AUTOMATED ANALYSIS OF BUG DESCRIPTIONS TO SUPPORT BUG REPORTING AND RESOLUTION

by

Oscar Javier Chaparro Arenas

To my parents, for their inspiration and endless support.

To my wife, for her unconditional love and sacrifice.

AUTOMATED ANALYSIS OF BUG DESCRIPTIONS TO SUPPORT BUG REPORTING AND RESOLUTION

by

OSCAR JAVIER CHAPARRO ARENAS, BEng, MEng

DISSERTATION

Presented to the Faculty of

The University of Texas at Dallas

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY IN

SOFTWARE ENGINEERING

THE UNIVERSITY OF TEXAS AT DALLAS

August 2019

ACKNOWLEDGMENTS

This dissertation is the result of constant learning, work, dedication, and sacrifice. Many people were involved one way or another, and I am extremely grateful to them for their unconditional support and advice.

My deepest gratitude goes to my advisor, teacher, mentor, and friend, Dr. Andrian Marcus, for his invaluable lessons, guidance, and encouragement during my graduate studies. He opened my eyes about what research in software engineering is all about, and taught me how exciting and challenging academia truly is. He taught me to always keep pushing my limits, look beyond the details and the obvious, and follow a practical approach in research and teaching, and also in life. This achievement would not have been possible without his amazing feedback and mentorship. Thank you very much, Andi.

I also want to thank the current and past members of our research lab, Juan Florez, Laura Moreno, and Venera Arnaoudova, for their feedback, help, and support during all these years. Many thanks to my PhD committee members, Dr. Vincent Ng, Dr. Tien Nguyen, and Dr. Lingming Zhang, for their invaluable feedback to my work.

I would like to thank Dr. Massimiliano Di Penta, Dr. Denys Poshyvanyk, Dr. Martin Robillard, and Dr. Tom Zimmermann for taking the time and effort to write and send reference letters for me during my job search.

It was my pleasure to collaborate with extraordinary researchers from around the world. My sincere gratitude to my collaborators Jing Lu, Vincent Ng, Fiorella Zampetti, Massimiliano Di Penta, Gabriele Bavota, Carlos Bernal, Kevin Moran, Denys Poshyvanyk, and many others. My special thanks to Massimiliano Di Penta and Gabriele Bavota for their hospitality and amazing support during my visit to their research labs in Italy.

I also want to express my profound gratitude to my parents, who inspire me to continuously grow as a person; to my wife, who encourages and supports me in good and bad times; and to my brothers, who show me different perspectives in life. Thank you all for your endless love and support. I also want to thank my friends, who have been by my side one way or another.

Many thanks to all the people involved in this dissertation and my graduate studies, including the faculty and staff members from the Computer Science department at The University of Texas at Dallas and Wayne State University.

Thank you all for making this journey one of the most enriching and rewarding experiences of my life.

June 2019

AUTOMATED ANALYSIS OF BUG DESCRIPTIONS

TO SUPPORT BUG REPORTING AND RESOLUTION

Oscar Javier Chaparro Arenas, PhD
The University of Texas at Dallas, 2019

Supervising Professor: Dr. Andrian Marcus, Chair

User-written bug descriptions are the main information source for software developers to triage and fix the reported software bugs. Unfortunately, bug descriptions are often unclear, ambiguous, and miss critical information. In consequence, developers spend excessive time and effort triaging and solving the bugs, and, in many cases, they are unable to reproduce the problems, let alone fix the bugs in the code. Current bug reporting technology, which is mostly passive and does not verify the information provided by the users, provides little help in collecting high-quality bug information and improving the quality of bug descriptions.

This research aims at improving: (1) the quality of bug descriptions, and (2) the accuracy of bug resolution tasks that rely on bug descriptions. The approach towards achieving our goals is the automatic analysis of bug descriptions in combination with software analysis techniques. In the context of our goals, we focus on two bug resolution tasks, namely, automated duplicate bug report detection and bug localization in source code. In addition, we focus on analyzing the main information in bug descriptions, namely, the unexpected software behavior (*i.e.*, the *observed behavior*), the steps to reproduce such (mis)behavior (*i.e.*, the *steps to reproduce*), and the normal software behavior (*i.e.*, the *expected behavior*).

In this dissertation, we propose automatic techniques to identify, verify, and leverage the *observed behavior*, the *expected behavior*, and the *steps to reproduce* from bug descriptions, for generating automated feedback to reporters about problems in their bug descriptions, and for improving the accuracy of bug resolution tasks. Specifically, we propose techniques and present empirical results on (i) discovering discourse patterns used by reporters to describe bugs; (ii) automatically detecting missing information in bug descriptions; (iii) automatically assessing the quality of the steps to reproduce provided in such descriptions; and (iv) leveraging bug descriptions and query reformulation to improve automated bug localization and duplicate bug report detection.

The proposed techniques aim to reduce the effort and time devoted by developers when triaging and solving software bugs. We expect that the feedback provided by these techniques will help write higher-quality bug descriptions, while facilitating the detection of duplicate bug reports and the localization of the reported bugs in source code.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Problem Description

When software does not behave as expected, users and developers report the problems using bug/issue trackers [250]. Specifically, problems are frequently reported as *bug reports*, documents that describe software bugs and are expected to contain the information needed by the developers to triage and fix the bugs in the software [249, 237]. While much of the information in bug reports is structured (*e.g.*, operating system or system version affected by the bug), the main content of a bug report is unstructured, that is, expressed in natural language [249, 98, 198]. Natural-language bug descriptions[1], produced by reporters, are meant to include the unexpected software behavior, observed by the user (*i.e.*, the Observed Behavior or OB); the steps followed by the user to reproduce the (mis)behavior (*i.e.*, the Steps to Reproduce or S2Rs); and the normal software behavior (*i.e.*, Expected Behavior or EB). These three types of information in bug descriptions are essential for developers to triage and fix the reported bugs [249].

While considered extremely useful, bug descriptions are often unclear, incomplete, and ambiguous [1, 108, 249, 248, 73, 115]. Recently, developers from more than 1.3k open-source projects wrote a letter to GitHub expressing their frustration that essential information (*e.g.*, S2Rs) is often missing in bug descriptions [1]. In addition, developers ask for a solution that would make users report high-quality information in such descriptions.

Low-quality bug descriptions negatively impact users and developers in many ways during bug triage and resolution. Low-quality bug content is one of the main reasons for non-reproducible bugs [108], unfixed bugs [248], and additional bug triage effort, for example, when checking for duplicate bug reports [73], as developers have to spend extra time

[1] In this dissertation, we use *bug reports* and *bug descriptions* interchangeably, unless otherwise noted.

and effort understanding the reported bugs or asking for clarifications and additional information [108, 73]. Low-quality bug descriptions are also likely to gain low attention by developers [115], and contain noisy information that hinders automated bug localization and duplicate bug report detection approaches [90]. As indicated by developers, low-quality information in bug descriptions is the main cause for delay on bug localization and fixing [249, 248]. Low-quality bug descriptions are also the main problem with automated approaches attempting to generate test cases and reproduce the described bugs using bug reports [131, 109, 242].

One of the main reasons for low-quality content in bug descriptions is inadequate tool support for bug reporting [1, 249, 108]. In the aforementioned GitHub petition [1], developers called for improvements to GitHub's issue tracker to ensure that essential information is reported by the users. This problem extends to other bug/issue tracking systems, which capture unstructured bug descriptions through web forms without any content verification or enforcement. Some bug tracking systems (*e.g.*, Bugzilla in the Mozilla Firefox project [10]) provide semi-structured reporting of natural language information, using separate text fields or predefined text templates that explicitly ask for the OB, EB, and S2Rs. Such a solution is insufficient to address the problem, as it does not guarantee that reporters will provide high-quality information as expected.

Little research has been conducted to help users improve the quality of their bug descriptions. Zimmerman *et al.* [249] proposed an approach that predicts the quality level of bug reports. However, this approach does not provide actionable recommendations to reporters on how to improve their bug descriptions. Moran *et al.* [164] developed a technique to augment the steps to reproduce in bug reports via screenshots and graphical component images. Unfortunately, this approach does not focus on improving the textual content of bug descriptions. More recently, the same authors proposed an approach that finds, reproduces, and reports potential crashes in mobile applications without any user intervention [163].

However, this approach is not intended to support users when textually describing software bugs. Despite these attempts to support users on bug reporting, the fact remains that *bug descriptions are of low-quality and there is limited tool support to improve their quality.*

1.2 Research Goal

Our end goal is to support users and developers during bug reporting and resolution, by improving: (1) the quality of bug descriptions and (2) the accuracy of bug resolution tasks that rely on bug descriptions. The approach towards achieving our goal is the automatic analysis of bug descriptions in combination with software analysis techniques. In the context of our research, we focus on two bug resolution tasks, namely, duplicate bug report detection and bug localization in source code. In addition, we focus on analyzing the main information in bug descriptions, namely, the observed behavior, the expected behavior, and the steps to reproduce the bug.

Towards achieving our end goal, our research addresses the following research challenges:

RC1. Understand how users describe software bugs in bug descriptions.

RC2. Automatically identify the main information of bug descriptions.

RC3. Automatically assess the quality of bug descriptions.

RC4. Determine the most relevant information from bug descriptions with respect to bug resolution tasks.

We address the third challenge (*i.e.*, RC3.) towards improving the quality of bug descriptions, and the fourth challenge (*i.e.*, RC4.) towards improving the accuracy of bug localization and duplicate bug report detection techniques. The first two challenges (*i.e.*, RC1. and RC2.) are addressed toward determining how automated textual analysis of bug descriptions can be used to overcome the remaining two challenges.

1.3 Contributions

This dissertation makes the following research contributions:

1. *A catalog of discourse patterns that reporters use for describing bugs.* We verified the hypothesis that reporters use a well-defined set of *discourse patterns* to express the OB, EB, and S2Rs in bug descriptions. Discourse patterns are linguistic rules that capture the syntax and semantics of sentences/paragraphs that convey an OB, EB, or S2R. We verified the hypothesis by conducting a qualitative study based on open coding on nearly 1.1k bug reports from nine software projects. The study resulted in a catalog of 154 discourse patterns that reporters recurrently use to describe the OB, EB, and S2Rs in bug descriptions. From this set, only a handful of patterns (*i.e.*, 14.3% or 22) are used to describe the OB, EB, and S2Rs in most of the analyzed bug descriptions (*i.e.*, 82% on average across projects). The results indicate that bug descriptions present textual regularities that enable the automatic detection and analysis of the OB, EB, and S2Rs. This work targets the first research challenge (*i.e.*, RC1.) and is detailed in Chapter 3.

2. *An automated approach for detecting missing information in bug descriptions.* We proposed an approach (called DeMIBuD) that automatically detects missing EB and S2Rs in bug descriptions. DeMIBuD leverages the inferred discourse patterns and combines Natural Language Processing (NLP) and Machine Learning (ML) techniques. DeMIBuD is meant to alert reporters about missing EB and S2Rs, so that they can provide such information while they are writing and submitting their bug reports. We evaluated DeMIBuD on 1.8k bug reports from nine software systems to determine its detection accuracy. The results indicate that DeMIBuD, in its best setting, detects missing EB (S2R) with 85.9% (69.2%) average precision and 93.2% (83%) average recall. We also found that the best version of DeMIBuD can be deployed in new

projects without the need of being retrained, in which case, similar accuracy levels are expected. This work targets the second and third research challenges (*i.e.*, RC2. and RC3.). Chapter 4 provides further details of the approach and its evaluation.

3. *An automated approach for assessing the quality of the steps to reproduce in bug descriptions.* We proposed an approach (called Euler) that automatically analyzes a bug description, identifies the provided S2Rs, assesses the quality of each step, and generates actionable feedback to reporters about ambiguous steps, steps described with unexpected vocabulary, and steps missing in the bug description. This feedback is expected to help reporters improve their bug descriptions when they are writing and submitting them via the issue/bug tracker. Euler combines NLP, ML, dynamic system analysis, and textual matching techniques to automatically identify the S2Rs and assign to each S2R a set of quality annotations with specific feedback to the reporter. The feedback provided by Euler was assessed by external evaluators for 24 bug reports of six Android applications. The results indicate that Euler correctly identified 98% of the existing S2Rs and inferred 58% of the missing ones (with a 31% precision), while 73% of its quality annotations are correct. The evaluators agree that Euler is potentially useful in helping reporters improve their bug reports. This work targets the second and third research challenges (*i.e.*, RC2. and RC3.). Chapter 5 provides further details about Euler and its evaluation.

4. *A query reformulation approach based on bug descriptions to improve bug resolution tasks.* We proposed an approach to reformulate initial queries that fail to retrieve the relevant software documents when using Text Retrieval (TR)-based techniques for automated bug localization (*a.k.a.* TRBL) and duplicate bug report detection (*a.k.a.* TRDD). Within this approach, we proposed a set of query reduction strategies that retain the most relevant content of the bug description as a (new) query and discard

the remaining, irrelevant content with respect to TRBL and TRDD. We empirically evaluated the effectiveness of the queries reduced by the proposed strategies compared to initial queries formulated from the entire bug description, using state-of-the-art TR-based techniques on a large set of bug descriptions from different software systems. The results reveal that these strategies significantly improve the detection performance of existing techniques across multiple data sets and document granularities. This work targets the fourth research challenge (RC4.). We describe the query reformulation strategies and their evaluation in Chapters 6 and 7.

The proposed techniques aim to reduce the effort and time devoted by developers when triaging and solving software bugs. We expect that the feedback provided by these techniques will help write higher-quality bug descriptions, while facilitating the detection of duplicate bug reports and the localization of the reported bugs in source code.

1.4 Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we present an overview of existing methods of bug reporting, bug report quality assessment, textual analysis of bug reports, and text-retrieval-based duplicate bug report detection and bug localization. Chapter 3 details our investigation of the discourse used by reporters to describe bugs. Later on, in Chapter 4, we introduce DeMIBuD, an automated technique that detects missing information in bug descriptions, and in Chapter 5, we present Euler, an automated approach that identifies and assesses the quality of the *steps to reproduce* provided in such descriptions. Chapters 6 and 7 specify the proposed query reduction techniques based on bug descriptions to automatically detect duplicate bug reports and locate the reported bugs in source code. Finally, in Chapter 8, we conclude by summarizing our main contributions and discussing directions of future research.

1.5 Bibliographical Notes

Parts of this dissertation were previously published in international conferences or journals, in collaboration with other researches.

Chapters 3 and 4 were published and presented at the 11th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17). The paper published in this conference [88] is the result of collaboration with Jing Lu and Dr. Vincent Ng from The University of Texas at Dallas; Fiorella Zampetti and Dr. Massimiliano Di Penta from the University of Sannio; Dr. Laura Moreno from Colorado State University; and Dr. Gabriele Bavota from Università della Svizzera italiana.

Chapter 5 was accepted for publication at the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19). This work [80] is a collaborative effort with Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk from the College of William and Mary; Jing Lu and Dr. Vincent Ng from The University of Texas at Dallas; and Dr. Massimiliano Di Penta from the University of Sannio.

Chapter 6 was published at the Empirical Software Engineering journal [85]. This publication is an extended version of the paper [83] published and presented at the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17). These manuscripts were published in collaboration with Juan Manuel Florez from The University of Texas at Dallas.

Chapter 7 was published and presented at the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19). The paper published in this conference [86] is a collaborative effort with Juan Manuel Florez and Unnati Singh from The University of Texas at Dallas.

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Bug Reporting

When using software, users and developers often find unexpected system behavior in the form of crashes, errors, functional misbehavior, performance issues, *etc.* [237]. In these cases, users have the option to report the problems to developers using issue or bug tracking systems (*a.k.a.* issue/bug trackers). Specifically, users report the bugs[1] in bug reports, which are documents that contain bug-related information, required by developers to triage, locate, and fix the reported bugs in the code.

Issue/bug trackers, such as Bugzilla [16], Jira [19], GitHub's Issues [28], Launchpad's bug tracker [36], and Trac [52], are commonly used in software development projects, including open source projects [250]. Most issue trackers allow the users to report software bugs via (web) forms that ask for relevant information about the problem, either textually or graphically. These forms may be embedded in the software itself, rather than being implemented in the issue trackers. This is the case of systems such as Microsoft Word [37] or Google Chrome [34]. Products, such as AppSee [25], TestFairy [46], Instabug [35] and BugClipper [26], offer features for collecting bug information graphically, for example, via screenshots or videos.

Traditionally, issue trackers capture different bug-related information, including: (i) bug metadata (*e.g.*, bug severity, system component and version affected by the bug, bug priority, *etc.*), often reported by using a list of predefined options/values given in the issue tracker; (ii) natural language (NL) descriptions of the bugs (*a.k.a.* bug descriptions), written by reporters in free text form [237]; and (iii) attachments that give additional details of the

[1]Users often report bugs related to functional misbehavior, since these cannot be detected and reported automatically (unlike crashes or errors).

bugs (*e.g.*, screenshots, program logs, *etc.*). Depending on the issue tracker and software project, all this information may be collected in a bug report [249].

Bug descriptions are probably the most important part in bug reports [249, 98, 198]. Most of issue trackers, even those that offer graphical bug reporting, allow the user to provide bug information in textual form. Some issue trackers (*e.g.*, GitHub's issue tracker for the TensorFlow project [45]) provide semi-structured reporting capabilities when asking for bug descriptions. In these cases, within the corresponding text field, issue trackers use predefined text templates that explicitly ask for different information (*e.g.*, the steps to reproduce the bug, the problem description, *etc.*). Some software projects (*e.g.*, Mozilla Firefox [10]) prefer to collect such information using separate text fields.

Our research focuses on unstructured bug descriptions provided by reporters. Bug descriptions are expected to include the unexpected software behavior, observed by the user (*i.e.*, the Observed Behavior or OB), the sequence of steps followed by the user to reproduce the (mis)behavior (*i.e.*, Steps to Reproduce or S2Rs), and the normal software behavior (*i.e.*, the Expected Behavior or EB). These three types of information in bug descriptions are essential for developers to triage, locate, and fix the bugs in the software [249, 139]; hence, these are the main focus of our research.

2.2 Bug Report Quality Assessment

Little research has been conducted to assess and improve the quality of bug reports. Most of the research in this area has focused on measuring properties such as readability, coherence, and presence of important information in bug reports.

Zimmerman *et al.* [249] proposed an approach to predict the quality level of bug reports (*i.e.*, good, neutral, or bad), based on the presence of different kinds of information in the reports (*e.g.*, attachments, keywords, or test cases) and based on readability metrics. Dit *et al.* [104] measured the semantic coherence in bug report discussions, by relying on the average textual similarity between each pair of adjoining comments in a bug report. Linstead *et al.* [148] measured such coherence by determining the number of discourse threads (*i.e.*, topics) in the discussions. In another work, Hooimeijer *et al.* [123] measured quality properties of bug reports (*e.g.*, readability) to predict when a bug report would be triaged. Zanetti *et al.* [235] identified valid bug reports, as opposed to duplicate, invalid, or incomplete reports, by relying on reporters' collaboration information. The main disadvantage of these approaches is the lack of specific and actionable feedback given to reporters about bug report (low) quality.

To enhance bug reports, Moran *et al.* focused on augmenting the S2Rs in bug reports via screenshots and graphical component images [164], and on automatically finding, and reporting potential crashes in mobile applications [163]. Zhang *et al.* [238] enriched new bug reports with textually similar sentences from past reports.

As this body of research, our goal is to improve bug reports' quality. Our focus, however, is improving the quality of bug descriptions by automatically (1) determining when the *expected behavior* and the *steps to reproduce* are missing in bug descriptions (see Chapter 4), and (2) detecting problems in the *steps to reproduce* in the descriptions, when provided by reporters (see Chapter 5). We aim to provide actionable feedback to reporters about problems in their bug descriptions, so that they provide higher-quality bug information when writing and submitting bug reports.

2.3 Bug Report Characterization and Analysis

Bug reports/descriptions have been characterized from different angles and for different purposes. Prior research focused on (1) understanding the information provided in bug reports, and on (2) analyzing linguistic properties of bug reports.

Existing research focused on determining the structure of bug reports and its importance in bug triaging and fixing [98, 250, 198, 196, 249, 139]. The main content of a bug report is unstructured, that is, expressed in natural language [249, 98, 198], which includes the description of the system's *observed behavior*, the *expected behavior*, and the *steps to reproduce.* Prior studies found that this information is highly important for developers when triaging and fixing bugs [249, 139].

Other research analyzed bug reports to identify unwanted behavior types [91], defect types [213, 77], stakeholders' information needs [73], and software design decisions [135]. Other work investigated which bugs get fixed [115], and user roles in bug reporting [134, 190].

The work by Ko *et al.* [136] on linguistic analysis of bug report titles aims at understanding how users describe software problems. Sureka *et al.* [211] analyzed the part-of-speech and distribution of words in titles to find vocabulary patterns for predicting bug severity. Our previous research [82], on duplicate bug report analysis, found low vocabulary agreement between bug descriptions. In another direction, some research has focused on summarizing bug reports, including bug descriptions and user-developer conversations [186, 150, 155].

2.4 Duplicate Bug Report Detection

Software systems with large user bases constantly receive many bug reports, especially after major releases, as new bugs are encountered by the users [59, 239, 97, 79, 237]. For example, Eclipse [4], a popular open-source development environment, received more than 45,000 issue reports during 2008 alone [209].

Before the reported bugs are confirmed and assigned to developers for fixing, someone usually checks if these bugs were reported before (*i.e.*, the new reports duplicate previous reports) [97]. When the duplicates are found, the bug reports are marked accordingly, thus avoiding potentially unnecessary work or complementing the information of their duplicates for bug fixing [67]. When the number of bug reports is large, finding duplicates can be a time-consuming and error-prone activity. For example, among the bugs reported for Eclipse in 2008, more than 3,000 were marked as duplicates [209]. Hence, researchers have proposed automated tool support that is meant to save time and improve the accuracy of duplicate bug report detection. For simplicity, for the remainder of this section, we will use *duplicate detection* (or just *DD*) for "duplicate bug report detection".

As proposed by Lin [147] and reused by Kang [130], duplicate detection approaches can be classified into three categories:

- *Ranking* approaches [57, 70, 72, 120, 144, 147, 149, 168, 177, 183, 184, 193, 210, 209, 214, 223, 211, 244, 119], which use a bug report as input (*i.e.*, query) and output a ranked list of duplicate candidates from a corpus of existing bug reports;
- *Binary* approaches [216], which take a bug report as input and label it as either duplicate or not duplicate; and
- *Decision-making* approaches [53, 133, 140, 119], which consider a pair of bug reports and determine whether they are duplicates of each other.

We focus our discussion on ranking approaches since they are the most common techniques. Ranking approaches rely on Text Retrieval (TR) to find duplicate bug reports. Text-Retrieval-based Bug Duplicate Detection (TRDD) approaches model duplicate detection as a document retrieval problem, where the full textual description of a bug report is used as query to search the space built from past bug reports, and retrieve a list of reports relevant to the query. The relevance between the query (a new bug report) and a past bug report is determined by the textual similarity between them: the higher the textual similarity, the more likely the past bug report is to describe the newly-reported bug.

All TRDD approaches support the following process [60]:

1. **Corpus creation.** Each past bug report represents a document. The textual content of each document (*i.e.*, the title/summary and/or description) is normalized using common preprocessing techniques: stop word removal, stemming, *etc.*

2. **Corpus indexing.** The documents from the bug report corpus are transformed into a mathematical representation (*e.g.*, a term-by-document matrix) that is utilized for fast document retrieval and relevance computation.

3. **Query formulation.** The triager/developer formulates a query from the new bug report. Existing TRDD approaches automatically use the full content (*i.e.*, the title/-summary and description) of bug reports as query. The text normalization applied to the corpus is also applied to the query.

4. **Document ranking.** A similarity measure is computed between each document in the corpus and the query, based on the used TR model. Additional information can be used by different techniques to improve the ranking (see below). Then, the documents are sorted according to their similarity scores and presented to the triager. The relevance of a document to a query is given by its similarity, and indirectly, by its rank: the closer a document is to the top of the list, the more relevant it is to the query and, therefore, the more relevant it is deemed to be a report duplicated by the new report.

5. **Results inspection.** The triager/developer examines the top n documents in the list of results, to decide whether or not the new report describes the same bug in any of the inspected documents. If a report is found to describe the same bug, the triager/developer marks the new bug report as duplicate in the issue tracker, and the duplicate detection process ends.

6. **Query reformulation.** If the query does not retrieve any duplicate bug report in the top n documents of the list, then the developer reformulates (or refines) the query, either manually or automatically (by utilizing a query reformulation method). The process continues at step 4.

Researchers have studied the value of using extra information from bug reports to improve the initial document ranking (*i.e.*, step 4). For example, stack traces provided in bug reports are used to boost the textual similarity between bug reports [223, 144]. The bug report creation date is used to retrieve recently reported bugs (*e.g.*, those reported in the past 2 months) [193, 144], or the bugs reported in the same time frame [177]. The reports' metadata, including the system component and version affected by the bug, or the bug priority, is also used to narrow down the search space and give a combined similarity measure between bug reports [53, 70, 119, 133, 140, 149, 183, 184, 193, 209, 216]. Boisselle *et al.* [70] group categorical fields (*i.e.*, bug report metadata) that are unique to two issue trackers into an additional textual field. Domain information, extracted from textbooks, project documentation, or Wikipedia, has also been leveraged, mainly in machine learning approaches [53, 119, 149]. Recently, Wang *et al.* used screenshots provided in crowd-testing testing reports to detect duplicates [219]. Other research efforts have adapted or combined classic TR models. For example, Sun *et al.* [209] incorporate document normalization to the BM25F model [236], and Nguyen *et al.* [168] combines topic modeling (based on LDA [69]) and BM25F. In other work, Rocha *et al.* [189] utilize the VSM algorithm [197] to retrieve bug reports that require changing similar source code files during bug fixing.

2.5 Text-Retrieval-based Bug Localization

Text Retrieval (TR) has been widely used by researchers to support developers during bug localization in source code. TR-based Bug Localization (TRBL) approaches formulate bug localization as a document retrieval problem. A bug report is used as query to search a document space built from source code artifacts of a software system and retrieve a list of code documents (*e.g.*, files, classes, functions, or methods) relevant to the query. The relevance of a source code document to a query is determined by the textual similarity between them: the higher the textual similarity, the more likely the document is to contain the bug described in the bug report. The targeted code documents are expected to be retrieved in the top of the result list.

All TRBL approaches support the following process [60, 105], similarly to TRDD – see Section 2.4:

1. **Corpus creation.** The source code is divided into documents, each one representing a unique code artifact in a software system, according to a granularity level (*e.g.*, file, class, or method). Besides stop word removal and stemming, identifier splitting is often utilized to normalize the code corpus vocabulary.

2. **Corpus indexing.** The documents are indexed for fast document retrieval and relevance computation, using a mathematical representation, such as a term-by-document matrix based on the term frequency/inverse document frequency statistic.

3. **Query formulation.** The developer formulates a query from a bug description. Most existing TRBL approaches automatically use the full content (*i.e.*, the title/summary and description) of a bug report as query. The query is also normalized.

4. **Document ranking.** The query is executed using a TRBL approach, which returns a list of code documents. The documents are sorted according to their similarity scores with the query and are presented to the developer. Additional information to the bug report is used by different techniques to improve the ranking (see below). The closer a software document is to the top of the list (*i.e.*, the highest the similarity score), the more likely it is to contain the reported bug.

5. **Results inspection.** The developer examines the top n software documents in the list of results, to decide whether or not the documents require changes for removing the bug. The developer performs this step by inspecting each document's name as well as its internal code. Deciding whether each code document is buggy or not, with respect

to the bug report, largely depends on the developer's knowledge of the system [158]. If a buggy code document is found, then the bug localization process ends.

6. **Query reformulation.** If the query does not retrieve the buggy code artifacts in the top n documents of the list, then the developer reformulates the query and the process continues at step 4.

TRBL is an instance of TR-based concept/feature location in source code [157, 105] and TR-based traceability link recovery [100]. What differentiates TR-based bug localization from the more general code retrieval approaches is the use of bug reports as queries.

Previous research in concept/feature location and traceability link recovery focused on improving all the six steps of this process and TRBL techniques utilize much of that research. The main research efforts in TRBL focused primarily on step 4. While some research has focused on improving/optimizing traditional TRBL techniques, for example, via parameter tuning, advanced machine learning, or extending the mathematical models behind them [241, 231, 141, 107, 121, 226, 230, 232], most of the research has focused on leveraging additional information related to the bug reports, to adjust the (initial) ranking of code documents.

Additional information leveraged by existing TRBL techniques includes: code structure [220, 195, 221, 233, 55, 212, 240], part-of-speech tags [246], similar bug reports [245, 220, 221, 233, 195, 99, 225, 187], code version history [204, 221, 233, 220], stack traces [167, 225, 221, 233, 203, 224], or combinations of the above [221, 233, 220, 195, 225, 201, 96].

