. Our approach supports exploratory testing by the crowd knowledge of user tasks and failures distilled from bug reports, which would enrich the tester's personal knowledge with realistic user tasks and past failures. In some sense, our approach bears the same spirit as crowd testing [30, 38]. Crowd testing [38] mines user action sequential patterns from recorded low-level events for enhancing automated testing. In contrast, our approach extracts a system knowledge graph from the textual content of bug reports for supporting exploratory testing.

Techniques have been proposed to reproduce the bugs from the S2Rs description of bug reports [15, 61] or bug video [6, 17]. However, these techniques do not generate new tests from bug reports. BUGINE [32, 52] is similar to the baseline tool in our user study. It recommends bugs in some apps to the developers of similar apps. Unlike our approach, BUGINE does not extract any knowledge from the recommended bugs, and the tester takes the sole responsibility to understand the bug and design test scenarios in other similar apps. Furthermore, it does not explore the interactions between bug reports to automatically compose hypothetical scenarios. Different from existing duplicate bug detection [18, 41], our approach finds relevant bugs with shared steps but relevant bugs may not be overall similar. This enables the creation of unexpected test "dramas".

Some recent work constructs knowledge graphs from API documentation and source code to support entity-centric search of APIs and programming tasks [31, 35, 51], comparison of similar APIs [36], and API misuse detection [45]. Su et al. [50] constructs a knowledge graph of bug-component triaging history to reduce bug triaging confusion. The entities in this KG are software components and the relations are the bug tossing relations between components. In contrast, our knowledge graph models fine-grained information of user tasks and failures extracted from bug reports.

6 THREATS TO VALIDITY

The quality evaluation of the constructed KG may be subject to human bias. Two authors performed evaluation independently and obtained substantial inter-rater agreement. Our approach assumes the explicit sections of S2Rs, OBs and EBs in bug reports. For those without explicit sections, techniques [48, 60] can be used to classify bug report content. A major external threat is our approach has been evaluated on only one subject system (Mozilla Firefox). The generalizability of our open information extraction methods requires further validation on more systems. Our user study shows promising results, but it is small scale. We release our KG and exploratory testing tool for community evaluation.

7 CONCLUSIONS AND FUTURE WORK

This paper presents an automatic approach for generating soap opera test scenarios. The approach is built on a novel system knowledge graph of user tasks and failures extracted from the textual context of bug reports. This knowledge graph allows real-world scenarios and oracles in bug reports to be explored in a creative manner and to be combined into hypothetical scenarios for inspiring expert uses of the system. We build a proof-of-concept KG from a large number of Mozilla Firefox bugs, and confirm the effectiveness of our open information extraction methods in face of noisy bug report content. Through a small-scale user study, we demonstrate the usefulness of our soap opera test generation method for effective and efficient exploratory testing. During our tool development and user study, we reported 64 bugs for Firefox, many of which have been confirmed and fixed. Some of these bugs ignite more discussions within the Mozilla team about some fundamental system design, usability and security issues. These outcomes provide external usefulness evidence and confirmation of our approach. In the future, we will enhance our KG construction with more robust information extraction methods and improve our test scenario generation and presentation inspired by our user study. Moreover, supporting exploratory testing with a knowledge graph is a general idea, which could be used for various testing tasks and data sources, especially for systems including complex logic and services. For example, the knowledge graph could also be derived from run-time logs for online services and be learned from the deployed experiments for the A/B testing.

REFERENCES

[1] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th international conference on Software engineering*. 361–370.
[2] 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.
[3] Jonathan Bach. 2000. Session-based test management. *Software Testing and Quality Engineering Magazine* 2, 6 (2000), 32–37.
[4] James Bach. 2004. Exploratory testing. *The testing practitioner* (2004), 253–265.

[5] Victor R Basili and Richard W Selby. 1987. Comparing the effectiveness of software testing strategies. *IEEE transactions on software engineering* 12 (1987), 1278–1296.

[6] Carlos Bernal-Cárdenas, Nathan Cooper, Kevin Moran, Oscar Chaparro, Andrian Marcus, and Denys Poshyvanyk. 2020. Translating video recordings of mobile app usages into replayable scenarios. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 309–321.

[7] Stefan Berner, Roland Weber, and Rudolf K Keller. 2005. Observations and lessons learned from automated testing. In *Proceedings of the 27th international conference on Software engineering*. 571–579.

[8] Nicolas Bettenburg, Sascha Just, Adrian Schröter, Cathrin Weiss, Rahul Premraj, and Thomas Zimmermann. 2008. What makes a good bug report?. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. 308–318.

[9] 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.

[10] Hans Buwalda. 2004. Soap opera testing. *Better Software* 6, 2 (2004), 30–37.

[11] JD Cem Kaner. 2013. An introduction to scenario testing. *Florida Institute of Technology, Melbourne* (2013), 1–13.

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

[13] 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.

[14] 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.

[15] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 141–152.

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

[17] Madeleine Havranek, Carlos Bernal-Cárdenas, Nathan Cooper, Oscar Chaparro, Denys Poshyvnayk, and Kevin Moran. 2021. V2S: a tool for translating video recordings of mobile app usages into replayable scenarios. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 65–68.

[18] Jianjun He, Ling Xu, Meng Yan, Xin Xia, and Yan Lei. 2020. Duplicate bug report detection using dual-channel convolutional neural networks. In *Proceedings of the 28th International Conference on Program Comprehension*. 117–127.