Software history information is used by TRBL approaches to boost code artifacts with high defect/change probability based on code change records (*e.g.*, version control records). The code artifacts boosted are those found in change-sets that were intended to fix bugs. The boost amount can depend on different factors, for example, the number of times a code artifact has been fixed [204, 221, 224, 233] or how long ago this happened [204, 220, 224].

Bug fix history is also used to complement textual similarity. A set of previously fixed bug reports is kept, each one with its corresponding fix-set: the set of code documents that were modified in order to fix the bug. A query (*i.e.*, the current bug report) is compared to each previously-fixed bug report. The documents in each fix-set are boosted according to some criteria, for example, the textual similarity of the fixed bug with the query [245, 220, 221, 233, 195, 99, 225]. Recently, feature requests have been leveraged in addition to bugs, in the same way described before [187].

Bug reports sometimes contain stack traces, which are also used to alter the text-based ranking. Some TRBL approaches work on the assumption that the buggy code artifacts could be directly referenced by these traces, and use regular expressions to identify referenced classes/files [167, 225, 221, 233, 203]. The set of suspicious classes/files is expanded by identifying artifacts (in)directly referenced in the code of the ones found in the stack trace. These relationships can be found by using the system's call graph [167] or the files' import statements [225, 233].

Some approaches exploit query and document structure by simply splitting the query into two parts (bug report title and description) and the document in four parts (classes, methods, variables, and comments). Besides the score calculated from the full text of both the query and the document, additional scores are calculated from the similarities between each of the two query components and each document component (eight additional scores in total), and then all scores are added together. This assigns a greater weight to terms appearing in multiple fields of a document, increasing their discriminating power for retrieval [220, 195, 221, 233, 55].

It has also been proposed to use code smells as a separate source of information [212]. For this approach, code smells are detected along with their severity, and this severity score is combined with the textual similarity while ranking code elements. Finally, part-of-speech information has been explored as a possible source of improvement. Zhou *et al.* [246] propose boosting the retrieval weight of bug report terms tagged as nouns by an automatic part-of-speech tagger.

2.6 Query Reformulation in Software Engineering

Query reformulation has been studied mostly in the context of code retrieval, which includes TRBL, TR-based concept/feature location, traceability link recovery, and code search. Most of the proposed methods and findings are also utilized by TBDD techniques.

Existing research highlighted the challenges that developers face when re-formulating queries for code retrieval [208, 63, 95]. On one hand, TRBL approaches mitigate the problems associated with formulating an initial query by utilizing the entire bug report [90]. On the other hand, prior research provides little or no guidance on what parts of the bug reports to use for reformulating a query when no relevant results are retrieved within the first few entries of the result list [132].

Three query reformulation strategies are well-known, namely, *query expansion* [76], *query replacement* [112, 114], and *query reduction* [151, 181, 132]. Query expansion consists in adding alternative terms (or phrases) to a query; query replacement changes (part of) a query with a new set of terms; and query reduction focuses on removing query terms.

Most existing research on query reformulation in code retrieval (including TRBL) has focused on *query expansion*. The methods to determine the alternative terms include: relevance feedback from developers [110]; pseudo-relevance feedback [116, 205], which leverages the lexicon of the previous top code documents retrieved; the use of English or software ontologies such as WordNet or custom-built models [200, 180], which contain related terms to the ones in a query (*e.g.*, synonyms); and co-occurring term information from various software sources, such as source code, Stack Overflow (SO) questions, or regulatory documents [179, 158, 102]. Similar techniques have been applied in the context of code search, where the initial queries are reformulated based on thesauri such as lexical databases from SO [146, 143, 111], relevance feedback from users [222], pseudo-relevance feedback from SO results [169], co-occurrence and frequency of query terms with previous results and source code [118, 191], and textual similarity between the query and Application Programming Interfaces [152].

Query replacement has been utilized mostly for traceability link recovery [112], where the terms from similar web and domain-specific documents to the query are leveraged to select a set of candidate terms to replace the initial query. Another query replacement method is learning frequent terms from existing requirement-regulation trace corpora, and using them as the new query [114].

Regarding *query reduction*, our prior research showed that removing noisy terms from the query (*i.e.*, from bug reports) leads to substantial retrieval improvement in TR-based bug localization [90]. Similarly, Mills *et al.* [162] found that near-optimal (reduced) queries from bug report lead to high improvement on code retrieval. The (few) works that include some kind of query reduction rely on heuristics to remove the noisy terms. Specifically, Rahman *et al.* [179] discarded the terms different from nouns or those occurring in more than 25% of the code documents, since they are likely to be non-discriminating. Haiduc *et al.* [116] followed a similar strategy. Kevic *et al.* [132] recommended the top three terms in a change request that have the highest predictive power to retrieve the relevant code documents (*i.e.*, in top-10 of the list). Their findings suggest that terms that appear in both the summary and description of change requests are good candidates to be used as query [132]. In another work, Rahman *et al.* [181] leveraged term co-occurrences and syntactic dependencies to select the most important terms in a change request as a query. Recently, the same authors proposed weighting and selecting query terms based on how these relate to each other and whether they reference code entities and/or appear in particular parts of the bug reports, for example, stack traces [182]. Related to term selection, other research focused on weighing terms (from the query) that occur in method names and calls [65] or terms corresponding to source code file names [103].

CHAPTER 3

THE DISCOURSE USED IN BUG DESCRIPTIONS[1]

3.1 Introduction

We aim to understand the discourse used by reporters to describe the observed behavior (OB), expected behavior (EB), and steps to reproduce (S2Rs) in bug descriptions. Our hypothesis is that *reporters use a limited vocabulary and a well-defined set of discourse patterns when describing the OB, EB, and S2Rs*. Verifying this hypothesis means that we can automatically detect the presence or absence of the OB, EB, and S2Rs in bug descriptions with high accuracy, and thus leverage such information to improve bug resolution tasks that rely on these descriptions. The converse situation would mean that the automatic analysis of unstructured bug descriptions would be impractical.

To the best of our knowledge, no existing research validates or invalidates our hypothesis. The work on bug description analysis has mainly focused on investigating linguistic properties of bug report titles [136], identifying frequently asked questions [73], investigating unwanted behavior types [91], and studying the structure of bug reports [98, 198, 249]. Other work has focused on identifying linguistic patterns in other software engineering sources, such as development e-mails [101] or app reviews [171]. Little is known about the discourse that reporters use to describe software bugs.

To validate our hypothesis, we conducted an empirical study based on open coding [160] and qualitative discourse analysis [173, 54] on a set of 2,912 bug reports from nine software projects. The goal of our study is to understand how essential bug-related information is expressed in bug descriptions. To this end, we identified the *discourse patterns* that reporters

[1]© 2017 ACM. Reprinted with permission from Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng, "Detecting missing information in bug descriptions", Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17).

use to describe the OB, EB, and S2Rs in bug descriptions. *Discourse patterns* are rules that capture the syntax and semantics of the text. Figures 3.1, 3.2, and 3.3 are examples of OB, EB, and S2R discourse patterns, respectively.

3.2 Research Questions and Context

Our study aims at answering the following research questions:

> $\boldsymbol{RQ_1}$: *To what extent bug descriptions contain the OB, EB, and S2Rs?*
>
> $\boldsymbol{RQ_2}$: *Do bug reporters describe the OB, EB, and S2Rs in bug descriptions using a well-defined set of discourse patterns?*

$\boldsymbol{RQ_1}$ investigates if reporters tend to provide the OB, EB, and S2Rs, and motivates the need for automated support to improve bug descriptions. $\boldsymbol{RQ_2}$ aims at understanding *the discourse* followed by reporters to describe the OB, EB, and S2Rs in bug descriptions.

The context of our research is the analysis of *bug reports*, which are issues[2] that describe potential software bugs or defects. We do not analyze issues describing *feature requests*, *enhancements*, *questions*, or *tasks.* In addition, our analysis task focuses on *bug descriptions.* However, we do not analyze the title of bug reports, which are part of the entire bug description. This means that we only focus on analyzing the description of the bug reports. The reason for this is that titles rarely describe completely the OB, EB, or S2Rs, for example, they can simply be noun phrases or words referring to the reports' topics [136]. We expect to find a set of *discourse patterns* for the sentences and paragraphs (*i.e.*, the *units of discourse*) of the bug descriptions. A *discourse pattern* is a rule that structures a sentence or a paragraph to convey either the OB, EB, or S2Rs. This means that a pattern captures the syntax and semantics of sentences and paragraphs.

[2]Issues are documents collected in issue trackers. Issues describe tasks or change requests intended to improve a software system [250, 237]. Bug reports are a type of issues.

3.3 Bug Report Sampling

We collected a sample set of issues from nine software projects of different types and domains. These projects rely on different issue/bug trackers to capture potential software bugs found by the users. We briefly describe each one of the selected projects. *Docker* [3] is a set of command line tools to automate the deployment of applications inside software containers; *Eclipse* [4] is an IDE for multiple programming languages; *Facebook* [5] is a social networking web platform; *Firefox* [9] is the Mozilla web browser; *Hibernate* [7] is an Object-Relational Mapping framework for Java; *Httpd* [2] is the Apache web server; *LibreOffice* [8] is a desktop office suite; *OpenMRS* [11] is a web-based health care system; and *Wordpress-Android* [13] is the Android mobile application of Wordpress (a content management system). *Eclipse*, *Firefox*, *Httpd*, and *LibreOffice* use Bugzilla as issue tracker [16]; *Hibernate* and *OpenMRS* use Jira [19]; *Docker* and *Wordpress-Android* (*a.k.a.* Wordpress-A) use GitHub's Issues [28]; and *Facebook* uses a proprietary issue tracking system. These projects, except for Facebook, are open source.

To create our bug report sample set, we relied on the data set collected by Davies *et al.* [98] for Eclipse, Facebook, Firefox, and Httpd. This data set is composed of 1.6k issues randomly sampled from their corresponding issue trackers. From this data set and the online issue repositories of the remaining projects, we performed random sampling, making sure to exclude issues that were not bug reports (*e.g.*, feature requests) by manually inspecting the type of issue and its comments. In total, we collected 2,912 bug reports: 324 reports per project on average, including the ones collected by Davies *et al.* [98]. From these, we used 1,091 reports for discourse pattern discovery and the remaining ones (*i.e.*, 1,821) for validation purposes. We refer to the former data set as the *discourse bug reports* and to the latter as the *validation bug reports*.

3.4 Coding Procedure

While we coded the presence of the OB, EB, and S2Rs in the *discourse bug reports* and *validation bug reports*, we only used the *discourse bug reports* to infer the discourse patterns.

Discourse Pattern Coding. Five coders (four PhD students and one professor) conducted the sentence and paragraph coding task for the *discourse bug reports*. In order to define a starting coding framework, one of the coders conducted a pilot study on 25 bug reports from Davies *et al.*'s issues [98]. These were not used in the *discourse bug reports*. The goal of this task was to analyze the bug descriptions, identify the sentences or paragraphs that corresponded to the OB, EB, and S2Rs, and infer the discourse patterns from them. This task resulted in twelve preliminary discourse patterns with specific textual examples from the issues, a set of textual characteristics of the bug descriptions, and the initial coding criteria. Once the pilot study was completed, this person trained the rest of the coders in a 45-minute session that involved discussing the results and some ambiguous sentences.

To facilitate the coding process, we built a tool that automatically splits the bug description text into sentences and paragraphs. The tool relies on the sentence parser from the Stanford CoreNLP toolkit [156] and a set of heuristics, such as detecting punctuation for sentences and two or more line breaks for paragraphs.

The 1,091 *discourse bug reports* were evenly and randomly distributed among coders, to ensure that each coder received a subset of reports from each of the nine projects. Each person coded 218 reports except for one person who coded 219 (*i.e.*, 25 reports per system per person, on average). For each bug report, the coders analyzed the bug description and marked each sentence or paragraph as OB, EB, or S2R. A sentence/paragraph can fall into more than one of these categories at the same time. Then, the coders inferred a discourse pattern from each marked sentence/paragraph and assigned a *code* to it. A *code* is a label that uniquely identifies a discourse pattern. Note that it is possible to infer more than one pattern from a sentence/paragraph. A catalog of inferred patterns was shared among coders via an online spreadsheet. In this way, all coders were aware of the patterns inferred by each coder and were able to reuse existing patterns or add new ones to the catalog. When one of the coders identified a new pattern, it was included in the catalog and the other coders were notified. Each new pattern was verified by all the coders and disagreements were solved via open discussion. For each new pattern, the existing catalog was inspected for similar patterns and, when appropriate, with unanimous agreement, similar patterns were merged into a new one (*i.e.*, a more general pattern) and the existing labels were updated accordingly. This process was fully iterative, and included constant refinement of the pattern catalog as well as discussion of ambiguous cases. Every decision taken during the pattern extraction was representative of the opinion of all coders.

To minimize subjectivity, we recruited four additional coders (one CS master student, two developers, and one business analyst) and asked them to code the same 1,091 reports coded by the first group of coders. In a 40-minute session, one member of the first group trained the new coders on the coding procedure and criteria (see Section 3.5). We randomly distributed the reports among the new coders ensuring that each one coded a subset of bug descriptions coded by each of the original coders. The task of the additional coders was to mark the sentences and paragraphs that corresponded to any OB, EB, or S2Rs. This time, the pattern inference was not part of the task, as the iterative and collaborative nature of the pattern coding procedure already aimed at minimizing subjectivity. In the end, each report from the *discourse bug reports* was coded by two distinct coders. The inter-coder agreement is discussed in Section 3.6.

Validation Set coding. As for the *validation bug reports* (*i.e.*, remaining 1,821 bug reports from our initial sample), all nine coders were requested to follow the same coding process, without pattern inference. Each report was coded by two different coders. On average, 202 reports were assigned to each pair of coders. The bug reports were randomly distributed so that each pair of coders received a subset of reports from each system. Again, the coders marked the sentences and paragraphs that corresponded any OB, EB, and S2Rs (*i.e.*, no pattern inference this time).

3.5 Coding Criteria

We summarize the most important criteria followed by the coders (the full list can be found in the replication package [89]). The coders were provided with examples of each criterion.

The coding focused only on natural language (NL) content written by the reporters, as opposed to code snippets, stack traces, or logs. However, the NL referencing this information was coded. In addition, only explicit mentions of OB/EB/S2Rs were labeled. Note that it is possible to infer EB from OB descriptions, as the former is usually the opposite of the latter. Such cases were not labeled.

Regarding the OB, uninformative sentences such as *"The system does not work"* are insufficient to be considered an OB. There must be a clear description of the observed (mis)behavior of the software. Code explanations and root causes are not considered OB. Regarding the EB, solutions or recommendations to solve the bugs are not considered EB. In some cases, imperative sentences such as *"Make Targets not automatically filled..."* may be considered EB according to the context of the reported bug. Sometimes, however, these suggest tasks instead of EB. Regarding the S2Rs, one or more sentences (*i.e.*, a sentence or a paragraph) can describe the steps to reproduce the bug. Conditional sentences such as *"when I click on apache.exe it returns an error like this"* may describe S2Rs, if they provide enough details about how to reproduce the bug. Finally, S2R paragraphs may also contain OB and EB sentences.

3.6 Inter-coder Agreement

We analyzed the reliability of the coding process regarding the presence and absence of OB, EB, and S2Rs in bug descriptions. As mentioned before, each bug description was coded by two coders. We measured the observed agreement between coders as well as Cohen's Kappa (k) [93] and Krippendorff's alpha (α) [137] coefficients. Our analysis reveals high inter-coder agreement levels. Coders agreed on: the presence or absence of OB in 91% of the cases (avg. $k = 37.3\%$, $\alpha = 40.4\%$, *i.e.*, fair agreement [218]); the presence or absence of EB in 85.5% of the cases (avg. $k = 70.2\%$, $\alpha = 67.7\%$, *i.e.*, substantial agreement [218]); and the presence or absence of S2Rs in 76% of the cases (avg. $k = 49.2\%$, $\alpha = 51.9\%$, *i.e.*, moderate agreement [218]).

Overall, 1,131 bug reports (*i.e.*, 38.9%) had some type of disagreement. We solved the disagreements by applying a third person scheme. We distributed the conflicting reports among the nine coders in such a way that a third coder (different from the original two coders) would judge and solve the disagreements. Our analysis revealed that the main causes for disagreement were omissions, mistakes, and, in the case of S2Rs, misunderstandings, as in several cases it was not clear if (single) conditional sentences were specific enough to be considered S2Rs (*e.g.*, "*When I started the project, it crashed*").

3.7 Bug Description Information

To answer $\boldsymbol{RQ_1}$, we computed the proportions of bug reports describing any OB, EB, and S2Rs. Table 3.1 reveals that, while most of the bug reports contain OB (*i.e.*, 93.5%), only 35.2% and 51.4% of the reports explicitly describe EB and S2Rs, respectively. 22.1% of the reports contain all three types of information (*i.e.*, OB, EB, and S2Rs). These results indicate that essential information is often missing in bug reports and motivate the need for the automated detection of such information. Firefox is the system with the highest number of reports having EB and S2Rs (*i.e.*, 67.4% and 76.4%, respectively) and having all three types of information. We attribute this result to the use of predefined templates and distinct text fields in the issue tracker that explicitly asking for this information. Wordpress-Android has the lowest number of reports with OB. We observed that images, rather than textual descriptions, are commonly used in this project.

Table 3.1: Number of bug reports containing OB, EB, and S2Rs.

Project	**#OB**	**#EB**	**#S2Rs**	**Total**
Docker	314 (93.5%)	113 (33.6%)	207 (61.6%)	**336**
Eclipse	271 (90.9%)	101 (33.9%)	173 (58.1%)	**298**
Facebook	327 (96.7%)	81 (24.0%)	133 (39.3%)	**338**
Firefox	335 (96.5%)	234 (67.4%)	265 (76.4%)	**347**
Hibernate	315 (95.2%)	89 (26.9%)	150 (45.3%)	**331**
Httpd	350 (96.4%)	102 (28.1%)	104 (28.7%)	**363**
LibreOffice	322 (97.3%)	122 (36.9%)	241 (72.8%)	**331**
OpenMRS	275 (93.5%)	94 (32.0%)	104 (35.4%)	**294**
Wordpress-A	215 (78.5%)	88 (32.1%)	121 (44.2%)	**274**
Total	**2,724 (93.5%)**	**1,024 (35.2%)**	**1,498 (51.4%)**	**2,912**

3.8 Bug Descriptions' Discourse

To answer $\boldsymbol{RQ_2}$, we analyzed the discourse patterns inferred by the coders. Our open coding approach resulted in a catalog of 154 patterns that capture the discourse followed by reporters to describe the OB, EB, and S2Rs. Most of the patterns are sentence-level patterns (135) and most of the paragraph-level patterns correspond to S2Rs (13 out of 19). We summarize and discuss our pattern catalog and the discourse used for each type of information in bug descriptions (the full catalog is available in the replication package [89]).

OB discourse. We observe that many patterns in our catalog correspond to OB (*i.e.*, 90 or 58.4% — see Table 3.2). Out of these, 85 are sentence-level patterns and five are paragraph-level patterns. Software (mis)behavior is usually described following a negative discourse. The six most frequent OB patterns correspond to negative textual content and

Table 3.2: Number of patterns used to express the OB, EB, and S2Rs.

Project	**# of bug reports**	**# of patterns**			
		Overall	**OB**	**EB**	**S2Rs**
Docker	113	88	64	11	13
Eclipse	148	95	64	12	19
Facebook	153	102	67	16	19
Firefox	132	97	62	17	18
Hibernate	103	86	64	10	12
Httpd	133	100	72	13	15
LibreOffice	120	88	59	10	19
OpenMRS	92	70	48	11	11
Wordpress-A	91	69	44	11	14
Overall	**1,085**	**154**	**90**	**31**	**33**

account for 68.9% of the *discourse bug reports* that contain OB. Three of these patterns are: NEG_AUX_VERB, VERB_ERROR, and NEG_VERB. The first one is the most frequent one, which corresponds to negative sentences containing auxiliary verbs (see Figure 3.1). The second one corresponds to sentences with verb phrases containing error-related nouns, such as *"VirtualBOx GUI gives this error:"* (from Docker 1583), and the third one, to sentences with non-auxiliary negative verbs such as *"Writer hangs on opening some doc, docx or rtf files"* (from LibreOffice 55917). We also observed OB positive discourse. For instance, the COND_POS pattern represents conditional sentences with positive predicates, such as *"When the merge was completed, I saw that the entries in the value_coded column remained as they were originally"* (from OpenMRS TRUNK-3905). The BUT pattern corresponds to sentences containing contrasting terms followed by affirmative predicates, such as *"You require at least 7 letters, but our name (Delupe) only consists of 6"* (from Facebook 13084). The top six most frequent positive discourse patterns account for 33% of the reports describing OB. Overall, the top six most frequent negative and the top six positive patterns appear in 82.5% of the OB bug descriptions.

EB discourse. Reporters describe expected system behavior using 31 patterns (*i.e.*, 20.1% of our pattern catalog—see Table 3.2). Most of them (*i.e.*, 30) are sentence-level

Pattern code: S_OB_NEG_AUX_VERB
Description: negative sentence with auxiliary verbs
Rule: ([subject]) [negative aux. verb] [verb] [complement]
Definitions:
[negative aux. verb] ∈ {are not, cannot, does not, did not, *etc.* }
Example: [*The icon*] [***did not***] [*change*] [*to an hourglass...*] (from Eclipse 150)

Figure 3.1: Most common OB discourse pattern.

patterns. The most frequent pattern is SHOULD (see Figure 3.2), which represents sentences using the modal terms *"should"* or *"shall"*. These types of sentences appear in 44.2% of the reports that describe EB. Other frequent discourse for describing EB is represented by the EXP_BEHAVIOR, INSTEAD_OF_EXP_BEHAVIOR, EXPECTED, and WOULD_BE patterns. The former, EXP_BEHAVIOR, represents sentences with explicit EB labels, such as *"Expected Results: Taken away the dialog box..."* (from Firefox 226732); INSTEAD_OF_EXP_BEHAVIOR accounts for sentences using *"instead of"* (or similar terms), such as *"When you try to schedule a saved draft, it is published immediately instead of being scheduled for the future date you select"* (from Wordpress-Android 3913); the EXPECTED pattern represents sentences using noun phrases or conjugated verbs of the word *"expect"*, such as *"The expectation was that objects would be loaded identically regardless of using scrollable results or using get result list from JPA."* (from Hibernate HHH-10062); and WOULD_BE corresponds to sentences containing *"would be + positive adjective"* phrases, such as *"It'd be optimal if the UX updated to reflect the actual updated follow state for given users/blogs"* (from Wordpress-Android 447). These five patterns appear in 86.3% of the reports describing EB.

S2R discourse. The *steps to reproduce* discourse is represented by 33 patterns (see Table 3.2), 13 of which are paragraph-level patterns. This means that reporters often use more than one sentence to describe the *steps to reproduce*. While the most frequent pattern to describe S2Rs is paragraphs containing a labeled list of actions (see Figure 3.3—*i.e.*,

Pattern code: S_EB_SHOULD
Description: sentence using the modals "should" or "shall" with no preceding predicates that use negative auxiliary verbs
Rule: [subject] should/shall (not) [complement]
Example: [*Apache*] ***should*** [*make an attempt to print the date in the language requested by the client*] (from Httpd 40431)

Figure 3.2: Most common EB discourse pattern.

it accounts for 30.7% of the reports describing S2R), the S2Rs are also expressed using a single sentence. For example, the COND_OBS pattern corresponds to conditional sentences containing non-negative OB predicates, such as *"When saving a new (transient) entity ..., Hibernate will generate [at least] two INSERT statements..."* (From Hibernate HHH-6630). In addition, the CODE_REF pattern describes sentences with noun phrases and adverbs of location to refer to code, scripts, or other non-natural language information used to reproduce the observed behavior. An example of this type of sentences is: *"The following statement produces a compilation error in JDT..."* (from Eclipse 52363). The top five most frequent S2R discourse patterns are present in 77.2% of the S2R bug descriptions.

Unique discourse patterns. We found overlap among OB, EB, and S2R patterns. Either some patterns are equivalent or are part of others across the three types of information. Specifically, we found that the INSTEAD_OF OB pattern is equivalent to the INSTEAD_OF_EXP_BEHAVIOR EB pattern. The COND_POS and COND_NEG OB patterns are part of two S2R paragraph-level patterns and three S2R sentence-level patterns (all related to conditional content). The IMPERATIVE EB pattern is part of two S2R paragraph-level and two S2R sentence-level patterns (all related to imperative content). In the end, 87 OB, 29 EB, and 24 S2R patterns are unique in our catalog (see the replication package [89]).

Summary. Our discourse analysis revealed that reporters use 154 discourse patterns to describe the OB, EB, and S2Rs in bug descriptions; and 82% (on average) of the instances of OB, EB, and S2Rs are described using only 22 (14.3%) of the patterns.

Pattern code: P_SR_LABELED_LIST
Description: paragraph containing a non-empty labeled list of sentences that indicate actions. The label is optional and indicates S2R terms. The "action sentences" may be simple or continuous present/past sentences or imperative sentences. The list may contain OB and EB sentences in no particular order.
Rule: ([S2R label])
    [[number/bullet] [action sentence]]+
    [([number/bullet]) [OB/EB sentence]]*
Definitions:
- [S2R label] ∈ {"how to reproduce", "STR", "To replicate", "Steps to reproduce", ...}
- [number/bullet] ∈ { "1.", "1 -", "-", "*", ... }
- [action sentence] ∈ {[present/past continuous sentence], [simple present/past sentence], [imperative sentence]}

Example: (from Firefox 215939)
[*Steps to Reproduce:*]
[*1.*] [*Start Firebird.*]
[*2.*] [*C-t to open a new tab.*] [*The second tab is now displayed.*]
[*3.*] [*Type 'hello'.*] [*This text appears in the location bar.*]
[*4.*] [*Click on the header for the first tab to switch to that tab.*]
[*5.*] [*Click on the header for the second tab...*]

Figure 3.3: Most common S2R discourse pattern.

3.9 Discourse Analysis across Projects

Table 3.2 shows how many patterns of every kind we identified in any of the sentences/ paragraphs of the bug reports for each project. As seen in the table, not all patterns are used in each system.

Figure 3.4 depicts the pattern distribution across projects, where each bar indicates the number of patterns that appear in a given number of projects. Out of the 154 patterns in our catalog, 21 (*i.e.*, 13.6%) patterns appear in only one project each, 42 (*i.e.*, 27.3%) appear in two to four projects, 67 (*i.e.*, 43.5%) appear in five to eight projects, and 24 patterns (*i.e.*, 15.6%) appear in all nine projects.

Examples of rare patterns that appear in one project only are: IMPOSSIBLE, which is used to describe OB in *"...it's impossible to click the post button..."* (from Facebook

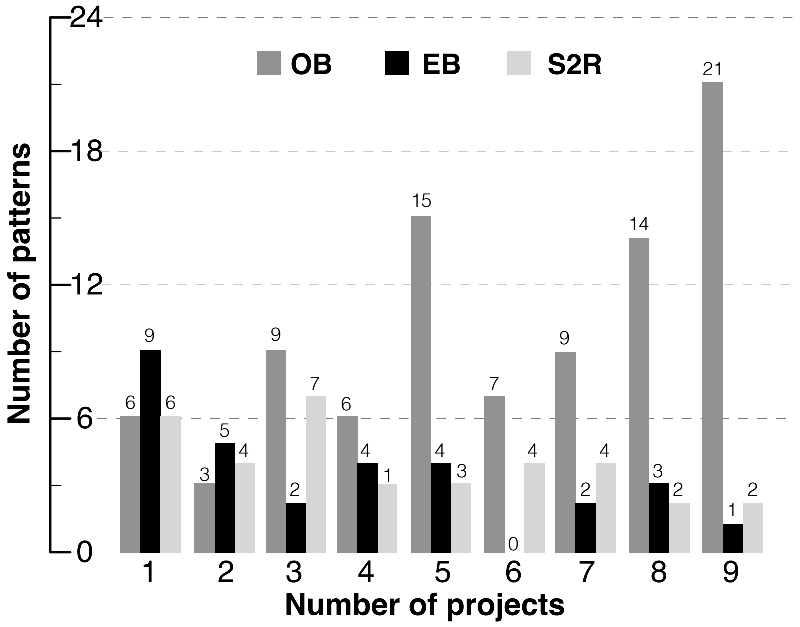


Figure 3.4: Distribution of pattern appearance across projects.

8978); WHY_FIRST_PLACE, which is used to describe EB in *"...why the outgoing changes didn't appear in the first place"* (from Eclipse 41883); and COND_THEN_SEQ, which is used to describe S2Rs in *"If you enter..., then switch..., then switch back... "* (from Firefox 215939). Examples of frequently used patterns which appear in all nine projects are: NEG_AUX_VERB (OB), SHOULD (EB), and COND_OBS (S2R)—these are previously described in Section 3.8.

Figure 3.4 reveals that the distribution of OB patterns differs from the distribution of EB and S2R patterns. Nearly three quarters of the OB patterns (*i.e.*, 66 or 73.3%) appear in more than half of the projects (*i.e.*, in five to nine projects). In contrast, less than a third of the EB patterns (*i.e.*, 10 or 32.3%) and less than half of the S2R patterns (*i.e.*, 15 or 45.5%) appear in more than half of the projects. These results indicate that reporters use many patterns to describe OB and they reuse them often across systems. Reporters use far fewer patterns to describe EB and only a third of them are reused frequently. About half of the S2R patterns are reused frequently across systems.

Our previous analysis identified 22 patterns that are used in 82% of the OB/EB/S2R descriptions, on average. We deepened our analysis to determine how many patterns are used to achieve similar coverage for *each* system in our sample set. Nearly a third of the patterns (*i.e.*, 49 out 154—31.8%) are used to describe OB, EB, or S2R in at least 82% of the cases. Among the 22 patterns, twenty (10 OB, 5 EB, and 5 S2R patterns) are frequently reused in each project.

3.10 Threats to validity

We briefly discuss the threats that affect the validity of our study.

The main threat to *construct validity* is the subjectivity introduced in *discourse patterns* discovery and in the construction of the labeled bug reports (see Section 3.4). To minimize subjectivity, we ensured that each bug report was coded by two coders independently. We assessed coding reliability by measuring the inter-rater agreement (see Section 3.6). Regarding pattern inference, our coding procedure was based on open coding practices [160] that aimed at minimizing subjectivity. The five coders extracted the patterns in a strict, iterative, and open manner [160], which led to continuous discussion of ambiguous cases, refinement of our pattern catalog and coded data, and assessment of our coding process. We also defined coding criteria and trained the coders on them via interactive tutorials.

In order to strengthen the *external validity*, we coded 2,912 bug reports from nine software projects of different domains (*e.g.*, programming or web browsing) and types (*e.g.*, desktop or mobile applications). These projects are open source (except for Facebook) and use different issue trackers. From these bug reports, we used 1,091 reports for discourse pattern discovery.

3.11 Related work

We followed the methodology proposed by Polanyi [173] for the analysis of the linguistic structure of discourse. We built on this analysis to identify discourse patterns based on grounded theory practices [160], particularly, open coding. This technique has been extensively used in Software Engineering (SE) to identify, for example, types of knowledge in Application Programming Interface (API) documents [154], API privacy policy information [206, 68], and information relevant to development activity summaries [217].

Prior research focused on analyzing textual characteristics of bug reports. Ko *et al.* [136] performed a linguistic analysis of bug report titles to understand how users describe software problems. The results showed that distinct types of words, such as verbs or prepositions, play a particular role in conveying the problem described in issues titles, as well as the context where it occurred. With a different focus, Sureka *et al.* [211] analyzed the part-of-speech and distribution of the words contained in issue titles to find vocabulary patterns that would help predict the severity level of bug reports. Moreno *et al.* [166] measured the vocabulary agreement between bug reports and source code and found that 80% of the faulty classes share vocabulary with bug reports and the overall average agreement between these sources is 24%. More recently, our previous research measured the vocabulary agreement between duplicate bug reports and Stack Overflow questions, and found that pairs of duplicate issues and questions share little vocabulary (about 25%, on average) [82]. Chilana *et al.* [91] investigated unwanted (or observed) behaviors described in bug reports to determine different types of user expectations. Different from existing work, our study focused on identifying the OB, EB, and S2R discourse that reporters use in bug descriptions.

3.12 Conclusions

The analysis of 2,912 bug reports from nine software systems revealed that while most of the reports (*i.e.*, 93.5%) describe the OB, only 35.2% and 51.4% of them explicitly describe the EB and S2Rs. These findings motivate our effort in developing automated techniques to detect missing OB, EB, and S2R content and improve bug descriptions. In addition, from the discourse analysis of a subset of 1,091 bug report descriptions, we found that reporters recurrently use 154 discourse patterns to describe the OB, EB, and S2Rs, and few of them (*i.e.*, 22 or 14.3%) appear in most of the bug reports that contain such information (*i.e.*, 82% on average). These results indicate that OB, EB, and S2R content can be automatically detected with high accuracy.

As part of future work, we will conduct additional analyses of the inferred patterns. For example, we will analyze the distribution of patterns across the types of bug described in bug reports [213, 77]. The hypothesis is that different types of failures in bug reports are described with particular types of discourse (represented by the patterns). To validate our hypothesis, we will identify the type of failures in our set of 1,091 bug reports, and analyze the frequency of the patterns across bug types and software projects. We expect that the results of this analysis will further characterize the discourse used by reporters to describe the OB, EB, and S2Rs in bug descriptions.

3.13 Acknowledgments

The chapter was published and presented at the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17). This publication [88] is a collaborative effort with Jing Lu and Dr. Vincent Ng from The University of Texas at Dallas; Fiorella Zampetti and Dr. Massimiliano Di Penta from the University of Sannio; Dr. Laura Moreno from Colorado State University; and Dr. Gabriele Bavota from Università della Svizzera italiana.

Oscar Chaparro, Jing Lu, Fiorella Zampetti, and Laura Moreno contributed to the qualitative coding and analysis of bug descriptions. Gabriele Bavota contributed to the quantitative analysis of the coded data. Massimiliano Di Penta contributed to the design of the empirical study and the quantitative analysis of data. Vincent Ng provided valuable feedback to conduct the empirical study.

CHAPTER 4

DETECTING MISSING INFORMATION IN BUG DESCRIPTIONS[1]

4.1 Introduction

The study presented in Chapter 3 revealed that reporters recurrently use 154 discourse patterns to describe the OB, EB, and S2Rs in bug descriptions, which means that such information can be automatically detected with high accuracy. Inspired by these findings, we developed an automated approach (called DeMIBuD – **De**tecting **M**issing **I**nformation in **Bu**g **D**escriptions) for detecting missing EB and S2Rs in bug descriptions. We focused on EB and S2Rs because, as discussed in Chapter 3, these are more likely to be missing in the bug descriptions. DeMIBuD can be used either to alert submitters while writing and submitting bug reports or as a quality assessment tool for triagers, so that they can contact the reporters right away to solicit the missing information, while the facts are still fresh in memory. DeMIBuD can also be used to augment existing bug report quality models [249, 123].

Very little research has been done on detecting the presence/absence of the OB, EB, or S2Rs in bug descriptions. Most of the approaches proposed in the literature are meant to detect other types of information, such as source code snippets or stack traces [61, 174, 188, 230, 66]. The few approaches that detect the OB, EB, or S2Rs [249, 66, 98] rely on keyword matching, such as *"observed behavior"* to detect OB, or basic heuristics, such as enumerations/itemizations identification to detect S2Rs. Unfortunately, while simple and straightforward, these approaches are suboptimal in accurately detecting such content, as they lead to excessive number of cases undetected (*i.e.*, false negatives) [98].

[1]© 2017 ACM. Reprinted with permission from Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng, "Detecting missing information in bug descriptions", Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17).

We designed and evaluated three versions of DeMIBuD that automatically detect missing EB and S2Rs in bug descriptions: DeMIBuD-R, based on regular expressions; DeMIBuD-H, based on heuristics and Natural Language Processing (NLP); and DeMIBuD-ML, based on Machine Learning (ML). The first two approaches are unsupervised, while the third one requires the use of a labeled set of bug reports explicitly reporting the ones containing and not containing EB and S2Rs.

4.2 Regular Expressions-based DeMIBuD

DeMIBuD-R uses regular expressions to detect if a bug report contains (or does not contain) the EB and S2Rs. The regular expressions rely on frequently used words found in our EB and S2R discourse patterns, for example, keywords explicitly referring to EB or S2Rs (*e.g.*, "expected result/behavior" or "steps to reproduce/recreate"), and keywords commonly used to describe EB (*i.e.*, modal verbs such as "should", "could", or "must", or other terms such as "instead of"). For S2Rs, DeMIBuD-R also detects enumerations (*e.g.*, 1., 2., *etc.*) and itemizations (*e.g.*, '*', '-', *etc.*). If any of the sentences or paragraphs of a bug report matches any of the regular expressions, then DeMIBuD-R labels the report as containing EB/S2Rs, otherwise, DeMIBuD-R labels it as missing EB/S2Rs. DeMIBuD-R extends existing approaches to detect EB/S2Rs [249, 66, 98]. The full list of regular expressions used by DeMIBuD-R can be found in the replication package [89].

4.3 Heuristics-based DeMIBuD

DeMIBuD-H uses part-of-speech (POS) tagging and heuristics to match sentences and paragraphs to our discourse patterns. We implemented each one of the patterns in our catalog by using the Stanford CoreNLP toolkit [156]. For example, to detect EB sentences that follow the discourse pattern SHOULD, DeMIBuD-H first identifies the clauses of a sentence by finding coordinating conjunctions (*i.e.*, tokens tagged as "CC") or punctuation characters (*e.g.*, commas), and splits the sentence using these tokens. Then, for each clause, it identifies the modal terms "should" or "shall" by processing the tokens labeled as "MD" (*i.e.*, modal). Finally, DeMIBuD-H checks for the absence of any predicate that uses negative auxiliary verbs prior to the modal. This is done by identifying the adverb "not" preceded by auxiliary verbs (*i.e.*, the verbs[2] "do", "have", or "be", or the modals "can", "would", "will", "could", or "may"). DeMIBuD-H also checks for the complement after the modal and for some exceptions (*e.g.*, phrases, such as "should be done"). If any of the clauses satisfy these rules, then the sentence is detected as following the SHOULD discourse pattern and it is labeled as an EB sentence. Each pattern implementation is used to classify all sentences/paragraphs in a bug description as having or not having EB/S2Rs. A bug report is labeled as containing EB/S2Rs if at least one sentence/paragraph of the bug report matches any EB/S2R pattern implementation. Otherwise, the bug report is labeled as missing EB/S2Rs.

4.4 Machine Learning-based DeMIBuD

DeMIBuD-ML is based on state-of-the-art approaches in automated discourse analysis and text classification [74, 75, 126], which utilize textual features, such as *n-grams* and *POS tags* (*i.e.*, part of speech tags) [126]. DeMIBuD-ML relies on two binary classifiers, one that detects missing EB, and another one that detects missing S2Rs.

4.4.1 Textual Features

We use our *discourse patterns* as features of bug descriptions for classification purposes. Our patterns capture the structure and (to some extent) the vocabulary of the descriptions. Each EB and S2R pattern is defined as a boolean feature indicating the presence or absence

[2]A verb is a token labeled with one of the VBx POS tags, such as VBD or VBN (*i.e.*, verb in past tense or past participle).

of the pattern in any of the sentences/paragraphs of a bug report. We use the pattern implementations of DeMIBuD-H to produce the pattern features. EB and S2R features are used in turn by the corresponding classifier (*i.e.*, the one for EB or S2Rs, respectively). We also use *n-grams* to capture the vocabulary of bug descriptions. *N*-grams are contiguous sequences of *n* terms in the text. We use unigrams, bigrams, and trigrams, where each *n*-gram is defined as a boolean feature indicating the presence or absence of such an *n*-gram in any of the sentences of a bug report. Finally, we use *POS tags* to capture the type of vocabulary used in the bug descriptions. Similar to *n*-grams, we use contiguous sequences of *n*-POS tags in the text. We define {1, 2, 3}-POS tags as boolean features indicating the presence or absence of a tag combination in any of the sentences of a bug report.

4.4.2 Learning Model

Our current implementation of DeMIBuD-ML uses linear Support Vector Machines (SVMs) (from SVM-Light [125]) to classify the bug reports as missing or not missing EB and S2Rs. Linear SVMs are robust state-of-the-art learning algorithms for high-dimensional and sparse data sets, commonly used for text classification based on *n*-grams [126, 125, 127]. Investigating the use of other classifiers is subject of future work.

4.5 Empirical Evaluation

We conducted an empirical evaluation with the *goal* of determining how accurately DeMIBuD can detect missing EB and S2Rs in bug descriptions and comparing the accuracy of the different instances of DeMIBuD.

4.5.1 Context and Research Questions

The *context* of our study is represented by the *validation bug reports* from the nine software projects used for open coding (see Section 3.3). This data set represents the ground truth (see Table 4.1). The empirical evaluation aims to answer the following research question:

RQ: *Which* DEMIBUD *strategy has the highest accuracy in detecting missing EB and S2R content in bug descriptions?*

To answer the research question, use preprocessed the bug reports in our dataset (see Section 4.5.2), tuned DEMIBUD's parameters and executed the approach using two different settings (see Section 4.5.3), and used standard classification metrics to measure its performance (see Section 4.5.4). We analyze the evaluation results and answer the research question in Section 4.5.5.

Table 4.1: Number of *validation bug reports* missing the EB and S2Rs.

Project	**EB**	**S2Rs**	**Total**
Docker	145 (65.3%)	82 (36.9%)	**222**
Eclipse	98 (66.2%)	63 (42.6%)	**148**
Facebook	137 (74.9%)	113 (61.7%)	**183**
Firefox	78 (36.3%)	56 (26.0%)	**215**
Hibernate	169 (74.1%)	118 (51.8%)	**228**
Httpd	172 (74.8%)	173 (75.2%)	**230**
LibreOffice	131 (62.1%)	57 (27.0%)	**211**
OpenMRS	135 (67.2%)	140 (69.7%)	**201**
Wordpress-A	126 (68.9%)	108 (59.0%)	**183**
Total	**1,191 (65.4%)**	**910 (50.0%)**	**1,821**

4.5.2 Text Preprocessing

We removed uninformative text that is likely to introduce noise to the detection process, by using different text preprocessing strategies. Specifically, we performed *code removal*: deletion of code snippets, stack traces, output logs, environment information, *etc.* This was done by using regular expressions and heuristics, defined after our observations of the text. We also performed *basic preprocessing*, which includes replacing URLs with the "_URL_" meta-token, and removing special characters (*e.g.*, punctuation), numbers, single characters, and tokens starting with numbers. In addition, we performed *stemming* using the Perl interface to SnowBall [12], and *stop-word removal*, which included deletion of common articles, prepositions, and adjectives, by using an adapted version of the Lemur stop word list [56]. The replication package contains the preprocessed bug descriptions, the list of stop words, and the code removal implementation [89].

When assessing the performance of DeMIBuD-R and DeMIBuD-H, we only use *code removal* as the preprocessing strategy, since, by design, these approaches need special characters (*e.g.*, the ones used for itemizations and enumerations), unstemmed vocabulary, and stop words (*e.g.*, "if", "when", "then", *etc.*). The performance of DeMIBuD-ML is determined by using the combination of all preprocessing strategies mentioned above.

4.5.3 Tuning and Evaluation Settings

We used the *discourse bug reports* to test DeMIBuD-R and DeMIBuD-H. This data set contains positive and negative instances that allowed us to test and tune our implementations. Since DeMIBuD detects the absence of EB/S2Rs, a *negative instance* is a sentence/paragraph/report that contains an explicit description of EB/S2Rs, whereas a *positive instance* is a sentence/paragraph/report that misses such a description.

By using the *discourse bug reports*, we determined the patterns that contribute most (and least) to DeMIBuD-H's accuracy. We followed a leave-one-out strategy for each one of the EB and S2R patterns. Having all the patterns activated, we deactivated one pattern at a time and measured DeMIBuD-H's accuracy (*i.e.*, the F_1 score — more details below) without using such a pattern. Overall, we identified three patterns that, when deactivated, drastically deteriorate the accuracy of DeMIBuD-H (*i.e.*, the F_1 score drops drastically[3]). These patterns are: SHOULD for EB; and LABELED_LIST and AFTER for S2Rs. Conversely, we also identified three patterns that drastically improve the accuracy of DeMIBuD-H when they are deactivated, namely, CAN and IMPERATIVE for EB, and CODE_REF for S2Rs. These latter three patterns negatively affect DeMIBuD-H's performance because

[3]The results of the tuning approach can be found in the replication package [89]

they occur frequently in sentences that do not describe EB/S2Rs. Hence, we call these "ambiguous patterns". We measured DeMIBuD-H's accuracy both by using all the patterns and by omitting the ambiguous patterns.

For DeMIBuD-ML, we performed 10-fold cross validation (10CV) using the *validation bug reports*. To avoid over-fitting [106], we used 70%, 20%, and 10% of the bug reports for training, parameter tuning, and testing, respectively. This strategy ensures that all bug reports are used for training, parameter tuning, and testing. The testing data set was used to measure DeMIBuD-ML's accuracy. To follow a realistic approach, we performed 10CV independently on the bug reports of each project. We call this setting *within-project evaluation*. We used stratified sampling to create the folds, thus ensuring that the proportions of negative and positive instances are similar to the proportions of all the reports in the corresponding project (remember that a negative instance indicates the presence of EB/S2Rs, while a positive instance indicates the absence). To assess feature generality in DeMIBuD-ML, we also conducted a *cross-project evaluation*, in which the bug reports of one project were used for testing, and the reports of the remaining eight projects were used for training and parameter tuning (approx. 80% and 20% of the reports were used for training and parameter tuning, respectively).

In our experiments, we tuned the penalty parameter C of the linear SVMs by using the parameter tuning data set of each fold. Larger C values mean higher penalty to classification errors. We experimented with the following parameter values: 1×10^{-4}, 2.5×10^{-4}, 5×10^{-4}, 7.5×10^{-4}, ..., 5, 7.5. We chose the best parameter C by maximizing the F_1 score of the trained SVMs to detect missing EB and S2Rs. We found that the parameters leading to the best accuracy fall in the ranges [0.05, 0.5] and [0.0025, 0.1] for EB and S2Rs, respectively.

4.5.4 Evaluation Metrics

We use standard metrics in automated classification to measure the accuracy of our approaches, namely, precision, recall, and F_1 score [106]. Precision is defined as the percentage of bug reports predicted as missing EB/S2Rs that are correct with respect to the ground truth ($Precision = TP/(TP + FP)$). Recall is the percentage of bug reports missing EB/S2Rs that are correctly predicted as missing EB/S2Rs ($Recall = TP/(TP + FN)$). F_1 score is the harmonic mean between precision and recall, which gives a combined measure of accuracy.

Intuitively, we prefer higher recall, as in a practical setting, we want DeMIBuD to alert reporters whenever EB or S2Rs are missing in their bug descriptions. Nonetheless, we also want DeMIBuD to achieve high precision, as many false alerts would hinder its usability. Experiments with users are needed to assess acceptable trade-offs between recall and precision. We leave such studies for future work. Therefore, we focus on the F_1 score as an accuracy indicator, as we did for the tuning. When two configurations yield the same F_1 score, we prefer the one with higher recall.

4.5.5 Results and Discussion

We present and discuss the accuracy achieved by our three instances of DeMIBuD when detecting the absence of EB and S2Rs using different strategies and features (see Table 4.2).

Table 4.2: Overall *within-project* detection accuracy of DeMIBuD.

Approach	**Strategy or Features**	**EB (Avg.)**			**S2Rs (Avg.)**		
		Prec.	**Recall**	**F_1**	**Prec.**	**Recall**	**F_1**
DeMIBuD-R	-	86.0%	85.9%	85.9%	63.3%	92.4%	74.3%
DeMIBuD-H	all patterns	96.7%	46.1%	62.2%	84.5%	31.0%	44.3%
DeMIBuD-H	no ambiguous patterns	95.1%	76.6%	84.7%	81.6%	38.5%	51.2%
DeMIBuD-ML	pos	73.8%	93.1%	82.0%	60.8%	75.8%	66.8%
DeMIBuD-ML	*n*-gram	75.1%	97.6%	84.7%	66.4%	83.4%	73.4%
DeMIBuD-ML	pos + *n*-gram	76.0%	95.1%	84.2%	65.3%	79.2%	71.1%
DeMIBuD-ML	patterns	85.9%	93.2%	89.4%	63.5%	80.3%	70.7%
DeMIBuD-ML	patterns + pos	77.9%	92.9%	84.6%	65.4%	76.0%	69.9%
DeMIBuD-ML	patterns + *n*-gram	76.9%	97.0%	85.6%	69.2%	83.0%	74.9%
DeMIBuD-ML	pos + patterns + *n*-gram	76.8%	95.8%	85.1%	67.2%	80.9%	73.0%

DeMIBuD-R's Accuracy. DeMIBuD-R achieves 85.9% avg. recall and 86% avg. precision when detecting missing EB (see Table 4.2). Based on F_1, this is the second best approach across all versions of DeMIBuD. Our analysis reveals that DeMIBuD-R fails to detect missing EB in 168 bug reports that do not describe EB (*i.e.*, false negatives). This is mainly due to the inherent imprecision of keyword matching via regular expressions. For example, we found usages of modal verbs to express OB instead of EB, as in *"This problem could also be related to some sites not copying URLs..."* (from Firefox 319364), and modal verbs appearing in error messages: *"... and the error could not load the item appeared on the screen"* (from Wordpress-Android 859). We also observed that DeMIBuD-R detects missing EB in 167 bug reports that describe EB (*i.e.*, false positives), as they do not match the keywords used by DeMIBuD-R. For example, we found EB sentences phrased with *"used to"*, as in *"...you used to be able to add new Obs to an already existing encounter order..."* (from OpenMRS TRUNK-211).

Regarding S2Rs, DeMIBuD-R achieves the highest recall (93.4%) but also one of the lowest precision values (*i.e.*, 63.3%) across the different versions of DeMIBuD. DeMIBuD-R's recall suggests that bug reports missing S2Rs usually do not contain explicit S2R keywords and/or itemizations/enumerations. Yet, in the few false negatives produced by DeMIBuD-R (*i.e.*, 18), we found non-S2R sentences using S2R keywords, such as *"I tried to reproduce the issue without luck..."* (from Wordpress-Android 1318), or templates that contain S2R keywords but are filled in with non-S2R content, for example, *"Steps To Reproduce: Unsure how to reproduce..."* (Eclipse 229806). DeMIBuD-R flagged missing S2Rs in 530 bug reports describing S2R (*i.e.*, false positives). This is somehow expected as users describe S2Rs using alternative wordings to enumerations/itemizations, which are not keyword specific. For example, users can describe S2Rs in a narrative way: *"Open the history view on a file with interesting revisions. Click the date column to sort by date..."* (from Eclipse 17774). Overall, DeMIBuD-R ranked as the second most accurate detector across all versions of DeMIBuD, in terms of F_1 score (*i.e.*, 74.3%). The results indicate that DeMIBuD-R is accurate in detecting missing S2R, yet produces a rather large number of false alarms.

DeMIBuD-H's Accuracy. When all the patterns are used, DeMIBuD-H is able to detect missing EB with 46.1% recall and 96.7% precision. When we deactivate the ambiguous EB patterns (*i.e.*, IMPERATIVE and CAN), DeMIBuD-H's recall improves substantially (*i.e.*, from 46.1% to 76.6%) at almost the same precision (*i.e.*, 95.1%). This large recall improvement is explained by the large number of bug reports missing EB that contain sentences matching the ambiguous patterns, which lead to many false negatives (*i.e.*, failing to detect missing EB). We found 438 and 305 reports missing EB that contain IMPERATIVE and CAN sentences (*i.e.*, 36.8% and 25.6% of the bug reports that do not describe EB), respectively. We observed that IMPERATIVE sentences are usually used to describe S2Rs, for example, *"1. Create a container with volumes in docker 1.8.3"* (from Docker 18467), or ask for information to the reporter via templates, for example, *"**Describe the results you received:**"* (from Docker 27112). CAN sentences describe other non-EB content, for example, *"the user can only tell the difference when he recognizes..."* (from Firefox 293527).

When the ambiguous EB patterns are deactivated, we observe that the main reason for false negatives is sentences describing non-EB content, yet following the SHOULD EB pattern. We found conditional sentences expressing actions: *"If that's the case, we should document this on the wiki..."* (from OpenMRS TRUNK-4907); questions using the modal "should": *"... should following tags be unavailable while signed out?"* (from Wordpress-Android 3270); and sentences expressing other types of non-EB content: *"OpenMRS shouldn't bomb in this situation"* (from OpenMRS TRUNK-2992). Our analysis of the 50 false positives produced by DeMIBuD-H revealed that our pattern implementation is unable to match some sentences. In addition, we found a handful of bug reports containing EB sentences that are not captured by any of our EB patterns, for example, *"...works as expected (as in the process is not killed)"* (from Docker 11503), or *"With FF2, the user sees the tab transition smoothly to the new tab with no nasty white flash"* (from Firefox 393335).

Regarding S2Rs, when all the patterns are used, DeMIBuD-H has the lowest recall (*i.e.*, 31%) but the highest precision (*i.e.*, 84.5%). We observe 7.5% recall improvement and 2.9% precision deterioration when DeMIBuD-H relies on all the patterns except CODE_REF (*i.e.*, the ambiguous S2R pattern). We found 338 bug reports missing S2Rs but containing sentences matching the CODE_REF pattern (*i.e.*, 37.1%). The main reasons behind the false negatives are the imprecision of our implementation (*i.e.*, regarding heuristics, sentence parsing, or code preprocessing) and the presence of ambiguous sentences, such as *"Here are the definitions of the file systems:"* (from Httpd 37077). When the CODE_REF pattern is deactivated, we observe two main reasons for false negatives, namely, the imprecision of our implementation and ambiguous content (*i.e.*, sentences and paragraphs describing non-S2R content yet following other S2R patterns). Regarding the latter, we found non-S2R paragraphs and sentences phrased imperatively that describe solutions: *"Possible solutions: ... 1. Disable Tomcat's default ..."* (from OpenMRS TRUNK-1581); or actions that do not intend to replicate the OB: *"See the user edit page for how the void patient..."* (from OpenMRS TRUNK-1781). Other ambiguous cases include non-S2R conditional sentences describing high-level tasks: *"I noticed that HSQLDB is not enforcing... while trying to troubleshoot a particular..."* (from OpenMRS TRUNK-27); and sentences that convey actions expressed in present perfect tense, present tense, or past tense, for example, *"I also asked about this in the Hibernate... and Steve Ebersole said that..."* (from OpenMRS TRUNK-2). These types of discourse are also used to describe S2Rs and are captured by our S2R patterns. Our analysis of false positives produced by DeMIBuD-H revealed content ambiguity and unusual text structure as the main reasons for hindering precision. We found labeled lists of S2Rs where each step was written as a separate paragraph (as in LibreOffice 77431);

paragraphs containing different sentences that describe S2Rs and other types of information, for example, *"I'm using LibreOffice 4.3.6.2... I downloaded 4.3.7 and installed... And I knew 4.3.7 requires..."* (from LibreOffice 91028); itemizations describing OB rather than S2Rs (as in LibreOffice 78202); sentences not related to OB replication, for example, *"I am seeing junk characters and I have to change the encoding setting manually"* (from Httpd 49387); and ambiguous sentences describing actions: *"I fixed the problem by using..."* (from Httpd 42731). Overall, we observed more content ambiguity related to S2Rs than to EB. This is one of the reasons for the lower accuracy of DeMIBuD-H (and other DeMIBuD versions) when detecting missing S2Rs.

DeMIBuD-H's high precision and low recall (when detecting missing EB and S2Rs) are explained by the focus of our pattern implementations on identifying all different ways to describe EB and S2Rs (*i.e.*, identifying EB/S2Rs in most bug reports), without focusing on filtering non-EB/S2R content that is similar to EB/S2R (*i.e.*, it incorrectly predicts EB/S2R in many bug reports). Compared to DeMIBuD-R, DeMIBuD-H's overall accuracy is lower when detecting missing EB and S2Rs in bug descriptions (in terms of F_1 score). The two main reasons for such (in)accuracy are: (1) imprecision of our heuristics, and (2) ambiguous content in bug descriptions. While the former issue may be addressed by refining some of the patterns, the latter one is more challenging. In any case, DeMIBuD-H's main advantage over the other versions of DeMIBuD is its ability to produce very few false alarms.

DeMIBuD-ML's Accuracy. When detecting missing EB, DeMIBuD-ML achieves the highest recall (*i.e.*, between 92.9% and 97.6%) at the expense of precision (*i.e.*, between 73.8% and 85.9%). The features used by DeMIBuD-ML that lead to the highest (*i.e.*, 97.6%) and lowest (*i.e.*, 92.9%) recall are *n-grams* and *patterns + POS tags*, respectively. The features that lead to the highest (*i.e.*, 85.9%) and lowest (*i.e.*, 73.8%) precision are *patterns* and *POS tags*, respectively. We observe that *n-grams* always increase recall when combined with other features, and *POS tags* deteriorate recall when combined with *patterns*. *Pattern* features always improve precision when combined with other features (at the expense of recall, unless they are combined with *n-grams*). The highest F_1 score (*i.e.*, 89.4%–85.9% precision and 93.5% recall) is achieved by DeMIBuD-ML using *pattern* features. We consider this version and configuration of DeMIBuD as the best for detecting missing EB.