[19] 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, 2 (2014), 303–342.

[20] 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.

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

[22] Nicholas Jalbert and Westley Weimer. 2008. Automated duplicate detection for bug tracking systems. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. IEEE, 52–61.

[23] Cem Kaner. 2008. A tutorial in exploratory testing. *Tutorial presented at QUEST2008.(Available online at: http://www. kaner. com/pdfs/QAIExploring. pdf, accessed: 26 Jan 2014)* (2008).

[24] Cem Kaner, Jack Falk, and Hung Q Nguyen. 1999. *Testing computer software*. John Wiley & Sons.

[25] Nikita Kitaev, Steven Cao, and Dan Klein. 2019. Multilingual Constituency Parsing with Self-Attention and Pre-Training. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Florence, Italy, 3499–3505. https://doi.org/10.18653/v1/P19-1340

[26] Nikita Kitaev and Dan Klein. 2018. Constituency Parsing with a Self-Attentive Encoder. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Melbourne, Australia, 2676–2686. https://doi.org/10.18653/v1/P18-1249

[27] Pavneet Singh Kochhar, Ferdian Thung, Nachiappan Nagappan, Thomas Zimmermann, and David Lo. 2015. Understanding the test automation culture of app developers. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 1–10.

[28] Gerald Kowalski. 2010. *Information retrieval architecture and algorithms*. Springer Science & Business Media.

[29] Daniel E Krutz, Samuel A Malachowsky, and Thomas Reichlmayr. 2014. Using a real world project in a software testing course. In *Proceedings of the 45th ACM technical symposium on Computer science education*. 49–54.

[30] Haoyu Li, Chunrong Fang, Zhibin Wei, and Zhenyu Chen. 2019. CoCoTest: collaborative crowdsourced testing for Android applications. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 390–393.

[31] Hongwei Li, Sirui Li, Jiamou Sun, Zhenchang Xing, Xin Peng, Mingwei Liu, and Xuejiao Zhao. 2018. Improving api caveats accessibility by mining api caveats knowledge graph. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 183–193.

[32] 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.

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

[34] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622.

[35] Mingwei Liu, Xin Peng, Andrian Marcus, Zhenchang Xing, Wenkai Xie, Shuangshuang Xing, and Yang Liu. 2019. Generating query-specific class API summaries. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 120–130.

[36] Yang Liu, Mingwei Liu, Xin Peng, Christoph Treude, Zhenchang Xing, and Xiaoxin Zhang. 2020. Generating concept based API element comparison using a knowledge graph. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. 834–845.

[37] James Lyndsay and Neil Van Eeden. 2003. Adventures in session-based testing. *Workroom Productions Ltd. May* 27 (2003).

[38] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 16–26.

[39] David Martin, John Rooksby, Mark Rouncefield, and Ian Sommerville. 2007. 'Good'organisational reasons for'Bad'software testing: An ethnographic study of testing in a small software company. In *29th international conference on software engineering (ICSE'07)*. IEEE, 602–611.

[40] 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.

[41] Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, David Lo, and Chengnian Sun. 2012. Duplicate bug report detection with a combination of information retrieval and topic modeling. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 70–79.

[42] 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.

[43] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 36–42.

[44] Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence Embeddings using Siamese BERT-Networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics. https://arxiv.org/abs/1908.10084

[45] Xiaoxue Ren, Xinyuan Ye, Zhenchang Xing, Xin Xia, Xiwei Xu, Liming Zhu, and Jianling Sun. 2020. API-misuse detection driven by fine-grained API-constraint knowledge graph. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 461–472.

[46] 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.

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

[48] 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.

[49] 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.

[50] Yanqi Su, Zhenchang Xing, Xin Peng, Xin Xia, Chong Wang, Xiwei Xu, and Liming Zhu. 2021. Reducing Bug Triaging Confusion by Learning from Mistakes with a Bug Tossing Knowledge Graph. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 191–202.

[51] Jiamou Sun, Zhenchang Xing, Rui Chu, Heilai Bai, Jinshui Wang, and Xin Peng. 2019. Know-how in programming tasks: From textual tutorials to task-oriented knowledge graph. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 257–268.

[52] 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.

[53] Jarle Våga and Ståle Amland. 2002. Managing high-speed web testing. In *Software quality and software testing in internet times*. Springer, 23–30.

[54] 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.

[55] 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.

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

[57] Bill Wood and David James. 2003. Applying session-based testing to medical software. *Medical Device and Diagnostic Industry* 25, 5 (2003), 90–103.

[58] Xin Xia, David Lo, Ying Ding, Jafar M Al-Kofahi, Tien N Nguyen, and Xinyu Wang. 2016. Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering* 43, 3 (2016), 272–297.

[59] Arvin Zakeriyan, Ramtin Khosravi, Hadi Safari, and Ehsan Khamespanah. 2021. Towards automatic test case generation for industrial software systems based on functional specifications. In *International Conference on Fundamentals of Software Engineering*. Springer, 199–214.

[60] Yu Zhao, Kye Miller, Tingting Yu, Wei Zheng, and Minchao Pu. 2019. Automatically extracting bug reproducing steps from android bug reports. In *International Conference on Software and Systems Reuse*. Springer, 100–111.

[61] Yu Zhao, Tingting Yu, Ting Su, Yang Liu, Wei Zheng, Jingzhi Zhang, and William GJ Halfond. 2019. Recdroid: automatically reproducing android application crashes from bug reports. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 128–139.

[62] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 14–24.