DeMIBuD-ML detects missing S2Rs with recall ranging between 75.8% and 83.4%. These recall values are lower than that achieved by DeMIBuD-R. DeMIBuD-ML achieves lower precision than DeMIBuD-H (*i.e.*, between 60.8% and 69.2%). However, DeMIBuD-ML represents the best compromise, achieving the highest F_1 score (*i.e.*, 74.9%–69.2% precision and 83% recall) when using the *patterns + n-gram* features. Once again, among the different features used by DeMIBuD-ML, we observe that individual *n-grams* are the features that lead to the highest recall (*i.e.*, 83.4%) and always improve it when combined with other features. Conversely, *POS tags* are the features that lead to the lowest recall (*i.e.*, 75.8%) and always deteriorate it when combined with other features. DeMIBuD-ML based on *POS tags* achieves the lowest precision (*i.e.*, 60.8%), while *n-grams* lead to the highest (*i.e.*, 66.4%) and always improve it when combined with other features. Although individual pattern features lead to lower precision (*i.e.*, 63.5%), they always lead to precision improvement when combined with other features. The highest F_1 score (*i.e.*, 74.9%–69.2% precision and 83% recall) is achieved by DeMIBuD-ML using *patterns + n-gram* features. We consider this configuration of DeMIBuD as the best for detecting missing S2Rs.

Explaining the effect of individual features on the results is harder than with heuristics or regular expressions. However, we conjecture that the positive effect of *n-grams* is its ability to capture the vocabulary and (to some extent) the structure of EB/S2Rs and non-EB/S2Rs discourse. Our *patterns* also capture such characteristics; however, they further capture the discourse structure. POS tags focus on capturing the type of vocabulary and (to some extent) the structure, which has a negative impact on DeMIBuD-ML's accuracy. In any case, all features are insufficient to resolve content ambiguity, especially regarding S2Rs. In our future work, we plan to address this problem by capturing semantic properties of the text, via semantic frames [64] or rhetorical relations [75].

DeMIBuD-ML's Cross-Project Accuracy. The machine-learning-based DeMIBuD achieves the best F_1 score, but relies on supervised training. Obtaining training data from a project often poses challenges, so using training data from other projects is often desirable. We analyze DeMIBuD-ML's accuracy when bug reports from different projects are used to train its underlying learning model. We compare DeMIBuD-ML's accuracy when using *cross-project* (see Table 4.3) and *within-project* training (see Table 4.2).

Table 4.3: Overall *cross-project* accuracy of DeMIBuD-ML.

Features	**EB (Avg.)**			**S2Rs (Avg.)**		
	Prec.	**Recall**	**F_1**	**Prec.**	**Recall**	**F_1**
pos	67.4%	94.3%	77.8%	60.1%	73.9%	63.8%
n-gram	77.9%	96.4%	86.0%	68.2%	86.3%	75.0%
pos + *n*-gram	76.5%	96.1%	85.1%	66.3%	86.5%	73.5%
patterns	87.3%	92.3%	89.5%	64.9%	89.1%	73.9%
patterns + pos	81.9%	92.2%	86.3%	63.7%	84.2%	71.5%
patterns + *n*-gram	86.9%	92.5%	89.5%	68.3%	87.5%	76.0%
pos + patterns + *n*-gram	85.2%	92.6%	88.7%	69.2%	82.5%	74.4%

In the case of EB, using *cross-project* training, we observe that DeMIBuD-ML's precision improves for all type of features (except for *pos*): 3% avg. improvement[4]. Conversely, DeMIBuD-ML's recall decreases 1.2% on average, except for *pos* and *pos + n-grams*. In the case of S2R, we observe that DeMIBuD-ML's precision improves for some features (*i.e.*, *n-gram*, *pos + n-gram*, *patterns*, and all features combined) and deteriorates for others (*i.e.*, *pos*, *patterns + pos*, and *patterns + n-gram*). We observe little precision improvement on average (*i.e.*, 0.4%). Instead, DeMIBuD-ML's recall improves substantially for most

[4]The average improvement is computed by averaging the differences between the *cross-* and *within-project* precision values, across the different types of features. The same computation was done for recall.

features (except for *pos*): 4.5% avg. improvement. The *pattern* features improve precision (*i.e.*, by 1.4%) and achieve the highest recall improvement among all features (*i.e.*, 8.8%). Overall, DeMIBuD-ML's accuracy is higher when using *cross-project* training than when with *within-project* training. One likely explanation is that the training data used in the *cross-project* training includes more patterns.

Remarkably, DeMIBuD-ML based on *patterns + n-gram* has the best accuracy[5] for both EB and S2Rs (see Table 4.3). Its high *cross-project* accuracy indicates that DeMIBuD-ML is extremely robust to the training strategy, and can be highly useful in a practical setting where labeled data from a new project is unavailable. This means that we can deploy DeMIBuD-ML in different projects (to the ones we used) without retraining and expect similar accuracy levels.

4.5.6 Threats to Validity

We briefly discuss the threats to the validity of DeMIBuD's evaluation. To strengthen the *internal validity*, we mitigated the effect of different design and experimentation decisions (*e.g.*, text preprocessing) by tuning our three instances of DeMIBuD on data sets different from the ones used to estimate DeMIBuD's accuracy. As we relied on the 2,912 bug reports manually coded in our first study (see Chapter 3), our results may or may not generalize beyond these set of bug reports. However, we assessed the generalization DeMIBuD-ML's accuracy by using a *cross-project* evaluation, thus strengthening (to some extent) the *external validity* of the evaluation.

[5]While individual *patterns* and *patterns + n-gram* features lead to the same F_1 score, the latter ones are preferred because they lead to a slightly higher recall.

4.6 Related work

Our research relates to work on issue classification [115, 247, 58, 215, 227], which relies on machine learning and textual features to classify issues as (for example) features requests, enhancements, or bug reports. Similar approaches have been proposed to classify e-mails [101], app reviews [159], forums [243], explanations of Application Programming Interfaces in tutorials [172], and, outside Software Engineering, discourse elements in essays [74, 75]. The essential difference between our (SVM-based) approach and existing software content classifiers is the use of discourse patterns from bug descriptions. More related to our research is Davies *et al.* [98]'s work, which proposed the explicit use of search terms (*e.g.*, *"observed behavior"*) to detect OB, EB, or S2R content in bug reports. Unfortunately, this approach produces excessive undetected cases (*i.e.*, false negatives).

4.7 Conclusions

Based on the discourse patterns extracted in our first study, we developed DeMIBuD, an automated approach that detects missing EB and S2Rs in bug descriptions. We implemented and evaluated three versions of DeMIBuD, based on regular expressions, heuristics and natural language processing, and machine learning (ML). Our ML-based approach (*i.e.*, DeMIBuD-ML) proved to be the most accurate in terms of F_1 score (*i.e.*, 89.4% for EB, and 74.9% for S2Rs), yet the other versions of DeMIBuD achieve comparable accuracy without the need for training. DeMIBuD-ML proved to be robust with respect to *within-* and *cross-project* training, which means that we can deploy it in different projects (to the ones we used) without retraining and expect it to achieve high accuracy detection.

Our future work will focus on (i) studying acceptable recall/precision trade-offs from the DeMIBuD users' perspective, and (ii) addressing bug content ambiguity to improve DeMIBuD's accuracy. On the one hand, we plan to extend DeMIBuD to detect missing

OB and integrate it in existing bug tracking systems, in order to evaluate its usefulness in practice. We expect to collect qualitative data, via online feedback capturing and user interaction tracking. On the other hand, we plan to adapt DeMIBuD to work at sentence-level, and use additional features to mitigate the text ambiguity. Specifically, we will explore the use of semantic frames [64] or rhetorical relations [75], to further characterize OB/EB/S2Rs and non-OB/EB/S2Rs sentences.

4.8 Acknowledgments

This chapter was published and presented at the 11th Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17). This publication [88] is a collaborative effort with Jing Lu and Dr. Vincent Ng from The University of Texas at Dallas; Fiorella Zampetti and Dr. Massimiliano Di Penta from the University of Sannio; Dr. Laura Moreno from Colorado State University; and Dr. Gabriele Bavota from Università della Svizzera italiana.

Oscar Chaparro, Jing Lu, Fiorella Zampetti, and Laura Moreno contributed to the design, implementation, and evaluation of the proposed approach. Gabriele Bavota and Massimiliano Di Penta contributed to the evaluation design. Vincent Ng contributed to the design of the machine learning version of our approach and provided feedback to the empirical evaluation.

CHAPTER 5
ASSESSING THE QUALITY OF THE STEPS TO REPRODUCE IN BUG DESCRIPTIONS[1]

5.1 Introduction

Along with the observed and expected behavior, bug descriptions often contain the steps to reproduce (S2Rs) the bug. The steps to reproduce in bug descriptions are essential for developers to replicate and correct the bugs [139, 249]. Unfortunately, in many cases, the S2Rs are unclear, incomplete, and/or ambiguous. So much so that developers are often unable to replicate the problems, let alone fix the bugs in the software [1, 108, 248, 73, 115, 249, 109, 131].

Ideally, low-quality S2Rs in bug reports should be identified at reporting time, such that the reporters would have a chance to correct them. With that in mind, we propose an approach, called EULER, that automatically analyzes the textual description of a bug report, assesses the quality of the S2Rs, and provides actionable feedback to reporters about: ambiguous steps, steps described with unexpected vocabulary, and missing steps in the bug report. In this chapter, we present the approach and we evaluate an implementation geared towards S2Rs corresponding to GUI[2]-level interactions in Android applications. EULER can be adapted to support any GUI-based system.

EULER leverages neural sequence labeling [124, 138] in combination with previously discovered discourse patterns [88] to identify S2R sentences in bug reports and then uses dependency parsing [156] to extract individual S2Rs. Next, it matches the identified S2Rs

[1]© 2019 ACM. Reprinted with permission from Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, Kevin Moran, Andrian Marcus, Massimiliano Di Penta, Denys Poshyvanyk, and Vincent Ng, “Assessing the Quality of the Steps to Reproduce in Bug Reports”, Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE’19).

[2]Graphical User Interface

to program states and GUI-level application interactions, represented in a graph-based execution model. A successful match indicates that the S2R precisely corresponds to an app interaction (*i.e.*, it is of high-quality). Conversely, a low-quality S2R may match to multiple screen components or app events, may not match any application state or interaction, or it may require the execution of additional steps. Euler assigns to each S2R quality annotations that provide specific feedback to the reporter about problems with the S2Rs.

We envision Euler being used in three different scenarios: (1) providing automated feedback to the reporter at bug reporting time, prompting a rewrite of the bug report; (2) providing useful information (*e.g.*, missing S2Rs) to the developers attempting to reproduce the bug; and (3) supporting automated approaches for test case generation [109, 131].

5.2 Euler: Assessing S2R quality

We describe Euler[3], an approach that automatically identifies and assesses the quality of the steps to reproduce (S2Rs) in bug reports. We focus on bug reports for GUI-based Android apps, yet Euler can be adapted to work for other platforms. The input of Euler is the textual description of a bug report and the executable file of the Android application affected by the reported bug. The output is a Quality Report (QR), which contains a set of Quality Annotations (QAs) for each S2R, automatically identified from the bug description. The QAs are described in Table 5.1 of Section 5.2.8.

Figure 5.1 shows Euler's main components and workflow. The first two major steps in Euler's workflow is automatically (1) identifying the S2Rs from the bug report (see Section 5.2.1), and (2) building a system execution model that captures user interactions of the app (see Section 5.2.2). Euler's S2R quality assessment algorithm receives as input the identified S2Rs from the bug report and the system execution graph. The output is

[3]Euler: assEssing the qUaLity of the steps to rEproduce in bug Reports

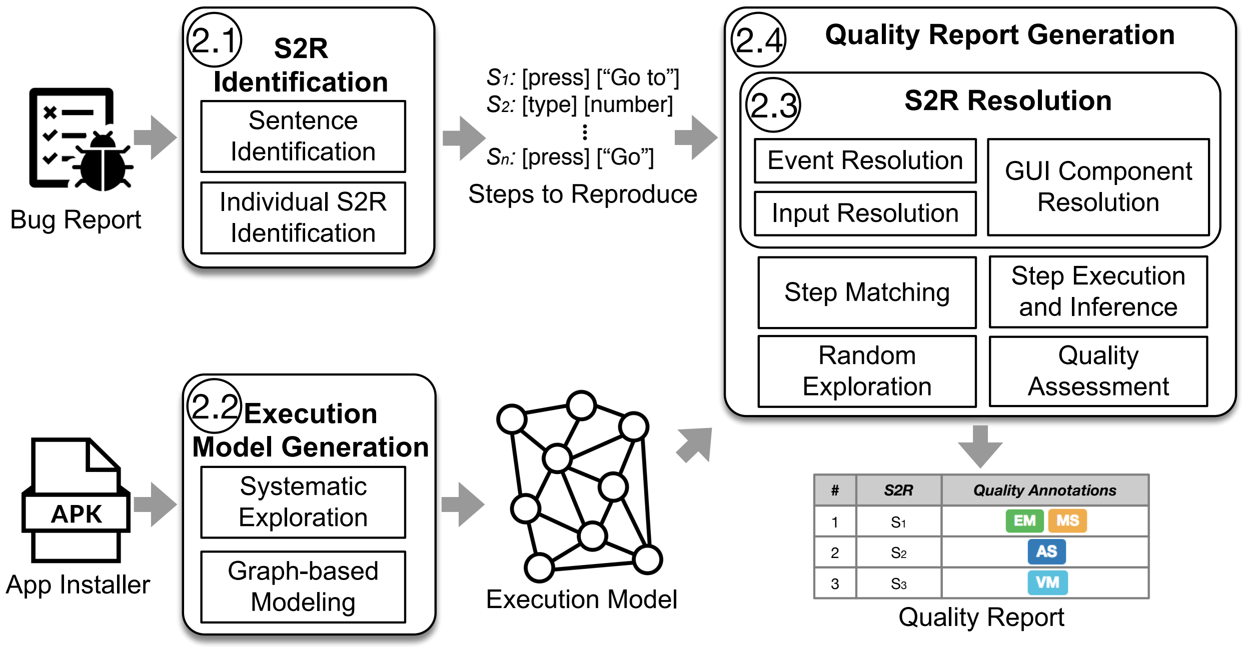


Figure 5.1: EULER's workflow and main components.

a Quality Report (QR), providing an assessment and feedback for each S2R. Based on a matching algorithm (see Section 5.2.3), EULER resolves the app interaction that most-likely corresponds to a S2R, given an app screen (see Section 5.2.4). This is called S2R resolution, which is used to match the identified S2Rs to the nodes and transitions of the model (see Section 5.2.5). The matched interactions are executed and the missing ones are inferred (see Section 5.2.6). If the step matching fails, then EULER executes random application exploration (see Section 5.2.7). In the end, the QR is generated based on the matching results of the S2Rs against the execution model (see Section 5.2.8).

5.2.1 S2Rs Identification

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

Identification of S2R Sentences. Euler identifies S2R sentences in a bug report using a neural sequence labeling model [124, 138], which contains the following components:

Model Input. The model input consists of paragraphs in the bug report. Each paragraph is a sequence of sentences, and each sentence is a sequence of words. As there are dependencies between S2R sentences (*i.e.*, often they appear in sentence groups), we use the Beginning-Inside-Outside (BIO) tagging approach [185], where for each sentence in a paragraph, we assign: (1) the label [B-S2R] if the sentence begins a S2R description; (2) the label [I-S2R] if the sentence is inside the S2R description; or (3) the label [O] if it is outside (*i.e.*, not part of) the S2R description.

Word Representations. We represent each word by concatenating two components: word embeddings and character embeddings. We use pre-trained word vectors from a corpus of 819K bug reports, collected from 358 open source projects, to capture word-level representations. In order to handle vocabulary outside of this corpus, we use a one-layer Convolutional Neural Network (CNN) with max-pooling to capture character-level representations [153]. We model word sequences in a sentence by feeding the above word representations into a Bidirectional Long Short-Term Memory (Bi-LSTM), which has been shown to outperform alternative structures [228]. The hidden states of the forward/backward LSTMs are concatenated for each word to obtain the final word sequence representation.

Sentence Representations. As suggested by Conneau *et al.* [94], we adopt the simple (yet effective) approach of averaging the vectors of words composing a sentence for capturing sentence-level properties. We represent each sentence by concatenating the averaged word representations from the previous step and a one-hot feature vector that encodes the discourse patterns we inferred in our first study (see Chapter 3), which capture the syntax and semantics of S2R descriptions as well as sentences describing the system's observed behavior (OB) and expected behavior (EB).

Inference Layer. In order to model label dependencies, we use a Conditional Random Field (CRF) for inference instead of classifying each sentence independently. CRFs have been found to outperform alternative models [228]. The output of the inference layer is a label for each sentence where the [B/I-S2R] labels indicate a S2R sentence and the [O] label indicates a non-S2R sentence. Section 5.3.3 details the model implementation, training, and evaluation.

Identification of Individual S2Rs. Once the S2R sentences are identified, EULER uses dependency parsing [156] to determine the grammatical relations between the words in each sentence and extract the individual S2R from them. EULER utilizes the Stanford CoreNLP toolkit [156] for extracting the grammatical dependency tree of S2R sentences. This tree varies across different types of sentences (*e.g.*, conditional, imperative, passive voice, *etc.*). Therefore, EULER implements a set of algorithms that extract the relevant terms from the dependency trees of each sentence type.

An individual S2R complies with the following format:

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

where the *[action]* is the operation performed by the user (*e.g.*, tap, minimize, display, *etc.*), the *[object]* is an "entity" directly affected by the *[action]*, and *[object2]* is another "entity" related to the *[object]* by the *[preposition]*. An "entity" is a noun phrase that may represent numeric and textual system input, domain concepts, GUI components, *etc.* A S2R example is: *"[create] [entry] [for] [purchase]"*.

We illustrate EULER's algorithm to identify individual S2Rs from conditional S2R sentences. The bug report #256 [6] from the GnuCash mobile application [29] contains the conditional S2R sentence: *"When I create an entry for a purchase, the autocomplete list shows up"*. Figure 5.2 depicts the tree of grammatical dependencies for this sentence. To extract the S2R from the parsed grammatical tree, EULER first locates the adverb *'When'* that is the adverbial modifier (*advmod*) of the verb (*'create'*). If this step is successful, then EULER verifies the existence of an adverbial clause modifier (*advcl*) between the verb and its

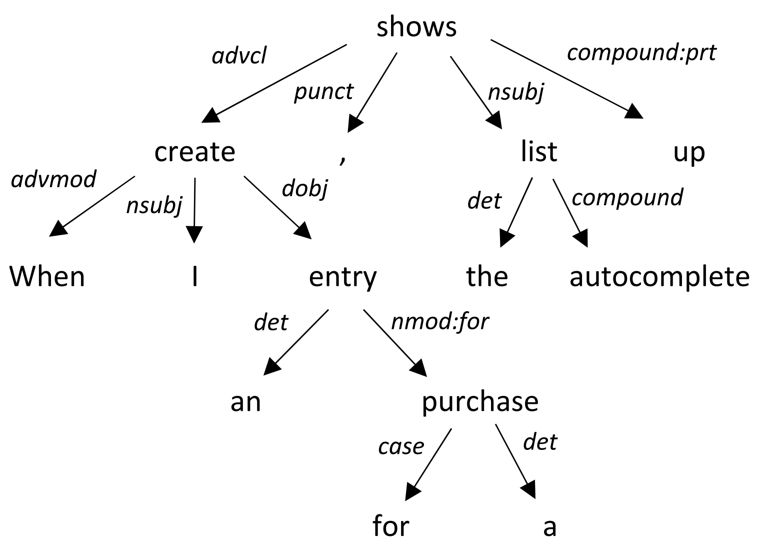


Figure 5.2: Grammatical tree for the sentence: *"When I create an entry for a purchase, the autocomplete list shows up"*.

parent word. If this is the case, then Euler captures the verb (*'create'*) as the S2R's *[action]*. Otherwise, the sentence is discarded, as it does not follow the grammatical structure of a conditional sentence. Next, Euler locates the nominal subject (*nsubj*), in this case, the word *'I'*. Euler captures the *[object]* by locating the verb's direct object (*dobj*), *'entry'* in the example, and identifies the nominal modifier of the direct object (*nmod:for*) as the *[object2]*, and the *[preposition]* of the nominal modifier (*case*). The resulting S2R, identified from the conditional sentence, is: *"[create] [entry] [for] [purchase]"*.

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

5.2.2 Execution Model Generation

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

In its current implementation, Euler utilizes a modified version of CrashScope's *GUI-ripping Engine* [163, 165] to generate a database of application execution data in the form of sequential interactions. This Engine is an automated system for dynamic analysis of Android applications that utilizes a set of systematic exploration strategies and has been shown to exhibit comparable coverage to other automated mobile testing techniques [163]. A detailed description of the engine can be found in Moran *et al.*'s previous work [163, 165].

Euler's next task is the generation of a graph that abstracts the sequential execution database produced by CrashScope's Engine. The granularity of states in this graph is important, as it will serve as an index for matching the identified natural language S2Rs with execution information. For instance, if the graph were built at the activity-level (meaning that each activity recorded by CrashScope represents a unique state in the graph), then there is potential for information loss, as the GUI-hierarchy of a single activity may change as a result of actions performed on it [62].

To avoid information loss, Euler generates a directed graph $G = (V, E)$, where V is the set of unique **application screens** with complete GUI hierarchies, and E is a set of **application interactions** performed on the screens' GUI components. In this model, two screens with the same number, type, size, and hierarchical structure of GUI components are considered a single vertex. E is a set of unique tuples of the form (v_x, v_y, e, c), where e is an application event (*e.g.*, tap, type, swipe, *etc.*) performed on a GUI component c from screen v_x, and v_y is the resulting screen right after the interaction execution. Similar execution models have been proposed in prior research on mobile app testing [234]. Each edge stores additional information about the interaction, such as the application data input (only for *type* events) and the interaction execution order dictated by the systematic exploration. The graph's starting node has one outgoing interaction only, which corresponds to the application launch. A GUI component is uniquely represented by a type (*e.g.*, a button or a text field), an identifier, a label (*'OK'* or *'Cancel'*), and its size/position in the screen. Additional information about a component is stored in the graph, for example, the component description given by the developer and the parent/children components. It is important to note that an interaction performed on a component can lead to the current screen. In this case, the source and target screens of the interaction corresponds to the same vertex, which means that the edge is a loop. The execution order of the interactions dictates the order of the loops.

5.2.3 Base Matching Algorithm.

Before describing how Euler resolves the app interaction to a S2R, we describe the algorithm used to match a textual sequence (*i.e.*, a query) to a GUI component from a screen.

The algorithm's input is a textual query q (*i.e.*, a sequence of terms), a list of GUI components GC (sorted in the order of appearance in a screen), and the application event e identified from the S2R. The output is the GUI component (from GC) most relevant to the query. The relevancy is determined by a set of heuristics and a scoring mechanism based on textual similarity. The algorithm comprises the following steps:

1. If q contains terms referring to the application or device screen (*e.g.*, 'screen', 'phone', *etc.*), then the first non-tappable component of the current screen (from top to down) is selected and returned as the most relevant component.

2. If q contains terms that refer to a component type, such as *text field* or *button*, then Euler checks if there is only one component in GC of that type (the first type found in the query). If that is the case, then the algorithm selects and returns such a component as the most relevant component.

3. If q does not contain any terms related to component types, then Euler computes a similarity score between q and each component c from GC and selects a set of candidates most-relevant to the query. The similarity score is computed as:

$$similarity(s_1, s_2) = \frac{|LCS(s_1, s_2)|}{avg\left(|s_1|, |s_2|\right)} \tag{5.1}$$

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

The similarity score accounts for common terms between the sequences and the order in which they appear. The order is important because the matching process should be as precise as possible for producing an accurate S2R quality assessment. Before computing the similarity, EULER applies lemmatization to the input word sequences (using the Stanford CoreNLP toolkit [156]).

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

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

1. There is one candidate. EULER returns such a component and the algorithm ends.

2. There is more than one candidate. To determine the most-relevant component, EULER executes a set of heuristics. For each component, if its type is Layout and it has only one child in the GUI hierarchy, then such a child is returned and the process ends. If none of the candidates satisfy the condition above, but all candidates are of the same type (*e.g.*, text fields), then the component with the highest similarity score is returned. Otherwise, EULER analyzes the candidates with respect to the event e. If

e is a typing event, and there is one text field among the candidates, then the field is returned. Otherwise, if e is a 'tap' or 'long tap' and there is only one button among the candidates, then EULER returns such a component. Otherwise, the algorithm ends with a multiple-match result and the candidates are saved for providing the quality feedback to the user.

3. There are no candidates. EULER reformulates the query following a query replacement approach, where a set of predefined synonyms for query terms are used as new queries. If there are no synonyms for the query terms, then the algorithm stops and returns a mismatch result. Each query is executed and if any matches a component, then it is returned. Otherwise, the process ends with a mismatch.

5.2.4 Step Resolution

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

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

First, EULER finds the *action group* that the *[action]* from the S2R corresponds to. An *action group* is a category for verbs having a similar meaning and used to express an app interaction. EULER finds the action group by matching the *[action]*'s lemma to the lemma of each verb in the group. EULER supports six action groups, namely OPEN, LONG_CLICK, CLICK, SWIPE, TYPE, and ROTATE. Each group has a set of verbs (*e.g.*, edit, input, enter, insert, *etc.* for TYPE). We defined the groups by analyzing the vocabulary used in the bug reports and applications used by Moran *et al.* [164, 163].

When the *[action]* maps to multiple action groups, EULER resolves the correct group by analyzing the *[object]* and *[object2]* from the S2R (*e.g.*, by identifying GUI-component types in them or matching these to screen components using the matching algorithm described in Section 5.2.3). Only the groups TYPE, CLICK, and ROTATE have common verbs. If EULER fails to disambiguate the action group, then it flags the S2R's *[action]* as matching multiple events and saves the corresponding action groups for providing user feedback.

If the *[action]* does not match an action group, then the verb is likely to refer to a generic interaction or an application feature (*e.g.*, "*[create] [purchase]*"). In this case, EULER assumes the *[action]* is expressed in the properties of a GUI component (*i.e.*, its ID, description, or label). Then, EULER attempts to resolve a GUI component that matches the whole S2R or the *[action]*, by using the matching approach (see Section 5.2.3). If there is a matched component, the action group is determined as CLICK (if the component is tappable), as LONG_CLICK (if the component is long-tappable), or TYPE (if the component is type-able). Otherwise, the event resolution process fails with an event mismatch result.

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

All the keywords mentioned in this section are based on our experience with Android apps and the analysis of Moran *et al.*'s bug reports and apps [164, 163].

GUI Component Resolution. The next step in EULER's step resolution workflow is determining the GUI component in the current screen that the event should perform on, according to the S2R. This step is completed only for tap, long tap, tap on menu button, and type events, as they are the only ones that require a component.

EULER uses the S2R constituents as queries, depending on the identified event e for a S2R. These queries are executed using the base matching algorithm of Section 5.2.3 to find the GUI component that the S2R most likely refers to.

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

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

1. A S2R with a literal in *[object]*, a non-literal in *[object2]*, and the *[preposition]* is one of the following: "on", "in", "into", "for", "of", "as", *etc.* For these cases, the *[object2]*

is used as query. For example, for the S2R *"[enter] ['10'] [on] [price]"*, the term *price* is used as query.

2. A S2R with a non-literal in *[object]*, a literal in *[object2]*, and the *[preposition]* is one of the following: "to" or "with". Then, the *[object]* is used as query. For example, for the S2R *"[set] [price] [to] [10]"*, the term *price* is used as query.

3. A S2R where the *[object]* is a literal and the *[preposition]* and *[object2]* are null (*e.g.*, *"[enter] ['10']"*), EULER selects and returns the focused component in the current screen, if any.

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

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

5.2.5 Step Matching

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

Starting from the current state, EULER traverses the graph in a depth-first manner until n levels have been reached. EULER performs step resolution on each state (Section 5.2.4). The result is a set of resolved interactions for the S2R on the selected states. If the S2R resolution fails for these states (either with a mismatch or a multiple-match result), then it means that either: (1) more states in the graph need to be inspected, hence, the parameter n should increase; (2) there are app states uncovered by the systematic exploration (*i.e.*, not present in the execution model); or (3) the S2R is of low-quality. The parameter n needs to be calibrated per each app. EULER discovers additional app states via random app exploration (see Section 5.2.7).

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

5.2.6 Step Execution and Inference

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

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

5.2.7 Random System Exploration

As mentioned before, if the step resolution fails for all the inspected states, then it means that the systematic app execution approach (see Section 5.2.2) failed to discover app states/screens. To address this issue, Euler performs a random app exploration, starting from the current app screen (shown on the device). The goal is to discover additional app states that could lead to resolving the interaction for a S2R successfully. To do so, Euler identifies the components (different than Layouts and List Views) that have not been executed in the current screen, and randomly selects and executes one clickable component from this set.

The random exploration is performed iteratively y times. At each iteration, x interactions are executed, unless there are no components left to interact with in the current screen. Right after each iteration, Euler updates the graph, the app is restored to the state before the

Table 5.1: Quality annotations for the S2Rs in bug reports

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

random execution, and the S2R matching, execution, and inference are performed again on the new version of the graph. If the S2R is matched against the graph (see Section 5.2.5), then no more iterations are executed. Else, the random exploration process continues.

5.2.8 Quality Report Generation

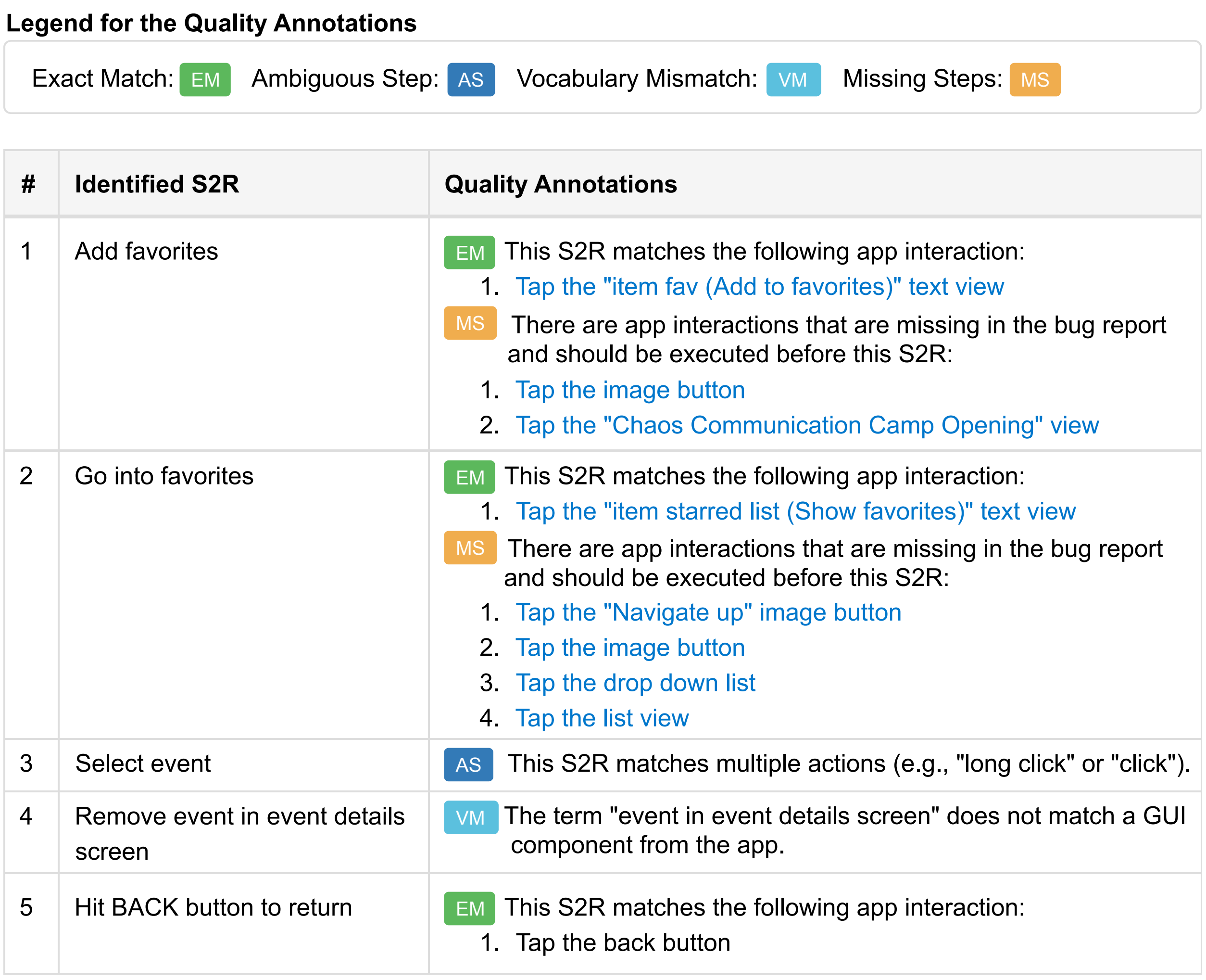
Euler assigns a set of Quality Annotations (QAs) to each S2R. The QAs are defined in Table 5.1. If the S2R is resolved and matched against the execution model successfully, then Euler labels the S2R as High-quality (HQ) - *a.k.a.* Exact Match (EM).

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

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

If a S2R is not resolved in any of the graph states because of a mismatch of the S2R with the application, then it is labeled with Vocabulary Mismatch (VM). In the feedback given to the user, EULER specifies the problematic vocabulary from the S2R constituents (*i.e.*, the *[action]*, *[object]*, *[object2]*, or any combination of these).

EULER generates a web-based Quality Report (QR) with the quality assessment for the S2Rs in a bug report, containing the feedback described above (see Figure 5.3). The user can click on the matched and suggested missing interactions to open a pop-up window showing a screen capture of the app, highlighting the GUI component affected by the interaction.



Legend for the Quality Annotations

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

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

Figure 5.3: EULER's Quality Report for Schedule #154 [42].

5.3 Empirical Evaluation

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

> **RQ$_1$** *What is the accuracy of* Euler *in identifying and assessing the quality of the S2Rs in bug reports?*
>
> **RQ$_2$** *What is the perceived usefulness and quality of the information provided in* Euler*'s quality reports?*

While the answer to RQ$_1$ will inform us on improvements to Euler's accuracy, the one to RQ$_2$ will inform us on the presentation and perceived usefulness of the information included in the QRs.

In order to answer the RQs, we selected a set of bug reports (Section 5.3.1), collected human-produced reproduction scenarios for them (Section 5.3.2), used Euler to identify and assess the quality of each S2R (Section 5.3.3), and asked external evaluators to assess Euler's QRs (Section 5.3.4). Based on the defined evaluation metrics (Section 5.3.5), we analyze the resulting evaluation data and answer the RQs (Section 5.3.6).

5.3.1 Bug Report Sample

We used 24 bug reports from six Android apps [164, 163]: (1) *Aard Dictionary*, a dictionary and Wikipedia reader [22], (2) *Droid Weight*, a body weight tracker [27], (3) *GnuCash*, a finance expense manager [29], *Mileage*, a vehicle mileage tracker [38], *Schedule*, a conference scheduler [41], and (6) *A Time Tracker* [48]. The apps were selected to cover different domains, as well as involve multiple events (*e.g.*, taps, types, swipes, *etc.*) for using their functionality. These apps are also well-studied, having been utilized in several past works on mobile testing and bug reporting [163, 164].

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

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

5.3.2 Ideal Reproduction Scenarios

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

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

5.3.3 Euler Implementation and Calibration

We implemented Euler's S2R identification component by adapting the NCRF++ toolkit [229]. We trained the word embeddings with dimension 200 on 819K bug reports collected from 358 open source projects using the *fastText*'s skip-gram model implementation [71]. We used data our previous work [88] to train the model, using data from GUI-based systems only. The character embedding layer consists of one convolution layer with kernel size of 3. The size of the character vectors is 50, the size of bi-LSTM vectors is 40, and the size of the discourse patterns vector is 154 [88].

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

We implemented the remaining Euler components using the Stanford CoreNLP library [156], our implementation of the discourse patterns [88], and CrashScope [163].

We used the bug reports by Moran *et al.* [164, 163] to test our implementation and calibrate the remaining parameters. In particular, EULER executes 3 random exploration iterations, with 10 steps each. The depth of graph exploration for the step matching is 6 levels from the current program state. These represent the best parameters, according to our tests.

5.3.4 Methodology

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

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

We randomly assigned six bug reports to each participant, for which he/she had to evaluate the QR; each QR is evaluated by three participants. The survey questionnaire, implemented through Qualtrics [40], consists of a demographics section and a section for each of the six QRs to evaluate. In the demographics section we ask questions about years of experience on (i) non-mobile app development, (ii) mobile app development, (iii) Android app development in particular, and (iv) use of Android phone. We also ask approximately how many bug reports the participant has ever reported.

For each QR, the questionnaire contains two sections. The first section contains, for each S2R, three questions, for answering RQ$_1$:

1. A yes/no question for checking whether EULER correctly identified the S2R (in case of a negative answer, questions (2) and (3) are skipped).

2. For each annotation produced by EULER for a given S2R, an agree/disagree question aimed at checking its correctness. In case the answer was negative, the respondents were instructed to explain their answer in a free-text form.

3. In case of missing steps, a third (four check-box) question is formulated for assessing whether EULER's suggested list of missing steps is: (i) correct; (ii) contains extra steps; (iii) is lacking one or more steps; or (iv) some steps are incorrectly ordered. We ask the respondents to use a free-text form to provide an explanation for their answer.

The second section of the survey addresses RQ$_2$, by asking:

1. Whether EULER's quality report is easy to read and understand (using a 5-level Likert scale [170]).

2. Whether the quality report is likely to help users to better write bug reports (using a 5-level Likert scale).

3. Four free-text questions to indicate what information was perceived useful, useless, and what information should be added to or dropped from the QR.

5.3.5 Metrics

For addressing RQ$_1$, we measure EULER's precision and recall at identifying the S2Rs from the bug report by comparing EULER's output with the ideal reproduction scenario (see Section 5.3.2). We also measure the proportion of correctly identified S2Rs judged by the participants. Since we involve three participants for each bug report, we consider the correctness assessment provided by the majority.

Regarding the QAs for each step, we compute, for each QA type (see Table 5.1), the proportion of annotations judged as correct. We consider the assessment of the majority of participants requiring at least two positive answers. Note that, in this case, a respondent might not have answered question #2 if she judged the S2R as incorrectly identified.

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

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

Regarding the free-text questions related to the usability/quality of the QRs information, we categorized the responses using a card-sorting approach [207] and analyzed each category.

5.3.6 Results and Analysis

Table 5.2 summarizes the evaluation results of Euler's quality assessment and feedback. It reports the number of S2Rs (identified by Euler) for each bug report (2nd column - # S2Rs), the (correct/total) number of quality annotations for all S2Rs (3rd column - # QAs), the (correct/total) number of annotations across the quality categories from Table 5.1 (4th-9th columns), and the number of MS annotations for which there are unreported and extra steps in the list of suggested missing steps (7th and 8th: # MS - Not Reported and Extra, respectively).

The replication package [81] contains evaluation data and additional results that enable the replication of the evaluation. The package includes bug reports, ideal reproduction scenarios, identified S2Rs in the reports, Euler's quality reports, study survey, Euler's calibration data, and detailed results.

Table 5.2: Accuracy results for EULER's Quality Annotations (QAs).

Bug Report	**# S2Rs**	**# QAs**	**# AS**	**# HQ**	**# MS**			**# VM**
		C/T	**C/T**	**C/T**	**C/T**	**NR**	**E**	**C/T**
Aard Dictionary #81	5	5/7	-	4/4	1/3	0	3	-
Aard Dictionary #104	1	0/1	0/1	-	-	-	-	-
A Time Tracker #1	4	4/4	-	1/1	1/1	0	1	2/2
A Time Tracker #10	2	3/3	-	2/2	1/1	1	1	-
A Time Tracker #25	7	9/9	-	7/7	2/2	0	2	-
A Time Tracker #35	2	1/3	-	0/2	1/1	1	1	-
A Time Tracker #46	7	6/10	-	5/6	0/3	0	3	1/1
Droid Weight #12	4	4/4	-	4/4	-	-	-	-
Droid Weight #25	2	3/3	-	1/1	1/1	1	1	1/1
GnuCash #471	5	5/5	-	2/2	1/1	0	1	2/2
GnuCash #615	2	3/3	-	2/2	1/1	1	1	-
GnuCash #616	4	4/5	-	3/3	1/1	0	1	0/1
GnuCash #618	3	3/4	-	2/2	0/1	0	0	1/1
GnuCash #620	3	3/5	0/1	1/2	2/2	1	1	-
GnuCash #633	2	3/3	-	2/2	1/1	0	0	-
GnuCash #663	2	4/4	-	2/2	2/2	1	0	-
GnuCash #699	6	5/11	-	1/5	3/5	2	5	1/1
GnuCash #701	8	4/13	1/1	1/6	1/5	0	4	1/1
Mileage #49	4	4/6	-	2/2	1/2	0	2	1/2
Mileage #53	2	3/3	1/1	1/1	1/1	0	1	-
Mileage #64	2	1/2	1/1	-	-	-	-	0/1
Mileage #65	4	4/6	-	1/3	2/2	0	2	1/1
Schedule #154	5	7/7	1/1	3/3	2/2	0	2	1/1
Schedule#169	3	3/3	-	2/3	1/1	0	1	-
Total	**89**	**91/124**	**4/6**	**49/64**	**26/39**	**8**	**33**	**12/15**
%	**-**	**73%**	**67%**	**77%**	**67%**	**21%**	**85%**	**80%**

The results are computed across the categories from Table 5.1. C=Correct, T=Total, NR=Not Reported, E=Extra.

S2R Identification Results. EULER identified 89 S2Rs in the 24 bug reports (see Table 5.2). Only four S2Rs were judged as incorrect, resulting in 96% overall precision. More specifically, the precision is 100% for 20 bug reports, with the exception of four: Aard Dict. #81 (80%) [24], A Time Tracker #1 (75%) [49], GnuCash #471 (80%) [30], and Schedule #169 (67%) [43]. In 73 out of 89 (*i.e.*, 88%) answers there is a perfect consensus among the three evaluators across bug reports. We also found that two S2Rs were not identified by EULER (*i.e.*, 98% recall).

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

Quality Assessment Results. Table 5.2 shows that (overall) 73% of the provided QAs were considered correct by the evaluators, with a percentage ranging between 67% and 80% of correct MS and VM annotations, respectively. The participants reached a perfect consensus in 56% of the cases. For 12 bug reports, EULER achieves 100% accuracy. For the remaining 12 reports, EULER's accuracy ranged from 0% to 80%. We determined the causes of such performance by manually analyzing the participants' answers and EULER's algorithm for those 12 cases, across the QA types.

For two bug reports (*i.e.*, Aard Dict. #104 [23] and GnuCash #620 [32]), EULER incorrectly produced two AS annotations (*i.e.*, for two S2Rs). According to the participants' explanations of their judgment, we found that the annotations were confusing to them, specifically, it was not clear which components EULER's feedback was referring to. For instance, for the Aard Dict. #104's only S2R: "*Tap link to another Wikipedia article*", EULER produced the AS annotation: "*This S2R matches multiple GUI components (e.g., the "1st Link" and "2st Link " views)*". In this case, EULER reached a Wikipedia webpage with multiple links having the labels shown in the annotation. This webpage was unknown to the participants (as it was not shown in the video), hence they did not understand the suggested matched components. In addition, we found that the AS annotation produced for the GnuCash #620's 1st S2R: "*Set the color of an account*" did not suggest the correct GUI component for this S2R (the color picker in the "creating/editing accounts" screens). The cause for such a mismatch lies in the priority that Euler gives to resolved interactions from program states closer to the current one. One possible improvement is to weight in the similarity score obtained by the resolved components across multiple program states, in such a way that candidates with higher similarity in screens further away from the current one are more likely to be suggested.

In six bug reports, Euler incorrectly assessed the quality of 15 S2Rs as High quality (HQ), which means that the interactions matched/suggested by Euler do not correspond to the S2Rs. We manually analyzed these cases, and found four main reasons: (1) the similarity threshold defined in Section 5.2.4 (*i.e.*, 0.5) is too restrictive for some reports; (2) the similarity used to resolve an S2R to a screen (*i.e.*, Formula 5.1) does not account for small term differences between the S2R (*i.e.*, the query) and the GUI components; (3) the synonyms for some terms, used to reformulate the query, may incorrectly boost the similarity score of unexpected GUI components; and (4) the quality of screen information for some applications is low.

We illustrate the first three problems with the report A Time Tracker #35 [51]. The first S2R for this report was identified as "*Restore backup*" and the expected component for that S2R is the menu option "*Restore from backup*", whose similarity to the S2R is 0.4 (the LCS is 'restore' and the average size of both strings is 2.5 - see Formula 5.1). Because the similarity is lower than the threshold, the component is not returned as a candidate. Next, using the predefined query synonyms, Euler reformulates the query by expanding the S2R to "*Restore back up*" which returns the menu option "*Back up to SD card*", whose similarity to the query is 0.54. In this case, the synonym for backup, “back up”, boosted the similarity of the menu option, which was returned as most similar to the S2R. To address these problems, we plan to improve Euler’s similarity formula for cases with little term variations, by utilizing information related to shared term frequency and how many terms are in-between the shared terms. To illustrate the fourth problem, consider the case of the 5th S2R from GnuCash #701 [33]: “*Click ‘save’*”. The incorrect matched component for this S2R was the button “Delete” from the ”delete account” screen. The component ID given by GnuCash developers was “btn save”, which matches the query. In this case, the mismatch could be used as feedback for developers about problems with the app screens information. Studying the impact of low-quality app information on Euler is subject of future work.

The three S2Rs (from three bug reports), for which Euler incorrectly detected a vocabulary mismatch (VM), involve more than one interaction. For instance, for the 2nd S2R from GnuCash #616 [31]: “*Select export to ‘Google Drive’*”, Euler failed to match “Google Drive” because of uncovered app states and imprecise S2R parsing/matching.

Analysis of MS annotations. We report and analyze the results for the missing step (MS) annotations, which include the list of steps inferred by Euler.

The participants reached a perfect consensus in 53% of the cases with MS annotations. For six bug reports, Euler incorrectly flagged 13 S2Rs as having missing steps (*i.e.*, they were assigned an MS annotation). For the remaining 26 S2Rs (*i.e.*, 66.7%), from 16 bug reports, the MS annotation was correct (*i.e.*, indeed there are missing steps). For the 13 S2Rs with incorrect MS annotations, all the MSs suggested by Euler are unnecessary for bug reproduction. For the 26 S2Rs with correct MS annotation, Euler suggested extra MSs for 20 of them (*i.e.*, 77%), according to the external evaluators. This means that 33 S2Rs, in total, were judged to have extra missing steps (see the 8th column of Table 5.2), which represents 85% of the cases. In addition, for 8 S2Rs total, the list of missing steps lacks additional steps (*i.e.*, not detected by EULER), which represents only 21%. In all MS annotations, the order of the suggested MSs is correct, meaning that EULER suggests feasible execution paths. However, in all cases, the suggested MSs lack some or have extra steps.

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

We analyzed the 13 cases (from 6 bug reports) for which EULER incorrectly indicated missing steps, and from the correct MS cases, the 20 cases with extra steps. Our analysis reveals two main reasons for such cases, namely, excessive application exploration, and imprecise S2R resolution/matching. Regarding the first limitation, we found that the systematic and random exploration strategies execute more interactions than needed. While this is done by design, trying to uncover as many program states/screens as possible, it leads to excessive inferred steps. Regarding the second problem, any mismatch in the first S2Rs from a bug report can divert EULER's execution, thus producing even more mismatches or no matching at all for the remaining S2Rs. In the latter case, the random exploration takes place, thus producing unnecessary inferred steps. We found that the reason for such mismatches comes from the inability of the similarity scoring formula (*i.e.*, Formula 5.1) to match the query with single-term text sequences (from the components), and also, from the fact that in some cases, the random exploration is executed late (after the first S2R matching

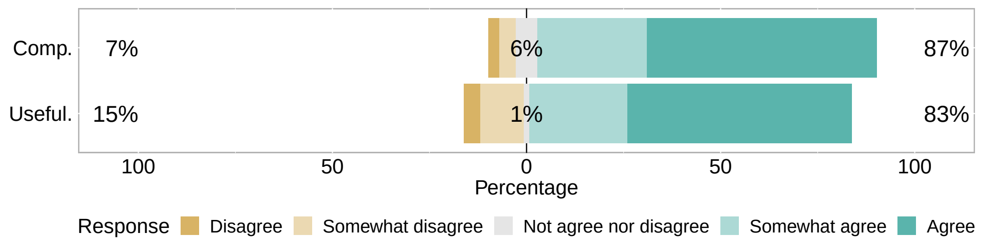


Figure 5.4: Perceived Comprehensibility and Usefulness of the Quality Reports.

fails) and unexpected components, with similar vocabulary to the S2R, are returned. Improving the (upfront) systematic application exploration to uncover as many program states as possible may help to alleviate this problem.

Finally, we manually analyzed 10 bug reports for which EULER obtained the lowest recall (within the [33% - 78%] range) when inferring the expected MSs. The main reasons for these cases include: (1) incorrect detection of the S2Rs' order from the bug report, and (2) failing to handle special S2Rs. We illustrate the first issue with A Time Tracker #10 [50]. Specifically, the S2R sentence "*If I press the Back button while viewing the Preferences*" implies the S2Rs "view preferences" and "press back button" executed in that order. EULER failed to identify the correct order in this case, provoking to not execute one of the MSs: "*click OK button*". One exemplar of the second problem is repetitive S2Rs (*e.g.*, "*enter few fill ups*" from Mileage #53 [39]), which in its current version, EULER does not support.

Perceived Usefulness. Figure 5.4 shows an asymmetric stacked bar chart depicting the perceived comprehensibility and usefulness of EULER's quality reports. The figure shows positive results. In particular, the study participants agree and somewhat agree that:

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

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

- Explicit, clear, and fine-grained S2R feedback. One participant mentions that the matched/suggested "*S2Rs are pretty descriptive and would guide the user to complete better the bug description*". Another participant supports that comment, stating that the suggested S2Rs "*are a good example of how to write steps-to-reproduce*". Another person acknowledges that "*Developers/maintainers would find this tool *very* useful for their debugging process*". Finally, one participant mentions that EULER "*provides detailed feedback for every single step, not just the overall feedback on all steps*"

- Feedback about incorrect S2R vocabulary. For example, one participant indicated that the tool correctly "*flags words such as 'find' and 'fix' that do not directly translate to an app action*". Another person notes that EULER detects cases that "*the user could improve including some more details in his error report*".

- Feedback about missing steps. For instance, one participant mentions that the suggested missing steps "*can guide the user to list them better*" in the report. Another person mentions that "*They help avoid the guessing part when reproducing the bug*".

- Screenshots for the matched interactions. Some evaluators considered the screenshots helpful for "*identifying the right scenario for reproducing the bug*" and they were found to "*complement the description of each suggested missing step*".

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

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

- Some feedback is unclear. For example, one participant mentions that "*Some missing steps have strange names*". We confirmed that some AS annotations are confusing and found that the suggested missing steps may be give "*little information so the user always needs to click on them and see the image to fully understand the nature of the step*". EULER phrases the suggested/matched S2Rs based on the internal information from the application (*i.e.*, component label, description, or ID). In some cases, this information is not available or may be only clear for the developer (*i.e.*, using "btn save" instead "Save button").

- Setup application steps are not needed. One participant commented that "*some steps that describe the app initial configuration... are not needed to reproduce the bug*". Another participant says "*they are irrelevant for the bug*".

5.3.7 Threats to Validity

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

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

Euler's calibration impacts the *internal validity* of our conclusions. As explained in Section 5.3.3, we used different bug reports (to the ones used in the evaluation) to find the best parameters of our approach. Another threat is subjectivity and error-proneness of the human-based evaluation. To mitigate this threat, we relied on three evaluators per bug report, and decided upon majority.

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

5.4 Related Work

Bug Report Quality Assessment. Zimmermann *et al.* [249] conducted a survey exploring the most useful information in bug reports and proposed a supervised approach to predict the overall quality level of a bug report (*i.e.*, bad, neutral, or good). This approach relies on features, such as readability, presence of certain keywords, code snippets, *etc.* Dit *et al.* [104] and Linstead *et al.* [148] measured the semantic coherence in bug report discussions based on textual similarity and topic models. Hooimeijer *et al.* [123] measured quality properties of bug reports (*e.g.*, readability) to predict when a report would be triaged. Zanetti *et al.* [235] identified valid bug reports, as opposed to duplicate, invalid, or incomplete, by relying on reporters' collaboration information. To enhance bug reports, Moran *et al.* focused on augmenting S2Rs via screenshots and GUI-component images [164]. Zhang *et al.* [238] enriched new bug reports with textually similar sentences from past reports.

Test Case Generation and Bug Reproduction from Bug Reports. Fazzini *et al.* [109] and Karagöz *et al.* [131] proposed approaches to generate executable test cases from bug reports. Zhao *et al.* [242] proposed a technique to reproduce crashes from bug reports. Different from these approaches, EULER is capable of automatically identifying S2Rs in free-form bug report text, and inferring missing steps in the report. EULER is complementary to these techniques, as it is aimed at improving the quality of reported S2Rs. High-quality S2Rs can help improve the effectiveness of these approaches.

5.5 Conclusions

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

Future extrinsic studies will confirm the reported perceived usefulness. Before such studies, improvements to EULER's accuracy and to the information included in the QRs are planned, based on the feedback obtained from the evaluators. Specifically, we plan to improve the quality of EULER's QR, with: (i) more complete application step sequences; and (ii) additional screenshots to help guide the reporters. We also plan to tune EULER's matching algorithm to account for minor variations between text sequences and matches in different parts of the execution model. As discussed in Section 5.3.6, optimizations to the application exploration strategies are also planned.

5.6 Acknowledgments

This chapter was accepted for publication at the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19). This work [80] is a collaborative effort with Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk from the College of William and Mary; Jing Lu and Dr. Vincent Ng from The University of Texas at Dallas; and Dr. Massimiliano Di Penta from the University of Sannio.

Oscar Chaparro, Carlos Bernal-Cárdenas, Jing Lu, and Kevin Moran contributed to the design, implementation, and evaluation of the proposed approach. Massimiliano Di Penta contributed to the evaluation design and execution. Denys Poshyvanyk and Vincent Ng provided feedback to the approach design and to the empirical evaluation.

CHAPTER 6
USING BUG DESCRIPTIONS TO REFORMULATE QUERIES DURING BUG LOCALIZATION[1]

6.1 Introduction

Text Retrieval (TR) has been widely used by researchers to support developers during bug localization in source code (see Section 2.5 for details). TR-based bug localization (TRBL) techniques address bug localization as a document retrieval problem where an initial query, formulated from the information provided in a bug report, is used to retrieve a ranked list of candidate code artifacts (*a.k.a.* code documents, such as files, classes, or methods) that are likely to contain the reported bug. In the TRBL process (see Section 2.5), once the TRBL technique produces the list of candidates, the developer proceeds to check the top-N (say, top-5) candidates, one at a time, and determine whether or not they contain the bug. The developer performs this process by inspecting each candidate's name as well as its internal code. Deciding how relevant each candidate is (*i.e.*, whether it is buggy or not) with respect to the bug report, is determined largely on the developer's knowledge of the system [158]. Once one buggy code document is found, the process ends successfully (*i.e.*, the query is *successful*). If the top-N candidates are not deemed buggy, the developer has three main options: (1) inspect additional candidates (say, N more) in the result list; (2) reformulate the initial query, run the reformulated query with the TRBL technique at hand, and inspect the returned candidate code documents; or (3) switch to other strategies for localizing the buggy code such as, navigating program dependencies.

Traditionally, TRBL approaches use the full textual information from bug reports (*i.e.*, bug descriptions) as queries. In many cases, the queries fail to return the buggy code arti-

[1]© 2019 Springer. Reprinted with permission from Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus, 'Using Bug Descriptions to Reformulate Queries during Text-Retrieval-based Bug Localization", Empirical Software Engineering, 2019.

facts within the top-N results (*i.e.*, the queries are *unsuccessful*) and the developer requires following any of the options described above for locating the buggy code. Exiting research on TRBL has focused on improving the ranking produced by the initial query, primarily by combining various types of software information, such as code dependencies, execution traces from the bug report, past bug reports and code changes, *etc.* (see Section 2.5 for details). Other research has focused on techniques to reformulate the initial query, mostly by leveraging relevance feedback from the user [110], pseudo-relevance feedback based on previous search results [116], or additional information to replace or expand the query (*e.g.*, adding synonyms) [200, 179, 158]. However, in many cases, the bug description contains terms that are irrelevant for code retrieval, that is, they act as noise and result in the retrieval of irrelevant (*i.e.*, non-buggy) code artifacts. This is because reporters do not write bug descriptions as queries for a text retrieval engine, instead, they write them to communicate the software problem to the developers. Previous research [90, 162] showed that removing the irrelevant terms from the queries (*i.e.*, *query reduction*) leads to substantial improvement in code retrieval. Unfortunately, little research has focused on identifying parts of bug descriptions that contain irrelevant terms with respect to code retrieval [179, 181, 182, 116, 90, 83].

We propose and investigate the effectiveness of several query reduction strategies, based on the structure of bug descriptions, which are easy to apply when using TRBL approaches. Such reformulation techniques can be used when the initial query does not retrieve the relevant code artifacts within the top-N results and the users choose to investigate more retrieved documents after reformulation. Typically, bug reports are composed of three major parts: the title, the description, and bug meta information. The title is a summary of the software problem, the description is a detailed account of the problem, and the meta information includes other data about the bug such as software version affected, operating system, bug severity, *etc.* The description may contain technical information such as code snippets (*i.e.*, CODE) or stack traces. More importantly, the description contains the user's account of the software (mis)behavior (*i.e.*, the Observed Behavior or OB), the steps to trigger the (mis)behavior (*i.e.*, the Steps to Reproduce or S2Rs), and the expected software behavior (*i.e.*, EB) [249, 98, 83, 88]. We hypothesize that the TITLE of the bug reports, the OB, EB, and S2Rs, as well as any CODE present in the bug description, contain the most relevant information with respect to TRBL. Conversely, we argue that other parts of the *bug descriptions* contain more irrelevant terms, which lead to false positives (*i.e.*, the retrieval of non-buggy code artifacts). We propose leveraging these parts of the bug description for query reformulation.

We introduce and empirically evaluates 31 different reformulation strategies, which reduce the initial query to parts that correspond to the TITLE, OB, EB, S2Rs, and/or CODE of the bug report. The reformulation strategies are independent of any TRBL approach and were used with five different techniques, namely Lucene [117], Lobster [167], BugLocator [245], BRTracer [225], and Locus [224], which retrieve code artifacts at different code granularities (*i.e.*, file, class, and method). Using existing TRBL datasets [245, 225, 167, 161, 83, 142], we randomly sampled a set of 1,221 queries that fail to retrieve the buggy code artifacts within the top-N results when using the TRBL techniques. We compared the performance achieved by the five TRBL approaches at three code granularities when using the complete bug reports as queries versus a reduced version produced by each reformulation strategy.

We envision a straightforward usage scenario for reformulation, where developers use the entire content of a bug report as initial query (*i.e.*, both title/summary and description), and optionally other information leveraged by the used TRBL technique, which is the typical TRBL scenario [105]. If none of the buggy code documents are found in the top-N candidates (by inspecting each candidate's name and/or source code), then the developer selects the TITLE, OB, and S2Rs/EB (if present) from the bug report and uses their combination as the new query, hoping to locate the buggy code artifacts. This reformulation approach is independent of the underlying TRBL technique and does not depend on the returned results or any information from other bug reports and external sources. In other words, it is easy to use by any potential user and should work with any existing TRBL technique based on bug descriptions.

We provide an online replication package [84] for our empirical study. The package includes code corpora, initial and reformulated queries, gold sets (*i.e.*, buggy code documents for each query), and additional material that enable the reproducibility of our study. The package also contains details and additional empirical results of our evaluation, which are not included in this dissertation.

6.2 Query Reformulation Strategies

We propose a user-driven query reformulation approach based on the structure of bug descriptions, with a two-step scenario for bug localization in mind. In the first step, the developer issues an initial query (manually or automatically) from the full text of the bug report and inspects the top-N code candidates returned by the TRBL technique at hand. Sophisticated TRBL techniques may require extra information to the bug report such as execution traces or past bug report data. In this case, the developer collects and provides such information to the TRBL technique, either manually or automatically. If any of the returned candidates is deemed buggy (*i.e.*, the query is successful), the bug localization process ends and the developer proceeds to fix the bug. Conversely, if none of the candidates are buggy (*i.e.*, the query is unsuccessful), the developer reformulates the initial query in the second step (via the proposed reformulation strategies - see below), runs it with the TRBL approach, and investigates additional N retrieved code artifacts. The N results retrieved in the second step should not include the N results returned by the initial query, as they were deemed non-buggy. If a buggy code artifact is found within the new result list (*i.e.*, the reformulated query is successful), then the bug localization process ends and the developer proceeds to fix the bug. Otherwise (*i.e.*, the reformulated query is unsuccessful), the developer may refine the query or switch to other methods for localizing the buggy code (*e.g.*, navigating code dependencies).

We contend that the following five parts of a bug description contain the most relevant terms for locating the buggy code:

- The bug title (*a.k.a.* **TITLE**): it is the summary of the software problem found by the user. Our assumption is that users carefully write the titles to include the most relevant terms. The title is found in all bug reports, hence it can be easily used for query reformulation. Some existing approaches [220, 195, 221, 233, 55] treat the bug title as an individual field, but unlike our reformulation approach, they use it along with the full description.

- Observed behavior (*a.k.a.* **OB**): it describes the software (mis)behavior observed by the user, which is typically deemed to be incorrect or unexpected. Our prior work [83] found that the OB contains relevant information that helps locate the buggy code, more than other parts of the bug description.

- Expected behavior (*a.k.a.* **EB**): it describes the normal or regular software behavior expected by the user. As the EB describes the opposite of the OB, we hypothesize that it contains relevant terms with respect to code retrieval.

- Steps to reproduce (*a.k.a.* **S2Rs**): it describes the steps that the user followed to trigger the OB. The S2Rs may contain terms that point to software features, hence it is also a good candidate for query reformulation.

- Code snippets (*a.k.a.* **CODE**): in many cases, especially for software libraries or frameworks, users provide code snippets that help developers better understand and reproduce the software problem. Code snippets are likely to reference places in the source code related to the bug.

Bug report title:
[code assist] the caret position is wrong after code assist [TITLE]

Bug report description:
Using 20030330-0500, I got this weird behavior.

```
import java.awt.Frame;
import java.awt.event.WindowAdapter;

public class Foo extends Frame {

  public void bar() {
    addWindow<CODE ASSIST HERE>Listener(new WindowAdapter());
  }
}
```
[CODE]

Select addWindowListener in the list of proposal. [S2Rs]

The result is:
addWindowListene<POSITION OF THE CARET>rListener [OB]

I would expect:
addWindowListener<POSITION OF THE CARET>Listener [EB]

This is pretty annoying and seems to occur only for method name proposal.

Legend: TITLE CODE OB EB S2Rs

The highlighted text corresponds to the title (TITLE), code snippets (CODE), observed behavior (OB), expected behavior (EB), and steps to reproduce (S2Rs).

Bug report found at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=89621`

Figure 6.1: Bug report #89621 from Eclipse.

Figure 6.1 shows an example of a bug report containing each one of these parts.

We propose 31 different reformulation strategies based on the combination of the five types of information described above: TITLE, OB, EB, S2Rs, and CODE. We denote their combination by using a plus sign (+) between them. For example, the strategy using OB and CODE is denoted as OB+CODE, and the strategy using EB, S2Rs, and TITLE is denoted as EB+S2R+TITLE. When using such a reformulation strategy, the user simply needs to select and concatenate the parts of the text corresponding to the types of information used by the strategy from the title and bug description, and remove the rest of the textual description. It is important to note that the strategy only applies if the bug contains *all* the types of information. As an example, reformulating the (initial) query from the bug report shown in Figure 6.1, by using the OB+TITLE strategy, will result in the following query: "*[code assist] the caret position is wrong after code assist The result is: addWindowListene< POSITION OF THE CARET>rListener*".

6.3 Empirical Evaluation Design

We conducted an empirical evaluation to assess the effectiveness of the proposed reformulation strategies. The evaluation aims at answering the following research question:

> ***RQ***: *Which query reformulation strategies help TRBL approaches retrieve more buggy documents within the top-N results when compared to the case in which query reformulation is not used?*

This section describes the procedure we followed to answer our research question, while Section 6.4 discusses the evaluation results. We used five TRBL techniques (Sections 6.3.1 and 6.3.2) to locate the buggy code artifacts for a large set of queries/bug reports (Section 6.3.3). Then, for a subset of the queries for which the tools failed to retrieve the buggy code artifacts within the top-N results (Section 6.3.4), we manually identified the structure of the corresponding bug descriptions (Section 6.3.5). We used the 31 strategies to reformulate the queries (based on the identified structure) and compared how many more buggy code artifacts are retrieved among the next-N candidates with and without reformulation (Sections 6.3.6 and 6.3.7).

6.3.1 TRBL Techniques

We used five TRBL techniques to perform our empirical evaluation on both initial and reformulated queries, namely Lucene [117], Lobster [167], BugLocator [245], BRTracer [225], and Locus [224]. We were interested in finding out whether the reformulation strategies are equally effective on different TBRL techniques.

Lucene [117] is a retrieval technique implemented in the open source library of the same name [14]. Lucene combines the standard information retrieval Boolean model and the Vector Space Model (VSM), based on the TF-IDF representation [197], to compute the similarity between a bug report (*i.e.*, the query) and a code document (*e.g.*, a file, class, or method). Lucene relies only on textual information to retrieve the relevant (buggy) documents, independently of the code granularity. Typically, a Lucene query is created by concatenating the bug report's title and description, including any information embedded in these sources (*e.g.*, code snippets).

The remaining four techniques also rely on textual similarity to rank the buggy code artifacts. However, they include additional information to boost the similarity score of the documents.

Lobster [167] is a TRBL technique that leverages stack traces found in bug reports. It boosts the classes that appear in these traces and also their related classes by using the system's call graph. Lobster works at class-level granularity and only makes a difference on bug reports that contain stack traces.

BugLocator [245] is a TRBL approach that combines information from bug fix history and file length to boost certain corpus documents. This approach uses a record of previously-fixed bug reports to boost the corresponding fixed files, according to the textual similarity of these reports to the query. Additionally, it boosts all corpus source files based on their length (*i.e.*, number of terms). BugLocator works at file-level granularity.

BRTracer [225] is an extension of BugLocator, which uses stack trace information from bug reports and source file segmentation to boost source code files retrieved by BugLocator. Similar to Lobster, this technique boosts the source code files that appear in the traces, and other files (or classes) that are used in their corresponding source code. In addition, the files are segmented into smaller documents, and the highest textual similarity between the segments and the query is used as the similarity of the whole file.

Locus [224] is a TRBL technique that leverages textual and additional information from past code changes to identify the buggy code documents for a bug report/query. This technique segments source code files into code hunks: small code segments, resulting from code changes throughout the project history. This means that this technique is able to retrieve code hunks and also entire source files. Locus utilizes textual similarity between bug reports and code hunks (including their corresponding commit messages), using two corpus extraction strategies, one that uses the whole textual content, and one that uses only the code entities referenced in the text (*i.e.*, package names, classes, and methods). The approach also increases the suspiciousness degree of a source file based on how many times the file was changed, and boosts the score of a hunk based on how recent it was applied in the code with respect to the current bug report.

6.3.2 TRBL Technique Implementation

In our evaluation, we used Apache Lucene v5.3.0 with the default similarity measure and parameters, and the original implementation of Lobster, provided by its authors [167]. We used the implementation of Locus provided in Bench4BL [142]. However, we did not use Bench4BL's scripts to execute Locus' implementation because we identified two issues.

The first issue relates to the corpus preprocessing. As the bug reports in Bench4BL are stored in XML files, some characters are escaped into their corresponding *character entity reference* (*e.g.*, `'&'` or `'<'` would be escaped to `'&amp;'` or `'&lt;'`, respectively). However, when running Locus using the Bench4BL's scripts, such characters are not unescaped correctly. The queries would contain the text corresponding to the entity references (*i.e.*, `'amp'` or `'lt'`), even after they are preprocessed (*e.g.*, using special-character removal). We confirmed that this issue leads to different TRBL results, compared to correct unescaping, which change the set of unsuccessful queries that require reformulation (see Section 6.3.4 for more details). Therefore, we implemented our own scripts for running Locus.

The second issue stems from Locus calling the Git executable as a subprocess to retrieve and read the project's commit history via the subprocess' *standard streams*. Note that Locus is implemented in Java. Due to Java's handling of subprocesses, reading the subprocess' output has to be done in separate threads – see `ExecCommand.exec(String command, String[] envp, String workpath)` in Locus' original implementation[2]. Reading the standard output on a separate thread would cause the read data (*i.e.*, the commits log) to be randomly truncated. This would cause the tool to behave unpredictably, sometimes producing different results on different runs with the same data, and sometimes crashing during the execution. We fixed the bug by reading the *standard output* on the main thread while letting the *standard error* to be read on a different thread, as this output was being ignored by the original implementation anyway. The replication package [84] contains the fix to Locus' issue, as well as the code we used to run our experiments.

We used our own implementation of BugLocator [245] and BRTracer [225], based on the description provided in their corresponding publications. We obtained the implementations of BugLocator and BRTracer made available by their authors, along with their experimental data [245, 225], and attempted to replicate the results of the empirical studies reported in each paper. However, for BugLocator, we could only replicate the results for two of the four systems: Eclipse and SWT [225]. The tool failed to complete the evaluation on the AspectJ system and it was not possible to acquire the source code for the ZXing system, because it

[2] `https://tinyurl.com/ybye2zhc`

was not provided by the authors and the corresponding system version is no longer available online. The results on Eclipse and SWT matched those reported in the paper.

Since we could not completely replicate the experimental results, we decided to implement our own version of the tool. Our implementation also failed to exactly replicate the experimental results reported in the mentioned study [245]. Since the source code for the authors' implementation was not available at the time, we examined the bytecode of the original implementation and compared it with our own code. We discovered two key differences between the approach from their paper and the implementation provided by its authors:

1. The paper proposes a normalization function for a source code file length x as part of its rVSM model, which is defined as:

   $$N(x) = \frac{x - x_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}$$

   However, after examining the bytecode of the original implementation, we found that it is actually being computed in the following manner:

   $$N(x) = \begin{cases} 0.5 & \text{if } x < b_l \\ 1 & \text{if } x > b_u \\ \dfrac{x - b_l}{b_u - b_l} & \text{otherwise} \end{cases}$$

   Where b_l and b_u correspond to a lower and upper bound, respectively, and are calculated as follows:

   $$b_l = \begin{cases} \mu - 3\sigma & \text{if } \mu - 3\sigma \geq 0 \\ 0 & \text{if } \mu - 3\sigma < 0 \end{cases}$$

   $$b_u = \mu + 3\sigma$$

   With μ and σ being the average length and standard deviation of document lengths in the code corpus, respectively.

Table 6.1: Performance differences of BugLocator implementations.

System	**MRR**			**MAP**		
	Original impl.	**Our impl.**	**Diff.**	**Original impl.**	**Our impl.**	**Diff.**
AspectJ v1.5.3	41%	35%	-6%	22%	18%	-4%
Eclipse v3.1	41%	25%	-16%	30%	19%	-11%
SWT v3.1	53%	33%	-20%	45%	30%	-15%

Our impl: our implementation of BugLocator.

Original impl: the original implementation provided by its authors [245].

2. As part of the rVSM model, the authors propose a length score for a source code file, defined in the paper as:

$$g(\#\text{terms}) = \frac{1}{1 + e^{-N(\#\text{terms})}}$$

However, in their implementation, what is being computed is:

$$g(\#\text{terms}) = \frac{e^{6N(\#\text{terms})}}{1 + e^{6N(\#\text{terms})}}$$

We decided to test these findings by modifying our implementation to imitate the original implementation. After testing it on the Eclipse system with the same preprocessing used by the original tool, we were able to replicate the results reported in the paper, with minor differences. We considered that our implementation is fit for experimentation. The differences between the performance of our implementation and the performance reported by the authors are presented in Table 6.1. The difference cast a threat to the validity of our findings for Buglocator, which we may address in the future.

A similar situation happened when replicating the results of the empirical study on the BRTracer tool [225]. We replicated the results reported in the paper using the tool provided by the authors, but we decided to use our own version to facilitate our process. After re-implementing the tool on top of our version of BugLocator, we found discrepancies when trying to replicate the study results. Since the source code of the authors' version is readily available, we compared it with the description of the approach in the paper.

This is a non-exhaustive list of findings:

1. The bugs for the evaluation are sorted by resolution date, however, their submission date is not checked. The paper states that for a bug report to be considered "previously fixed" for the current bug being located, both the submission and resolution dates must happen before the current bug's submission date. However, in the way that the tool is implemented, some bugs are considered previously fixed when their submission date happens after the current bug's submission date. This results in an unrealistic evaluation, *i.e.*, future bugs are used as previously fixed bugs.

2. The calculation of the SimiScore (the score boost of files according to previously fixed bugs) is done against file segments, instead of whole files, as it is explained in the paper.

3. The BoostScore (the score boost to files found in stack traces or related to these files) is formulated in the paper as:

$$BoostScore(x) = \begin{cases} \dfrac{1}{\text{rank}} & \text{if } x \in D \text{ and rank} \leq 10 \\ 0.1 & \text{if } x \in D \text{ and rank} > 10 \\ 0.1 & \text{if } x \in C \\ 0 & \text{otherwise} \end{cases}$$

   Where D is the set of source files appearing in stack traces in the bug report, and C is the set of files used in import statements by the files in D. However, the implementation assigns a constant value of 0.5 for files in D and 0.2 for files in C.

We decided to use our implementation, which works as described in the paper. Table 6.2 contains a comparison between the results reported by the authors and the ones achieved by our implementation. It can be observed that, even though the underlying BugLocator implementation achieves somewhat different results compared to the original, our implementation of BRTracer achieves nearly identical results to ones provided by the authors.

Finally, we executed BugLocator and BRTRacer with the parameter $\alpha = 0.2$. This value leads to the best TRBL performance, according to the respective evaluation results [225, 224].

Table 6.2: Performance differences of BRTracer implementations.

System	**MRR**			**MAP**		
	Original impl.	**Our impl.**	**Diff.**	**Original impl.**	**Our impl.**	**Diff.**
AspectJ v1.5.3	49%	53%	4%	29%	30%	2%
Eclipse v3.1	43%	40%	-3%	33%	31%	-2%
SWT v3.1	60%	64%	5%	53%	56%	3%

Our impl: our implementation of BRTracer.

Original impl: the original implementation provided by its authors [225].

6.3.3 TRBL Data

We compiled three TRBL data sets from existing TRBL data (see Table 6.3). Each data set contains corpora at a different code granularity, namely class-, method-, and file-level granularity. We aim to assess the effectiveness of the query reformulation strategies across these code granularities. The data sets we collected are the following:

- The class-level data set (*i.e.*, **CDS**) is based on Moreno *et al.*'s bug localization data [167] from 16 versions of 13 open source projects (*e.g.*, ArgoUML v0.22 or OpenJPA v2.0.1). This data set accounts for 815 queries.

- The file-level data set (*i.e.*, **FDS**) is based on Wong *et al.*'s bug localization data [225] and on data from the Bench4BL TRBL benchmark [142]. From Wong *et al.*'s data, we used two projects, namely Eclipse v3.1 and SWT v3.1, and from Bench4BL, we used nine projects (*e.g.*, Commons IO, Jboss Wildfly Core, and Spring Data MongoDB). This data set accounts for 4,429 queries from 172 versions from 11 software projects.

Table 6.3: Statistics of the TRBL datasets.

Data set	**Code granularity**	**# of systems**[a]	**# of code documents**[b]	**# of queries**	**# of buggy code documents**[c]
CDS	Class	13 (16)	2,366.9	815	2 (3.4)
FDS	File	11 (172)	1,669.2	4,429	2 (3.1)
MDS	Method	14 (65)	16,704.9	360	1 (5.9)
Total		**30 (248)**	**7,523.5**	**5,604**	**2 (3.3)**

[a]In parenthesis, # of system versions. [b]Average values across system versions. [c]Median (avg.) values across queries.

- The method-level data set (*i.e.*, **MDS**) is based on Mills *et al.*'s data on query quality assessment [161], and on Chaparro *et al.*'s adaptation [83] of Just *et al.*'s Defects4J data [128] for TRBL. Both existing data sets account for 360 queries from 65 versions of 14 open source projects (*e.g.*, Apache Lang and JEdit v4.2).

We built the three data sets from the data set we collected in our prior work [83]. However, we expanded the FDS data set[3] with nine systems from the Bench4BL benchmark, which span different domains across three open source ecosystems (*i.e.*, Apache, Spring, and Jboss). The total number of queries for these systems is 1,340.

As the data sources we used to compile our three data sets were built independently, some projects and versions are used in more than one data set. This is the case of Apache Derby v10.9.1.0, which belongs to the MDS and CDS data sets. The total number of systems and versions without this overlap is shown in Table 6.3. Given the overlap, our whole data set contains 98 queries from five projects that belong to both the MDS and CDS data sets (*i.e.*, they are duplicated). In addition to these cases, our whole data set includes extra duplicated queries. Since a bug can affect multiple versions of a software system, it is possible to have the same queries for multiple system versions. Our data set contains a subset of these cases: 6 queries that belong to two different versions of three projects (one from MDS and two from CDS). In addition, the FDS data set contains 98 additional queries that belong to

[3]This data set is called BRT in our prior work [83].

both Eclipse and SWT. These queries are originally duplicated in Wong *et al.*'s data [225]. The duplication stems from the fact that SWT is a subproject of Eclipse. In any case, the code corpus for both systems is different. In total, 202 queries in our whole data set are duplicated. We decided to keep these queries because they are likely to behave differently across different granularities, system versions, and code corpora. A query can fail to retrieve the buggy method(s) while succeeding at retrieving the buggy class(es). Likewise, a query can fail to retrieve the buggy code documents for one system version (or code corpus), while succeeding for another one. This means that, in some sense, they can be treated as different queries. Also, removing them from our data set would imply a lower number of queries, which is undesirable (especially for MDS and CDS). The online replication package includes the full list of projects, versions, and queries, including the duplicated ones [84].

We were not able to use all the queries from the Bench4BL data [142] for the nine systems we selected because, for several queries, the gold set files did not exist in the code corpus. We decided to discard these cases, which amount to 84 queries (out of 1,340 queries for the selected nine Bench4BL systems). In addition, we re-formatted the remaining 1,256 queries because of the XML-character-escaping problem we described in Section 6.3.2 and also so that we could use our existing code for running the queries with the file-level TRBL approaches (*i.e.*, Lucene, BugLocator, BRTracer, and Locus).

All three data sets include: (1) a set of queries generated from the bug reports submitted on the projects' issue tracker; (2) the corresponding fixed (*a.k.a.* buggy) code artifacts that represent the gold set; and (3) the source code corpora which represent the document search space for bug localization. In total, our data sets amount to 5,604 queries from 248 versions of 30 open source projects written in Java, which vary in size and domain (*e.g.*, software development, databases, bibliography management, or text edition), and span different software types (*e.g.*, desktop, web, libraries, or frameworks).

We created a document corpus from the source code of each software version (*i.e.*, one corpus per version) according to the granularity of each data set. The corpus was created by extracting the identifiers, comments, and string literals present in the source code. All (test and production) Java files within each project were included in the corresponding corpus. All documents in the code corpus and the queries were normalized using standard preprocessing for text retrieval, such as identifier splitting based on the camel case and underscore formats (*e.g.*, *CodeIdentifier* or *code_identifier* would split into *code* and *identifier*), special characters removal (*e.g.*, # or $), common English stop words and Java keywords removal (*e.g.*, for, while, at, with, *etc.*), and stemming based on Porter's algorithm [175]. We implemented and executed this preprocessing before running the TRBL techniques, except for Locus. This is because Locus incorporates the preprocessing by default in its implementation, which is similar to the preprocessing we just described – see [224] for more details.

6.3.4 Low-quality Queries

As mentioned in Section 6.2, our reformulation approach follows the scenario in which the developer issues the initial query and inspects the top-N code candidates returned by the TRBL technique. If none of the candidates are deemed buggy, the developer reformulates the query (via the reformulation strategies) and inspects additional N candidates. In this scenario, the developer would inspect a total of 2N candidates. Large N values (say 20 and beyond) would mean that our approach is impractical because, in the worst-case scenario, it would imply inspecting 40 results total, which could demand a significant effort from the developer. It is likely that developers will change strategies before investing so much in retrieval. Very small N values (say less than 5) would imply an unrealistic scenario. If the developers find the buggy code within the top-5 results, then they do not need a reformulation. According to existing research on code search[4] [178, 202], developers inspect (on average) between 10 and 13 results within a single search session (*i.e.*, issuing a query and inspecting the results). We contend that inspecting between 5 and 10 documents (*i.e.*,

[4]Code search is a task similar but more general than TRBL.

10 to 20 documents total, following reformulation) is a realistic scenario for TRBL. In other words, if a query retrieves the buggy code in top-5, then it is likely that no reformulation is needed. Similar thresholds have been used in prior TRBL research [161, 142, 157, 105, 220, 245, 167, 225, 83]. Given that there is no specific research on user behavior during query reformulation for TRBL (to the best of our knowledge), we do not want to limit the evaluation only to the thresholds we consider most realistic. Hence, we include results for the threshold set N={5, 6, 7, ..., 30}, which amounts to 26 thresholds total. The replication package includes the results for *high-quality* queries (*i.e.*, for N={1, 2, 3, 4}).

Our reformulation strategies focus on queries that fail to retrieve the buggy code artifacts within the top-N results (*i.e.*, *low-quality queries*). Therefore, in order to determine the set of *low-quality* queries, we executed each of the TRBL techniques with the initial queries (generated from the entire text of the bug report's title and description) and checked if none of the buggy code artifacts were retrieved in the top-N results, for N={5, 6, 7, ..., 30}.

As some TRBL techniques do not support all code granularities, we executed the techniques on the data corresponding to their granularity. Since Lucene does not depend on the granularity, we used it on all three data sets. Lobster was used only on the CDS data set, while BugLocator, BRTracer, and Locus were executed on the FDS data set. We executed Locus using all queries for all FDS projects, except for Eclipse. For this project, we used the queries that correspond to the Eclipse sub-projects JDT and PDE. The reason is that the Eclipse code repository, which is required for running Locus' implementation, is nowadays managed separately into sub-projects (*i.e.*, each sub-project has its own code repository), and JDT and PDE are among the largest sub-projects within Eclipse. In this way, we maximized the number of queries when running the four file-level TRBL techniques used in the evaluation. Note that these two sub-projects are also used in Bench4BL [142] and in Locus' original evaluation [224], as opposed to the entire Eclipse project. Finally, we executed the queries on the corpus of their specific system version; in other words, we adopted a "multiple version matching" strategy [142].

Table 6.4: Number of *low-quality* queries for N=5.

Data set	**Lucene**	**Lobster**	**BRTracer**	**BugLocator**	**Locus**
CDS	305 (37.4%)	49 (6.0%)	-	-	-
FDS	1,768 (39.9%)	-	1,721 (38.9%)	2,583 (58.3%)	715 (16.1%)
MDS	199 (55.3%)	-	-	-	-
Total	**2,272 (40.5%)**	**49 (0.9%)**	**1,721 (30.7%)**	**2,583 (46.1%)**	**715 (12.8%)**

All proportions are computed with respect to the total number of queries for each data set (*i.e.*, the values from column "# of queries" in Table 6.3.)

Table 6.4 shows the proportion of the queries for which each one of the TRBL techniques fail to retrieve the buggy code documents within the top-5 results (which contain the query subsets for larger N values). Except for Lobster and Locus, there is a large proportion of *low-quality* queries (from 37.4% to 58.3%) across TRBL techniques. The main reason for having fewer *low-quality* queries for Lobster is that this technique works only on bug reports that contain stack traces (*i.e.*, on 139 reports from the CDS data set). This means that the 49 *low-quality* queries for Lobster represent, in fact, 35.3% of the CDS queries. Similarly, for Locus, we obtained 715 *low-quality* queries because, as mentioned before, Locus was executed on two Eclipse sub-projects (*i.e.*, JDT and PDE) and not on the entire Eclipse project, so in total, Locus executed 2,261 queries of the FDS data set. Hence, the 715 *low-quality* queries represent 31.6% of the queries. These results motivate our research on reformulating the *low-quality* queries to improve TRBL.

We found that 658 (*i.e.*, 19.2%) queries consistently fail to retrieve the buggy code artifacts within the top-5 results across all TRBL techniques. In total, 3,431 (*i.e.*, 61.2%), 2,833 (*i.e.*, 50.6%), 2,474 (*i.e.*, 44.1%), 2,249 (*i.e.*, 40.1%), 2,065 (*i.e.*, 36.8%), and 1,916 (*i.e.*, 34.2%) queries are *low-quality* for at least one of the TRBL techniques, when the top-5, -10, -15, -20, -25, and -30 results are inspected, respectively (see Table 6.5).

Table 6.5: Number of *low-quality* queries for N={5, 10, ..., 30}.

Data set	**Top-5**	**Top-10**	**Top-15**	**Top-20**	**Top-25**	**Top-30**
CDS	307 (37.7%)	231 (28.3%)	202 (24.8%)	176 (21.6%)	161 (19.8%)	139 (17.1%)
FDS	2,925 (66.0%)	2,427 (54.8%)	2,115 (47.8%)	1,926 (43.5%)	1,773 (40.0%)	1,653 (37.3%)
MDS	199 (55.3%)	175 (48.6%)	157 (43.6%)	147 (40.8%)	131 (36.4%)	124 (34.4%)
Total	**3,431 (61.2%)**	**2,833 (50.6%)**	**2,474 (44.1%)**	**2,249 (40.1%)**	**2,065 (36.8%)**	**1,916 (34.2%)**

Size of the union set of queries across all TRBL techniques.

Proportions with respect to the total # of queries for each data set.

6.3.5 Structure Identification in Bug Descriptions

In order to answer our research question, we need to manually identify the terms corresponding to the system's observed behavior (**OB**), the expected behavior (**EB**), and the steps to reproduce (**S2Rs**) in the bug reports that require reformulation (*i.e.*, the ones that are *low-quality* queries), just as a potential user would do. We also need to identify the code snippets (**CODE**) in the bug reports. The bug title (**TITLE**) is present as a separate field within the bug report and its identification is trivial.

Bug Report Sampling. As shown in Table 6.5, the number of bug reports in the CDS and MDS data sets is manageable for manual identification (*i.e.*, 307 and 199 bug reports for N=5, respectively). This is not the case for the FDS data set, which contains 2,925 *low-quality* reports/queries for N=5. Therefore, we took a random sample of the FDS bug reports (for N=5), and selected *all* the reports from the other two data sets, manually excluding the ones referring to new features and enhancements (*i.e.*, non-bugs). We sampled a set of 792 (out of 2,925, *i.e.*, 27.1%) FDS bug reports, ensuring that the sample includes reports for each project in the FDS data set (see Table 6.6).

In total, our sample includes 1,221 bug reports used as queries (*i.e.*, 792 FDS + 270 CDS + 159 MDS bug reports), which fail to retrieve the buggy code artifacts within the top-5 results when using the TRBL techniques (see Table 6.6). This represents 35.6% of the 3,431 *low-quality* queries for N=5. This query set also contains a subsample of the queries that fail to retrieve to buggy code in top-10 (*i.e.*, 1,058 or 30.8% of the queries), in top-15 (*i.e.*, 958 or 27.9% of the queries), in top-20 (*i.e.*, 895 or 26.1% of the queries), in top-25 (*i.e.*, 837 or 24.4% of the queries), and in top-30 (*i.e.*, 785 or 22.9% of the queries) – see Table 6.6. It is important to note that while we experimented with top-N results for N={5, 6, ... , 30}, our reformulation strategies and their evaluation can be done for any threshold N. Table 6.7 shows the distribution of the sampled queries across TRBL techniques. The number of *low-quality* reports/queries in our sample varies from technique to technique because some queries are not *low-quality* when given as input to one or more techniques. Also, the total number of sampled queries for CDS and MDS shown in Table 6.5 is lower than the total number of *low-quality* queries shown in Table 6.6. The difference between these values represents the number of reports that we discarded manually (*i.e.*, new feature and enhancement requests, rather than bug reports).

As mentioned in Section 6.3.3, a handful of queries are duplicated across the three data sets and projects versions. Likewise, our sample contains 16 queries that belong to both MDS and CDS, one additional duplicated query for different versions of Derby in CDS, and 9 extra queries that are duplicated across Eclipse and SWT in FDS. We kept these (26) queries in our sample because their respective code corpus (and granularity) is different, hence, they can be treated as different queries. In any case, given the small proportion of these queries (*i.e.*, 4.3% total), we believe that their impact in the results is minimal.

Identification of the OB, EB, and S2Rs. Two PhD and one master student conducted the identification of the OB, EB, and S2Rs in the 1,221 sampled bug reports. The reports

Table 6.6: Number of sampled queries for N={5, 10, ..., 30}.

Data set	Top-5	Top-10	Top-15
CDS	270 (87.9%)	205 (66.8%)	181 (59.0%)
FDS	792 (27.1%)	715 (24.4%)	653 (22.3%)
MDS	159 (79.9%)	138 (69.3%)	124 (62.3%)
Total	**1,221 (35.6%)**	**1,058 (30.8%)**	**958 (27.9%)**

Data set	Top-20	Top-25	Top-30
CDS	159 (51.8%)	145 (47.2%)	123 (40.1%)
FDS	619 (21.2%)	587 (20.1%)	562 (19.2%)
MDS	117 (58.8%)	105 (52.8%)	100 (50.3%)
Total	**895 (26.1%)**	**837 (24.4%)**	**785 (22.9%)**

Size of the union query set across TRBL techniques. Proportions with respect to the total # of *low-quality* queries (for N=5) for each data set.

Table 6.7: Number of sampled queries for N=5.

Data set	Lucene	Lobster	BRTracer	BugLocator	Locus
CDS	268 (87.3%)	49 (16.0%)	-	-	-
FDS	653 (22.3%)	-	630 (21.5%)	728 (24.9%)	332 (11.4%)
MDS	159 (79.9%)	-	-	-	-
Total	**1,080 (31.5%)**	**49 (1.4%)**	**630 (18.4%)**	**728 (21.2%)**	**332 (9.7%)**

Proportions with respect to the total # of *low-quality* queries for each data set (for N=5).

were distributed evenly among the coders in such a way that one report was coded by one person. The three coders conducted sentence-level qualitative coding [199] on the bug reports using the coding framework and criteria defined in our prior work [88, 83]. The coders' job was to tag the sentences in the title and description of the reports that corresponded to the OB, EB, and S2Rs. This task was performed using the web-based text annotation tool BRAT [15], in combination with one of our own tools that splits the text into sentences and paragraphs, based on the Stanford CoreNLP toolkit [156] and heuristics (*e.g.*, using punctuation and line breaks).

We summarize the most important criteria used by the coders to tag the OB, EB, and S2Rs in the bug descriptions. The full list can be found in the replication package [84].

OB Coding Criteria:

- The coding of OB focused only on natural language content written by the reporters, ignoring code snippets, stack traces, or program logs. However, the natural language referencing this information may indicate OB. Such cases were allowed for coding. An example of this case is: "*When I click the File menu, I get the following error and stack trace: ...*".

- Internal behavior of the system, described by the reporters, was also allowed for coding, for example: "*The open() method in the class FileMenu reads the menu options from the XML file...*".

- Descriptions of graphical user interface issues can be considered as OB, for example: *"The menu's color is too light, it should be darker"*.

- Uninformative sentences, such as "*The File menu does not work*" are insufficient to be considered OB. There must be a clear description of the software's OB, *e.g.*, "*The File menu doesn't open when I click on it*".

- Explanations of attached code to the bug reports are not considered OB, for example: *"The attached code defines the openMenu() method, which iterates on the menu options..."*.

EB Coding Criteria:

- Only sentences written by the reporters corresponding to the expected software behavior were allowed for coding.

- Like for OB, uninformative sentences, such as "*The File menu should work*" are insufficient to be considered EB. Only sentences with a clear description of the EB were allowed for coding, for instance: "*The File menu should open when I click on it*".

- Solutions or recommendations to solve the bug are not considered EB, hence they were not allowed for coding. An example of these cases is: *"You should refactor the FileMenu class..."*

- Imperative sentences that do not describe S2Rs may convey EB, for example: *"Make the File menu not to open automatically when I hover over it"*. However, often times, imperative sentences describe tasks that should be completed by developers, instead of describing EB [83].

S2R Coding Criteria:

- One or more sentences in a bug report can express steps to reproduce. The sentences may form a complete paragraph or be part of one. A paragraph describing S2Rs may contain OB or EB sentences. In such cases, the OB/EB/S2Rs sentences must be tagged accordingly. In any case, only the S2R text written by users is allowed for coding. This means that source code, commands, or attachments to the bug report are excluded from coding. The natural language referencing this information may indicate S2Rs and was allowed for coding. An example of this case is: "*When I execute the script attached, I get the following error: ...*".

- Some S2Rs may be labeled with phrases such as "to reproduce:" or "steps to reproduce". The label per se cannot be considered S2Rs, only the natural language content that such phrases are labeling must be coded as S2Rs. Figure 6.2 shows an example.

- Imperative and conditional sentences are often used to describe the S2Rs [88]. However, only the sentences giving enough details about how to reproduce the bug were allowed for coding. For example, the sentence *"When I use the wall, facebook will not retrieve..."* does not give details on the S2Rs, thus should not be tagged. In contrast,

Bug report title:
[Contexts] performance: Slow cursor navigation in Text fields [TITLE]

Bug report description:
N20050623-0010-gtk.

Not sure whether this is my particular install or a new problem.

With the above build, cursor navigation in any text field is extremely slow. [OB]

Steps:
- open the Find dialog (alternatively: open the preference dialog, or the Team>Create Patch wizard)
- in a text field, enter some text, at least 20 characters
- press and hold the ARROW_LEFT / _RIGHT keys [S2Rs]

Expected: I see the caret move through the entered text [EB]
Actual: The cursor does not visibly change its position until after releasing the key plug some delay. [OB]

[...]

Legend: TITLE OB EB S2Rs

TITLE: bug report title, OB: observed behavior, EB: expected behavior, S2Rs: steps to reproduce.

Bug report found at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=101434`

Figure 6.2: Bug report #101434 from Eclipse.

the sentence *"When I share a URL in my Facebook wall page, Facebook will not retrieve..."* gives a specific and clear description of the S2Rs, hence should be tagged accordingly.

We made the choice of having one coder for each bug report in order to maximize the number of queries used in the evaluation. Given the nature of the coding task, one would expect differences between different coders [88, 83]. Our future work will investigate the differences between coders and assess the robustness of the proposed reformulation strategies with respect to these differences. In any case, our past experience when we had multiple coders per bug report, revealed high agreement between coders.

Identification of TITLE and CODE. The identification of TITLE and CODE in the sampled bug reports was performed automatically. The TITLE was given by the default structure of the bug reports collected from the issue trackers as they contain a separate field for the title. The CODE was identified using the StORMeD island parser provided by Ponzanelli *et al.* [44, 174], which automatically identifies (in)complete multi-language code elements within natural language documents. We only considered code snippets as CODE, as opposed to identifiers referenced in the text written by the reporters.

Structure Identification Results. Overall, 1,185 (*i.e.*, 97.1%) of the tagged bug reports describe an OB (see Table 6.8), and only 284 (23.2%), 625 (51.2%), and 481 (39.4%) of the bug reports contain sentences corresponding to EB, S2Rs, and CODE, respectively. These proportions are in line with the ones measured in other bug reports data sets [98, 88, 83]. The TITLE is always present in all bug reports, and the OB is found in almost all of them, hence they are more applicable for query reformulation than the other bug information. The coding required significant manual effort for the 1,221 reports, however, in an actual usage scenario, the user only needs to select the OB/EB/S2Rs/TITLE/CODE sentences from a single report, which takes seconds. For example, from the bug report shown in Figure 6.2, the user would only select the highlighted text and use it for reformulation, depending on the reformulation strategy. Note that the OB, EB, S2Rs, or CODE may be described in non-contiguous parts of the text, including in parts of the title. Other parts of the bug description are ignored when reformulating the query.

Table 6.8: Number of sampled queries containing TITLE, OB, EB, S2Rs, and CODE.

Data set	**TITLE**	**OB**	**EB**	**S2Rs**	**CODE**
CDS	270 (100%)	266 (98.5%)	65 (24.1%)	142 (52.6%)	118 (43.7%)
FDS	792 (100%)	763 (96.3%)	180 (22.7%)	421 (53.2%)	286 (36.1%)
MDS	159 (100%)	156 (98.1%)	39 (24.5%)	62 (39.0%)	77 (48.4%)
Total	**1,221 (100%)**	**1,185 (97.1%)**	**284 (23.3%)**	**625 (51.2%)**	**481 (39.4%)**

6.3.6 Evaluation Procedure and Metrics

The evaluation focuses on the initial queries that fail to retrieve the buggy code artifacts in top-N (*i.e.*, *low-quality* queries), in the first step of our proposed bug localization scenario (see Section 6.2). We reformulate the *low-quality initial queries* by retaining the sentences tagged as OB, EB, S2Rs, TITLE, or CODE, and removing the rest of the sentences in the bug description, depending on the reformulation strategy. We call the reformulated queries *reduced queries*. Note that if a sentence is tagged as more than one type of content (*e.g.*, both OB and S2Rs), we include the sentence only once in the *reduced query*. We reformulated all 1,221 initial queries using each one of the reformulation strategies. The only condition for having a valid reduced query, given a reformulation strategy, is the presence of all types of information in the bug descriptions corresponding to the strategy. For example, for the strategy OB+EB+CODE, we reformulated only the initial queries containing OB, EB, and CODE in their bug descriptions. When all five information types are present in the bug report (*i.e.*, TITLE, OB, EB, S2Rs, and CODE), we will have the initial query and 31 reduced queries.

We executed the initial and reduced queries with the five TRBL techniques, depending on the code granularity (see Table 6.9). We measured the TRBL performance using HITS@N, which is the proportion of queries for which a TRBL approach returns at least one buggy code document within the top-N candidates. This is one of the most commonly used measures in TRBL research [220, 245, 167, 225] and it is ideal for assessing the performance of TRBL

Table 6.9: Number of reduced queries for N=5.

Reformulation strategy	**Lucene**	**Lobster**	**BugLocator**	**BRTracer**	**Locus**
TITLE	1,080 (100%)	49 (100%)	728 (100%)	630 (100%)	332 (100%)
OB	1,047 (97%)	48 (98%)	699 (96%)	605 (96%)	318 (96%)
OB+TITLE	1,048 (97%)	48 (98%)	699 (96%)	605 (96%)	318 (96%)
S2R	544 (50%)	29 (59%)	387 (53%)	333 (53%)	202 (61%)
S2R+TITLE	547 (51%)	29 (59%)	391 (54%)	337 (53%)	202 (61%)
OB+S2R	542 (50%)	29 (59%)	387 (53%)	334 (53%)	200 (60%)
OB+S2R+TITLE	542 (50%)	29 (59%)	387 (53%)	334 (53%)	200 (60%)
CODE	403 (37%)	23 (47%)	249 (34%)	218 (35%)	161 (48%)
TITLE+CODE	415 (38%)	25 (51%)	255 (35%)	222 (35%)	161 (48%)
OB+CODE	402 (37%)	25 (51%)	244 (34%)	213 (34%)	155 (47%)
OB+TITLE+CODE	402 (37%)	25 (51%)	244 (34%)	213 (34%)	155 (47%)
S2R+CODE	244 (23%)	15 (31%)	166 (23%)	142 (23%)	110 (33%)
S2R+TITLE+CODE	244 (23%)	15 (31%)	166 (23%)	142 (23%)	110 (33%)
OB+S2R+CODE	242 (22%)	15 (31%)	164 (23%)	141 (22%)	109 (33%)
OB+S2R+TITLE+CODE	242 (22%)	15 (31%)	164 (23%)	141 (22%)	109 (33%)
EB	237 (22%)	5 (10%)	167 (23%)	135 (21%)	78 (23%)
EB+TITLE	240 (22%)	5 (10%)	167 (23%)	135 (21%)	78 (23%)
OB+EB	232 (21%)	4 (8%)	161 (22%)	129 (20%)	76 (23%)
OB+EB+TITLE	232 (21%)	4 (8%)	161 (22%)	129 (20%)	76 (23%)
EB+S2R	131 (12%)	1 (2%)	92 (13%)	75 (12%)	48 (14%)
EB+S2R+TITLE	131 (12%)	1 (2%)	92 (13%)	75 (12%)	48 (14%)
OB+EB+S2R	129 (12%)	1 (2%)	91 (13%)	74 (12%)	48 (14%)
OB+EB+S2R+TITLE	129 (12%)	1 (2%)	91 (13%)	74 (12%)	48 (14%)
EB+CODE	91 (8%)	3 (6%)	67 (9%)	50 (8%)	41 (12%)
EB+TITLE+CODE	91 (8%)	3 (6%)	67 (9%)	50 (8%)	41 (12%)
OB+EB+CODE	89 (8%)	3 (6%)	65 (9%)	48 (8%)	40 (12%)
OB+EB+TITLE+CODE	89 (8%)	3 (6%)	65 (9%)	48 (8%)	40 (12%)
EB+S2R+CODE	55 (5%)	1 (2%)	43 (6%)	30 (5%)	26 (8%)
EB+S2R+TITLE+CODE	55 (5%)	1 (2%)	43 (6%)	30 (5%)	26 (8%)
OB+EB+S2R+CODE	55 (5%)	1 (2%)	43 (6%)	30 (5%)	26 (8%)
OB+EB+S2R+TITLE+CODE	55 (5%)	1 (2%)	43 (6%)	30 (5%)	26 (8%)
Total	**1,080**	**49**	**728**	**630**	**332**

techniques as, in practice, developers would likely inspect the top-N results only, rather than the full list of results.

Our empirical evaluation mimics an actual usage scenario of our reformulation approach, where the developer issues the initial query and inspects the N results returned by a TRBL engine (step #1). If she does not find any buggy code artifact, then she makes the choice of reformulating the initial query (*e.g.*, via any of the proposed reformulation strategies) or using the same initial query (*i.e.*, no reformulation) to retrieve additional N candidates (step #2). The N results returned by the initial query in step #1 are removed from the result lists produced in step #2 by both the reformulated query and the initial query (because they are deemed non-buggy), and then HITS@N is computed for both the initial and the reformulated query in the second step. We repeat this process for the queries generated based on all 31 reformulation strategies, using the five TRBL approaches, for N={5, 6, 7, ..., 30} – 26 thresholds N total. The replication package also includes the results for N={1, 2, 3, 4} and for additional evaluation metrics (see below). In the end, if the HITS@N for the reformulated queries is higher than the one for the initial queries, we can conclude that the reformulation is a better strategy. If the measures are the other way around, we can conclude that it is not worth reformulating the query, as there is no gain over just simply investigating N more results returned by the initial query. We perform the comparison only in the cases where an initial query is successfully reduced since otherwise, the reformulation would have no effect.

When comparing different TRBL techniques, researchers also use Mean Reciprocal Rank (MAP) and Mean Average Precision (MAP) [105].

Mean Reciprocal Rank (MRR) is a statistic that measures the quality of the ranking of TRBL technique by capturing how close to the top of the result list a relevant (*i.e.*, buggy) code document (to a query q) is retrieved. MRR is given by the average of the reciprocal rank of a set of queries Q:

$$\text{MRR}(Q) = \frac{1}{|Q|}\sum_{q\in Q}\frac{1}{\text{rank}(q)} \tag{6.1}$$

where rank(q) is the rank of the first buggy code artifact found in the result list produced by q. The higher the MRR value, the higher the ranking quality of the bug localization approach will be. MRR is an aggregate measure of how high the first relevant document ranks.

Mean Average Precision (MAP) is a measure of the accuracy of a retrieval approach based on the average precision of each query q in the set Q. Given R_q, the set of documents relevant to query q, the average precision is computed as the average of the precision values at the resulting rank of each document. MAP is the mean of the average precision of the set of queries Q, defined as follows:

$$\text{MAP}(Q) = \frac{1}{|Q|} \sum_{q \in Q} \frac{1}{|R_q|} \sum_{r \in R_q} \text{precision}(\text{rank}(r)) \tag{6.2}$$

where rank(r) is the rank of the buggy document r in the result list and precision(N) = (# buggy docs. in top-N)$/N$, *i.e.*, the proportion of code documents found in top-N that are buggy. MAP reflects how well *all* the buggy code documents rank, in aggregate.

We measured the Magnitude of Improvement (Improv) for the metrics used in this evaluation (*i.e.*, HITS@N, MAP, and MRR), by computing the change percentage of metric M before (M_b) and after reformulation (M_a):

$$\text{Improv}(M) = \frac{M_a - M_b}{M_b} \tag{6.3}$$

We aim at maximizing *Improv*, avoiding negative values, which would mean deterioration rather than improvement. When M_a and M_b equal to zero, then *Improv* is zero. Otherwise, *Improv* is undefined when M_b is zero.

We assessed the statistical significance of our measures using the Mann-Whitney test [122], a non-parametric test for comparing paired samples whose distributions are not assumed to follow a normal distribution (which is our case). This method was used to test if an evaluation measure M, when applying a reformulation strategy (M_a), is higher than when using no reformulation (M_b). We carried out the test on the HITS@N, MRR, and MAP paired values that we collected across the 26 threshold values and 3 data sets, for each TRBL technique. For each metric M, we defined the null hypothesis as $H_0 : M_b \geq M_a$, and the alternative hypothesis as $H_1 : M_b < M_a$. We applied the test with a 95% confidence level, thus rejecting the null hypothesis, in favor of the alternative, if p-value $< 5\%$.

HITS@N vs. MRR/MAP. The main difference between HITS@N and MRR/MAP is that HITS@N is based on checking the top-N results only, while MRR and MAP are based on checking the entire list of retrieved code elements. HITS@N and MRR are based on the rank of the first buggy code artifact found in the result list (*i.e.*, the closest artifact to the top of the list), while MAP is based on the rank of all the buggy code artifacts in the result list (when there is more than one). Note that MRR and MAP are the same, when there is only one buggy code artifact for a query.

We focus the analysis on HITS@N for the following reasons: (1) Developers are likely to inspect the top-N results retrieved by a TRBL technique (rather than the entire list of results) before switching to other methods for localizing the bug (*e.g.*, navigating code dependencies). We contend that checking the top-N results only (captured by HITS@N) is more realistic than checking the entire result list (captured by MRR/MAP); (2) For the cases when more than one buggy code artifact exist, developers are likely to switch to other strategies when they find one of the buggy artifacts in the result list. It is likely that other strategies, such as navigating code dependencies, will lead to finding the other artifacts faster since the artifacts may be related in the code structure. In other words, it is more important for developers to retrieve one of the buggy code artifacts (rather than all of them) in top-N. The ranking of the other buggy artifacts, outside the top-N, is less important. HITS@N and MRR measure this phenomenon better than MAP, however, HITS@N is more intuitive and easier to interpret than MRR; (3) When comparing two TRBL techniques, MRR and MAP do a very good job in capturing the overall retrieval performance and support the comparison when the two techniques are tested with a large number of queries. However, we are not comparing two TRBL techniques using the same query, but comparing two queries used with the same TRBL technique: the reformulated query and the original one, for retrieving at least one buggy code artifact in the additional N results. In this case, HITS@N is more intuitive and easier to interpret than MRR and MAP, since it is based on the binary result of finding the first buggy artifact within the top-N results.

For the sake of completeness, we also measured MAP and MRR and included the results in the replication package [84]. We observed that MRR and MAP do not necessarily correspond with HITS@N. In some cases, we observed HITS@N improvement and MRR/MAP deterioration and also the other way around. For example, the reformulation strategy EB, when using Lucene, deteriorates HITS@N by 10.6%, but improves MRR/MAP by 35.5%/45.0%, on average[5]. As such, a MAP/MRR improvement may not mean that the developer will retrieve the buggy code faster after reformulation (*i.e.*, within the top-N results).

6.3.7 Analysis Framework

We define three criteria for determining the best reformulation strategies, and thus, answering our research question: **effectiveness**, **applicability**, and **consistency**.

We categorized the strategies by their **effectiveness**, in terms of HITS@N improvement. The strategies that lead to HITS@N improvement (*i.e.*, Improv(HITS@N) > 0) are called *effective*; the ones that lead to deterioration (*i.e.*, Improv(HITS@N) < 0) are called *ineffective*; and those that lead to no change of HITS@N (*i.e.*, Improv(HITS@N) $= 0$) are called *neutral*. We defined two sub-categories for the *effective* and *ineffective* categories, based on the entire set of HITS@N improvement values that we collected for each TRBL technique, each data set/granularity, and each threshold N. We relied on the distribution quartiles to define the criteria for categorizing the strategies across TRBL techniques, granularities, and thresholds N. Specifically, we used the 1st and 3rd quartiles of the entire HITS@N improvement distribution, whose values are -20.6% and 21.4%, respectively – the median is zero. Hence, we categorized the strategies that lead to improvement up to 21.4% (*i.e.*, $0\% <$ Improv(HITS@N) $\leq 21.4\%$) as *somewhat-effective*, and those that lead to higher improvement (*i.e.*, Improv(HITS@N) $> 21.4\%$) as *very-effective*. Likewise, the strategies that lead to deterioration up to 20.6% (*i.e.*, $-20.6\% \leq$ Improv(HITS@N) $< 0\%$) are categorized as

[5]See Table 6.11 and the replication package for more details.

Table 6.10: Reduced queries and applicability of each reformulation strategy.

Reformulation strategy	**Reduced queries**	**Applicability**
TITLE	1,221 (100.0%)	High
OB	1,185 (97.1%)	
OB+TITLE	1,185 (97.1%)	
S2R	625 (51.2%)	Moderate
S2R+TITLE	625 (51.2%)	
OB+S2R	619 (50.7%)	
OB+S2R+TITLE	619 (50.7%)	
CODE	481 (39.4%)	Somewhat
TITLE+CODE	481 (39.4%)	
OB+CODE	468 (38.3%)	
OB+TITLE+CODE	468 (38.3%)	
S2R+CODE	290 (23.8%)	Low
S2R+TITLE+CODE	290 (23.8%)	
OB+S2R+CODE	288 (23.6%)	
OB+S2R+TITLE+CODE	288 (23.6%)	
EB	284 (23.3%)	
EB+TITLE	284 (23.3%)	
OB+EB	275 (22.5%)	
OB+EB+TITLE	275 (22.5%)	
EB+S2R	159 (13.0%)	
EB+S2R+TITLE	159 (13.0%)	
OB+EB+S2R	156 (12.8%)	
OB+EB+S2R+TITLE	156 (12.8%)	
EB+CODE	120 (9.8%)	
EB+TITLE+CODE	120 (9.8%)	
OB+EB+CODE	118 (9.7%)	
OB+EB+TITLE+CODE	118 (9.7%)	
EB+S2R+CODE	77 (6.3%)	
EB+S2R+TITLE+CODE	77 (6.3%)	
OB+EB+S2R+CODE	77 (6.3%)	
OB+EB+S2R+TITLE+CODE	77 (6.3%)	

Size of the union query set across the five TRBL techniques.

somewhat-ineffective, and those that lead to higher deterioration (*i.e.*, Improv(HITS@N) $< -20.6\%$) as *very-ineffective*. Note that one strategy can fall in one sub-category for a particular TRBL technique, granularity, or threshold N, and fall in another sub-category for another {technique, granularity, threshold} combination.

Regarding **applicability**, we categorize the reformulation strategies according to the number of initial queries that can be reformulated by each one of the strategies. Table 6.10 shows that the OB, TITLE, and OB+TITLE strategies can be used to reformulate nearly all initial queries (*i.e.*, 97.1% - 100%), which means they are the most applicable strategies in an actual usage scenario. We call these strategies *highly-applicable*, which are characterized for retaining the OB and TITLE sentences. The strategies S2R, OB+S2R, S2R+TITLE, and OB+S2R+TITLE are applicable in ∼50% of the cases, hence we call them *moderately-applicable*. In addition to OB and TITLE, these strategies retain the S2R sentences found in the bug reports. The strategies CODE, TITLE+CODE, OB+CODE, and OB+TITLE+CODE are categorized as *somewhat-applicable* because they reformulate ∼38% of the initial queries. Note that the CODE is the common information type across these strategies. The remaining strategies can be applied to less than 25% of the queries, hence their applicability is *low*. The reformulation strategies using EB alone or in combination with other information types belong to this sub-category.

The third criterion is the **consistency** that a strategy achieves across all thresholds N. In other words, we aim to determine if a strategy is *effective*, *neutral*, or *ineffective* for most (if not all) thresholds N, within the selected set, *i.e.*, N={5, 6, ..., 30}. Therefore, the degree of consistency is determined by the proportion of thresholds N (out of the 26 values) for which a strategy is *effective*, *neutral*, and *ineffective*, depending on the {technique, granularity, threshold} combination. Ideally, a reformulation strategy improves HITS@N for all 26 thresholds. However, a strategy may lead to improvement for some thresholds, and to deterioration or no effect for some others. For instance, the strategy OB+S2R may be *effective* for 22 thresholds, *neutral* for another one, and *ineffective* for the remaining three. Hence, the best strategies are the ones that maximize the number of N values for which they improve HITS@N (*i.e.*, *effective*), while minimizing the number of thresholds N for which they deteriorate HITS@N (*i.e.*, *ineffective*).

Table 6.11: TRBL performance of each reformulation strategy when using Lucene.

Reformulation strategy	# of queries	HITS@N			Thresh.		
		No reform.	Reform.	Improv.	E	N	I
OB+EB+TITLE	155.2	39.5 (26.0%)	51.1 (33.7%)	30.3%	26	0	0
OB+TITLE	763.6	173.5 (23.3%)	224.0 (29.9%)	29.3%	26	0	0
OB+S2R+TITLE	409.0	90.7 (22.9%)	116.0 (29.0%)	29.2%	26	0	0
OB+EB+S2R+TITLE	90.3	20.8 (23.5%)	26.5 (30.3%)	28.6%	25	0	1
OB+EB	155.2	39.5 (26.0%)	49.4 (32.5%)	26.1%	26	0	0
OB+TITLE+CODE	285.1	69.2 (24.9%)	86.2 (30.8%)	25.6%	26	0	0
OB+EB+S2R	90.3	20.8 (23.5%)	25.8 (29.4%)	24.9%	24	0	2
OB+S2R	409.0	90.7 (22.9%)	110.1 (27.5%)	22.4%	26	0	0
TITLE	783.5	178.7 (23.4%)	214.4 (27.9%)	20.1%	26	0	0
S2R+TITLE	409.9	91.5 (23.0%)	107.7 (26.8%)	19.1%	26	0	0
EB+TITLE	161.3	40.3 (25.5%)	46.9 (29.9%)	17.3%	25	0	1
OB	762.6	173.5 (23.3%)	201.5 (27.0%)	16.2%	26	0	0
OB+CODE	285.1	69.2 (24.9%)	78.8 (28.2%)	14.7%	26	0	0
EB+S2R+TITLE	90.8	21.2 (23.8%)	23.9 (27.7%)	14.4%	14	4	8
TITLE+CODE	292.5	70.9 (24.9%)	79.3 (27.5%)	12.4%	24	1	1
OB+EB+TITLE+CODE	55.3	15.1 (27.4%)	15.9 (29.0%)	6.7%	14	4	8
OB+S2R+TITLE+CODE	176.8	41.3 (24.0%)	43.7 (25.2%)	6.2%	18	3	5
OB+EB+CODE	55.3	15.1 (27.4%)	15.3 (28.1%)	3.3%	12	5	9
OB+S2R+CODE	176.8	41.3 (24.0%)	42.5 (24.6%)	3.2%	12	2	12
S2R+TITLE+CODE	176.9	41.5 (24.0%)	39.7 (22.8%)	-4.1%	8	2	16
OB+EB+S2R+TITLE+CODE	37.1	8.3 (22.3%)	7.2 (19.3%)	-9.8%	5	4	17
EB+S2R	90.8	21.2 (23.8%)	18.6 (21.4%)	-10.6%	6	1	19
EB	158.9	39.4 (25.2%)	34.8 (22.5%)	-10.6%	2	2	22
OB+EB+S2R+CODE	37.1	8.3 (22.3%)	6.8 (18.4%)	-15.1%	4	2	20
EB+TITLE+CODE	57.3	15.1 (26.4%)	12.3 (21.1%)	-20.0%	1	0	25
S2R+CODE	176.9	41.5 (24.0%)	29.8 (17.2%)	-27.9%	0	0	26
EB+CODE	57.3	15.1 (26.4%)	10.7 (18.7%)	-29.3%	0	0	26
S2R	408.5	91.1 (23.0%)	62.3 (15.5%)	-30.6%	0	0	26
CODE	284.5	69.2 (25.0%)	43.3 (15.6%)	-37.2%	0	0	26
EB+S2R+TITLE+CODE	37.1	8.3 (22.3%)	5.0 (13.1%)	-41.5%	0	1	25
EB+S2R+CODE	37.1	8.3 (22.3%)	4.3 (11.1%)	-49.8%	0	0	26

Average # of queries and HITS@N values across the 3 data sets and 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). Strategies sorted by average HITS@N improvement (Improv). All strategies with positive improvement, except EB+S2R+TITLE, achieve a statistically-significant higher HITS@N, compared to no reformulation (Mann-Whitney, *p*-value$<$ 5%). Number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

Intuitively, the best strategies are the ones that are *effective* (ideally, *very-effective*), *highly-* or *moderately-applicable*, and *improve* HITS@N for most thresholds N (out of the 26 N values). Conversely, the worst cases are the most *ineffective* strategies, whose applicability is *low*, and consistently *deteriorate* HITS@N for most thresholds N. Note that the *highly-applicable* strategies that are *ineffective* are quite undesirable, as they lead to a significant negative impact from a practical point of view, *i.e.*, they can be used frequently but they lead to retrieving the buggy code documents in the top-N results for fewer cases, compared to no reformulation.

6.4 Evaluation Results and Discussion

We present and discuss the results obtained from the empirical evaluation of the 31 reformulation strategies, across the three code granularities/data sets and 26 thresholds (N=5, 6, 7, ..., 30), for Lucene (Section 6.4.1), Lobster (Section 6.4.2), BugLocator (Section 6.4.3), BR-Tracer (Section 6.4.4), and Locus (Section 6.4.5). In addition, we analyze the results for each code granularity (Section 6.4.6) and for all TRBL techniques on aggregate (Section 6.4.7). We provide examples and discuss the best and worst reformulation strategies (Section 6.4.8), including the trade-offs between successful and unsuccessful queries (Section 6.4.9).

6.4.1 Performance for Lucene

Tables 6.11 and 6.12 show the results obtained for Lucene across the 26 thresholds N and three code granularities (or data sets). The replication package [84] contains the breakdown for each data set and each threshold N (N=5, 6, 7, ..., 30). The way to interpret the results in Table 6.11 is as follows. For example, let us look at the OB reformulation strategy, reading its corresponding row in the table. OB is present in 762.6 queries (on average across all N and data sets) – second column "# of queries". If the user investigates N more returned code documents without reformulating the initial query, then 173.5 (23.3%) of them will retrieve

Table 6.12: Categorization of each reformulation strategy when using Lucene.

		Effectiveness			
		VE	**SE**	**SI**	**VI**
Applicability	**H**	O+T (29.3%)	T (20.1%) O (16.2%)		
	M	O+S+T (29.2%) O+S (22.4%)	S+T (19.1%)		S (-30.6%)
	S	O+T+C (25.6%)	O+C (14.7%) T+C (12.4%)		C (-37.2%)
	L	O+E+T (30.3%) O+E+S+T (28.6%) O+E (26.1%) O+E+S (24.9%)	E+T (17.3%) E+S+T (14.4%) O+E+T+C (6.7%) O+S+T+C (6.2%) O+E+C (3.3%) O+S+C (3.2%)	S+T+C (-4.1%) O+E+S+T+C (-9.8%) E+S (-10.6%) E (-10.6%) O+E+S+C (-15.1%) E+T+C (-20.0%)	S+C (-27.9%) E+C (-29.3%) E+S+T+C (-41.5%) E+S+C (-49.8%)

In parenthesis, average HITS@N improvement across the 3 data sets and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Somewhat Ineffective (**SI**), and Very Ineffective (**VI**). The strategies in **green** belong to the *effective* category and the strategies in **red** to the *ineffective* category.
Information types: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

a relevant code document in top-N – third column "No reform.". Conversely, using OB to reformulate the queries results in 201.5 (27%) of them returning relevant code documents in top-N – fourth column "Reform.". This means 16.2% avg. improvement when reformulating – fifth column "Improv.".

We measured the number of thresholds N (out of the 26 N we used) for which each reformulation strategy is *effective*, *neutral* and *ineffective*. Table 6.11 also shows the results we obtained for Lucene regarding this aspect. Let us focus on the OB strategy once more. The last three columns in the table show that OB is *effective* (E) across all 26 thresholds N, while never being *neutral* (N) and *ineffective* (I). Another example is the following: the OB+EB+S2R reformulation strategy (in row #8) is *effective* for 24 thresholds N (sixth column 'E'), and *ineffective* for the remaining two thresholds (seventh column 'I'). The strategy is never *neutral* (*i.e.*, the value of the last column 'N' is zero).

We categorized each reformulation strategy into the categories defined in Section 6.3.7 for *effectiveness* and *applicability*, according to how much a strategy improves HITS@N and the number of queries it can reformulate. For Lucene, this categorization is shown in Table 6.12, and the way to interpret the table is as follows. For example, let us keep looking at the OB[6] reformulation strategy. As OB can be used to reformulate 97.1% of the queries (see Table 6.10), its *applicability* is considered *high* (HI). Since OB's HITS@N improvement (*i.e.*, 16.2%) is positive but less than 21.4%, the strategy is considered as *somewhat-effective* (SE). Therefore, the OB strategy (labeled as 'O' in the table) is placed in the cell of the second row and third column of Table 6.12, which corresponds to the intersection of the categories "Applicability-H" and "Effectiveness-SE". Note that the TITLE strategy (labeled as 'T' in the table) also belongs to this *Applicability-H/Effectiveness-SW* category, however, since it achieves a greater avg. HITS@N improvement than OB (*i.e.*, 20.1%), it ranks above OB.

Tables 6.11 and 6.12 reveal that 19 (out of 31) strategies improve HITS@N by 3.2% - 30.3%, on average (*i.e.*, they are *effective*). Among these, 8 strategies are *very-effective* (*i.e.*, their HITS@N improvement is higher than 21.4%), OB+EB+TITLE leading to the highest HITS@N improvement: it retrieves the buggy code document(s) for 30.3% more queries (on average) than without using the reformulation (*i.e.*, ∼51 vs ∼40 queries). This strategy also achieves 54.2% (51%) MRR (MAP) average improvement with respect to no reformulation - see our replication package for the full MRR/MAP results [84]. Table 6.12 reveals that while OB+EB+TITLE is the most effective, it is one of the least applicable strategies (because of EB). Other strategies are more applicable and achieve comparable effectiveness. OB+TITLE is the *highly-applicable* strategy that achieves the highest avg. HITS@N improvement (*i.e.*, 29.3% – also 57.8%/60.5% avg. MRR/MAP improvement). In fact, OB+TITLE is the second most effective strategy, which consistently improves HITS@N for all 26 thresholds N (see Table 6.11). The other two *highly-applicable* strategies, TITLE and OB, achieve lower avg. HITS@N improvement (*i.e.*, 20.1% and 16.2%, respectively), and fall in the *somewhat-effective* category. Among the *moderately-applicable* strategies, OB+S2R+TITLE

[6]We changed the notation in the table for space reasons.

is the most-effective (3rd most-effective overall), as it improves TRBL for 29.2% more queries (on avg.) compared to no reformulation (*i.e.*, 45.9%/51.2% avg. MRR/MAP improvement). As for the other *moderately-applicable* strategies, OB+S2R is *very-effective*, S2R+TITLE is *somewhat-effective*, and S2R *very-ineffective*. All *highly-* and *moderately-applicable* strategies, except S2R, consistently improve HITS@N for all thresholds, and their improvement is statistically-significant (Mann-Whitney, p-value < 5%). Our replication package contains the full results of the statistical tests for HITS@N, MRR, and MAP [84].

The remaining 12 strategies are *ineffective* (*i.e.*, they deteriorate HITS@N compared to the initial queries). Among these, EB+S2R+TITLE+CODE and EB+S2R+CODE are the ones with the lowest applicability and highest deterioration: more than 40% HITS@N/MAP/MRR deterioration (see Tables 6.11 and 6.12, and the online replication package). Also, these strategies consistently lead to deterioration for 25 thresholds N (see Table 6.11). Note that the strategies that retain the EB, S2R, and CODE alone or in combination belong to the *very-ineffective* category (their HITS@N deterioration is greater than 20.6%), S2R and CODE being the ones with the highest negative impact in practice, *i.e.*, they are *moderately-* and *somewhat-applicable*, respectively, and achieve high deterioration levels.

We conclude that OB+TITLE is the best reformulation strategy when using Lucene, because it is *very-effective*, *highly-applicable*, and *consistently* leads to HITS@N improvement (with respect to no reformulation) for all 26 thresholds N.

6.4.2 Performance for Lobster

The results obtained for Lobster reveal that 25 (out of 31) reformulation strategies are *effective* (see Tables 6.13 and 6.14). In particular, 23 strategies return the buggy code artifacts in top-N for 20% - 222.2% more queries (on average) than without reformulation. Unlike when using no reformulation, the other two strategies, *i.e.*, OB+EB+S2R(+TITLE), are able to return the buggy code document(s) for the only query they can reformulate (see

Table 6.13: TRBL performance of each reformulation strategy when using Lobster.

Reformulation strategy	**# of queries**	**HITS@N**			**Thresh.**		
		No reform.	**Reform.**	**Improv.**	**E**	**N**	**I**
OB+EB+TITLE	3.2	1.0 (31.5%)	3.2 (100.0%)	222.2%	18	8	0
OB+EB	3.2	1.0 (31.5%)	3.2 (100.0%)	222.2%	18	8	0
EB+TITLE	4.2	1.0 (23.9%)	3.2 (75.0%)	216.7%	18	8	0
EB	4.2	1.0 (23.9%)	2.2 (52.2%)	122.2%	18	8	0
S2R+TITLE	20.7	4.8 (22.2%)	9.0 (42.0%)	116.2%	23	3	0
OB+S2R+TITLE	20.7	4.8 (22.2%)	8.9 (41.5%)	113.8%	23	3	0
OB+S2R	20.7	4.8 (22.2%)	8.6 (40.3%)	108.9%	23	3	0
EB+TITLE+CODE	2.2	1.0 (46.3%)	2.0 (92.6%)	100.0%	18	8	0
OB+EB+TITLE+CODE	2.2	1.0 (46.3%)	2.0 (92.6%)	100.0%	18	8	0
OB+EB+CODE	2.2	1.0 (46.3%)	2.0 (92.6%)	100.0%	18	8	0
TITLE	35.6	6.8 (18.5%)	13.0 (35.2%)	97.1%	26	0	0
OB+TITLE+CODE	17.2	4.0 (22.9%)	7.7 (42.6%)	95.2%	26	0	0
TITLE+CODE	17.2	4.0 (22.9%)	7.5 (41.8%)	92.0%	26	0	0
OB+CODE	17.2	4.0 (22.9%)	7.4 (41.5%)	89.9%	26	0	0
OB+TITLE	34.6	6.8 (19.0%)	12.2 (34.4%)	88.3%	26	0	0
OB	34.6	6.8 (19.0%)	11.7 (33.2%)	82.1%	26	0	0
S2R	20.7	4.8 (22.2%)	6.4 (30.0%)	60.8%	17	9	0
S2R+TITLE+CODE	10.9	3.4 (31.7%)	5.3 (47.4%)	58.4%	19	7	0
OB+S2R+TITLE+CODE	10.9	3.4 (31.7%)	5.1 (46.0%)	55.3%	16	10	0
OB+S2R+CODE	10.9	3.4 (31.7%)	4.9 (45.0%)	50.9%	16	10	0
S2R+CODE	10.9	3.4 (31.7%)	4.8 (44.2%)	45.7%	19	7	0
CODE	16.1	3.9 (23.9%)	4.8 (29.1%)	28.9%	17	8	1
EB+CODE	2.1	0.9 (41.7%)	1.1 (48.3%)	20.0%	4	22	0
EB+S2R+TITLE	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
EB+S2R	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
EB+S2R+TITLE+CODE	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
EB+S2R+CODE	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
OB+EB+S2R+CODE	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
OB+EB+S2R+TITLE+CODE	1	0.0 (0.0%)	- (0.0%)	0.0%	0	26	0
OB+EB+S2R	1	0.0 (0.0%)	1.0 (100.0%)	-	26	0	0
OB+EB+S2R+TITLE	1	0.0 (0.0%)	1.0 (100.0%)	-	26	0	0

Average # of queries and HITS@N values across CDS and the 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). Strategies sorted by average HITS@N improvement (Improv). All strategies with positive improvement, including OB+EB+S2R(+TITLE), achieve a statistically-significant higher HITS@N, compared to no reformulation (Mann-Whitney, p-value< 5%). Number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

Table 6.14: Categorization of each reformulation strategy when using Lobster.

		Effectiveness				
		VE	**SE**	**N**	**SI**	**VI**
Applicability	**H**	T (97.1%) O+T (88.3%) O (82.1%)				
	M	S+T (116.2%) O+S+T (113.8%) O+S (108.9%) S (60.8%)				
	S	O+T+C (95.2%) T+C (92.0%) O+C (89.9%) C (28.9%)				
	L	O+E+T (222.2%) O+E (222.2%) E+T (216.7%) E (122.2%) E+T+C (100%) O+E+T+C (100%) O+E+C (100%) S+T+C (58.4%) O+S+T+C (55.3%) O+S+C (50.9%) S+C (45.7%)	E+C (20.0%)	E+S+T (0.0%) E+S (0.0%) E+S+T+C (0.0%) E+S+C (0.0%) O+E+S+C (0.0%) O+E+S+T+C (0.0%)		

In parenthesis, average HITS@N improvement across CDS and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Neutral (**N**),
Somewhat Ineffective (**SI**), and Very Ineffective (**VI**). The strategies in **green** belong to
the *effective* category and the strategies in **red** to the *ineffective* category.
Information types: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

the last two rows of Table 6.13[7]). All the *effective* strategies achieve HITS@N improvement with statistical significance (Mann-Whitney, p-value $< 5\%$). The 6 remaining strategies have no effect on TRBL (*i.e.*, they are *neutral*).

Twenty-two of the *effective* strategies belong to the *very-effective* category, OB+EB+TITLE and OB+EB being the most *effective* ones: they both achieve 222.2% avg. HITS@N

[7]HITS@N improvement cannot be measured for these two strategies because the HITS@N achieved by the initial queries (*i.e.*, no reformulation) is zero, hence, the improvement is undefined (see Formula 6.3).

improvement, and 1,167.6%/839.8% avg. MRR/MAP improvement[8]. However, the applicability of these strategies is rather *low*, as they can reformulate ∼3 initial queries only. All *highly-applicable* are *very-effective* (see Table 6.14), and consistently improve HITS@N across all 26 thresholds N, with respect to no reformulation (see Table 6.13). Among these, TITLE is the strategy with the highest effectiveness (*i.e.*, 97.1% avg. HITS@N improvement and 286.4%/243% avg. MRR/MAP improvement), followed by OB+TITLE, which is able to retrieve the buggy code documents for 88.3% more queries than the initial queries (on average). All *moderately-applicable* strategies, except S2R, consistently improve TRBL for 23 thresholds and are *neutral* for the remaining 3 N, and all of them are categorized as *very-effective*. It is important to note that the improvement rates for Lobster are (significantly) higher than for the other four TRBL approaches. This is because Lobster can only be used for the bug reports that contain stack traces, which are not many in our data sets (*i.e.*, between 1 and 49 queries, see Table 6.9).

We conclude that TITLE is the best reformulation strategy when using Lobster, since it is *very-effective*, *highly-applicable*, and it *consistently* retrieves more buggy code documents than no reformulation across all 26 thresholds N.

6.4.3 Performance for BugLocator

Tables 6.15 and 6.16 shows the results obtained for BugLocator across the 26 thresholds N and FDS (*i.e.*, file-level granularity). The results reveal that only two reformulation strategies, namely OB+TITLE+CODE and OB+EB+TITLE+CODE, retrieve more code artifacts in top-N compared to no reformulation. OB+TITLE+CODE's improvement reaches 4.4% (on average) in terms of HITS@N (6.8%/21.9% avg. MRR/MAP improvement), and OB+EB+TITLE+CODE's improvement over the initial queries is minimal (*i.e.*, 0.2% avg. HITS@N and 7.5% avg. MAP improvement, and 7.3% avg. MRR deterioration). Further,

[8]See the replication package for the detailed MRR/MAP results [84].

none of these improvements are statistically significant (Mann-Whitney, 5% significance level) and these strategies present low consistency level across thresholds (*i.e.*, they improve HITS@N for 14 and 8 thresholds N while deteriorating it for 12 and 14 N, respectively). The remaining 29 reformulation strategies lead to HITS@N deterioration by 0.7% - 47.4% (*i.e.*, they are *ineffective*). Thirteen of these are *very-ineffective*, TITLE, S2R, and CODE being the ones with the highest negative impact in practice, given their *high*, *moderate*, and *somewhat* applicability, respectively. S2R and CODE are *ineffective* for all thresholds N, and TITLE for 24 N values. In fact, all *very-ineffective* strategies never retrieve more buggy documents than when using no reformulation for each one of the 26 thresholds (see Table 6.15).

The *highly-applicable* strategies OB+TITLE and OB, and the *moderately-applicable* strategies OB+S2R+TITLE, OB+S2R, and S2R+TITLE, are *somewhat- ineffective*, and lead to deterioration for 15 or more thresholds. OB+S2R, OB, and S2R+TITLE, always lead to deterioration for each threshold N. From all the strategies that lead to deterioration, only OB+CODE, OB+EB+CODE, OB+S2R+TITLE's deterioration is not statistically significant, compared to no reformulation (Mann-Whitney, 5% significance level). The results indicate that OB+TITLE+CODE, OB+EB+TITLE+CODE, OB+CODE, OB+EB +CODE, and OB+S2R+TITLE are nearly as effective as no reformulation, despite their corresponding improvement or deterioration level.

We conclude that OB+TITLE+CODE is the best strategy for BugLocator, as it leads to TRBL improvement with respect to the initial queries (in terms of HITS@N, MRR, and MAP). The downside of this strategy is its *somewhat* applicability and *low* consistency across thresholds N. The results indicate that BugLocator can retrieve the buggy code artifacts within the top-N candidates even if bug reports (used as input queries) contain noisy information. In other words, BugLocator is very robust to noisy queries, and query reduction has little effect on TRBL.

Table 6.15: TRBL performance of each reformulation strategy when using BugLocator.

Reformulation strategy	**# of queries**	**HITS@N**			**Thresh.**		
		No reform.	**Reform.**	**Improv.**	**E**	**N**	**I**
OB+TITLE+CODE	174.7	33.1 (18.9%)	33.7 (19.5%)	4.4%	14	0	12
OB+EB+TITLE+CODE	36.0	9.7 (26.1%)	9.2 (25.7%)	0.2%	8	4	14
OB+CODE	174.7	33.1 (18.9%)	31.6 (18.4%)	-0.7%	12	2	12
OB+S2R+TITLE	309.1	47.3 (15.5%)	46.1 (15.1%)	-2.1%	8	3	15
OB+TITLE	552.4	91.3 (16.8%)	85.5 (15.8%)	-6.0%	6	2	18
OB+EB+CODE	36.0	9.7 (26.1%)	8.4 (23.4%)	-8.4%	8	3	15
OB+S2R+TITLE+CODE	117.4	21.3 (18.0%)	19.2 (16.3%)	-8.9%	4	4	18
OB+S2R	309.1	47.3 (15.5%)	41.5 (13.6%)	-11.9%	0	0	26
EB+TITLE+CODE	38.0	9.7 (24.7%)	8.0 (21.3%)	-12.2%	5	3	18
OB+EB+S2R+TITLE+CODE	25.0	6.8 (26.9%)	5.9 (23.1%)	-13.1%	2	8	16
OB	552.4	91.3 (16.8%)	78.9 (14.6%)	-13.2%	0	0	26
S2R+TITLE	311.2	48.0 (15.6%)	40.7 (13.3%)	-15.1%	1	0	25
OB+EB+TITLE	113.5	23.1 (20.6%)	19.5 (17.5%)	-15.8%	0	1	25
OB+S2R+CODE	117.4	21.3 (18.0%)	17.6 (15.0%)	-15.9%	4	2	20
TITLE+CODE	182.1	35.6 (19.5%)	29.0 (16.1%)	-17.0%	0	0	26
EB+TITLE	118.7	23.8 (20.4%)	18.9 (16.3%)	-20.2%	0	0	26
OB+EB+S2R+TITLE	66.7	13.4 (20.6%)	10.8 (16.5%)	-20.3%	0	2	24
S2R+TITLE+CODE	117.7	21.5 (18.1%)	17.0 (14.3%)	-20.4%	0	1	25
OB+EB+S2R	66.7	13.4 (20.6%)	10.7 (16.5%)	-20.6%	0	0	26
OB+EB+S2R+CODE	25.0	6.8 (26.9%)	5.4 (20.9%)	-21.0%	2	8	16
OB+EB	113.5	23.1 (20.6%)	18.1 (16.2%)	-21.8%	0	0	26
TITLE	571.7	96.4 (17.1%)	74.1 (13.2%)	-22.9%	0	0	26
EB+S2R+TITLE+CODE	25.0	6.8 (26.9%)	4.7 (18.4%)	-30.4%	0	4	22
EB+S2R+TITLE	67.5	13.9 (21.1%)	9.4 (14.3%)	-32.6%	0	0	26
S2R+CODE	117.7	21.5 (18.1%)	13.8 (11.7%)	-34.9%	0	0	26
EB+CODE	38.0	9.7 (24.7%)	5.9 (15.3%)	-37.0%	0	0	26
EB+S2R+CODE	25.0	6.8 (26.9%)	4.2 (16.3%)	-37.6%	0	2	24
CODE	177.2	34.8 (19.6%)	20.2 (11.6%)	-40.1%	0	0	26
EB	118.7	23.8 (20.4%)	12.7 (11.0%)	-46.8%	0	0	26
S2R	309.0	47.2 (15.5%)	25.0 (8.3%)	-46.9%	0	0	26
EB+S2R	67.5	13.9 (21.1%)	7.4 (11.3%)	-47.4%	0	0	26

Average # of queries and HITS@N values across FDS and the 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). Strategies sorted by avg. HITS@N improvement (Improv). None of the strategies achieve a statistically-significant higher HITS@N, compared to no reformulation (Mann-Whitney, 5% significance level). Number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

Table 6.16: Categorization of each reformulation strategy when using BugLocator.

Applicability	VE	SE	SI	VI
H			O+T (-6.0%) O (-13.2%)	T (-22.9%)
M			O+S+T (-2.1%) O+S (-11.9%) S+T (-15.1%)	S (-46.9%)
S		O+T+C (4.4%)	O+C (-0.7%) T+C (-17.0%)	C (-40.1%)
L		O+E+T+C (0.2%)	O+E+C (-8.4%) O+S+T+C (-8.9%) E+T+C (-12.2%) O+E+S+T+C (-13.1%) O+E+T (-15.8%) O+S+C (-15.9%) E+T (-20.2%) O+E+S+T (-20.3%) S+T+C (-20.4%)	O+E+S (-20.6%) O+E+S+C (-21.0%) O+E (-21.8%) E+S+T+C (-30.4%) E+S+T (-32.6%) S+C (-34.9%) E+C (-37.0%) E+S+C (-37.6%) E (-46.8%) E+S (-47.4%)

In parenthesis, average HITS@N improvement across FDS and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Somewhat Ineffective (**SI**), and Very Ineffective (**VI**). The strategies in **green** belong to the *effective* category and the strategies in **red** to the *ineffective* category.
Information types: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

We experimented with the two scoring components of BugLocator in order to know which one is more robust to noisy query terms. The first component (*i.e.*, rVSM) computes the similarity score between the query and a file by using a VSM-based similarity and a boost factor for the file, according to its length. The second component (*i.e.*, SimiScore) computes the similarity score between the query and a file by using a VSM-based similarity between the query and past bug reports that lead to changes in the file. The full version of BugLocator combines the resulting similarity scores from both components in a linear fashion, giving a 0.8 weight to rVSM and 0.2 weight to SimiScore. We found that using rVSM alone leads to five *effective* reformulation strategies (*i.e.*, they improve HITS@N by 5.9%-14.2% on average, compared to no reformulation), and using SimiScore alone leads to twenty-two *effective* strategies (*i.e.*, they improve HITS@N by 0.2%-55% on average, with respect to no reformulation). In addition, we found that the HITS@N achieved by the initial queries when using rVSM is substantially higher than when using SimiScore (*i.e.*, 17.8% vs 8.9%, on average across reformulation strategies). When both are combined (*i.e.*, full BugLocator), the HITS@N achieved by the initial queries reaches 20.5% on average. These results mean that rVSM achieves such a high performance with the initial queries that the reformulations have little effect. Hence, rVSM is more robust to noisy information in the queries than SimiScore. We conjecture that increasing the frequency of the terms in the reformulated queries proportionally to their frequency in the full bug report can lead to higher retrieval improvement when using rVSM. Verifying this conjecture is part of our future research agenda.

6.4.4 Performance for BRTracer

The results for BRTracer (see Tables 6.17 and 6.18) reveal that 14 (out of 31) strategies improve TRBL with respect to no reformulation. OB+EB+S2R is the reformulation strategy with the highest average HITS@N improvement compared to no reformulation (*i.e.*, 16.5%). The improvement is statistically significant, according to the Mann-Whitney test (p-value$<$ 5%). This strategy also achieves 44.7%/35.4% avg. MRR/MAP improvement, and improves HITS@N for 20 thresholds while deteriorating it for only one. The downside of this strategy is its *low* applicability (see Table 6.18), as only ∼55 queries (out of ∼474[9]) can be reformulated with it, on average. OB+TITLE is the only *highly-applicable* strategy that achieves avg. HITS@N improvement (*i.e.*, 0.9%). However, the improvement is not statistically significant (Mann-Whitney, 5% significance level), and this strategy improves and deteriorates HITS@N for an equal number of thresholds (*i.e.*, 13 N). Among the *moderately-applicable* strategies, OB+S2R+TITLE is the one with the highest average HITS@N improvement (*i.e.*, 15.6%),

[9]The # of queries for TITLE in Table 6.17 represents the avg. total # of queries for BRTracer.

Table 6.17: TRBL performance of each reformulation strategy when using BRTracer.

Reformulation strategy	**# of queries**	**HITS@N**			**Thresh.**		
		No reform.	**Reform.**	**Improv.**	**E**	**N**	**I**
OB+EB+S2R*	55.2	10.7 (19.6%)	12.2 (22.5%)	16.5%	20	5	1
OB+S2R+TITLE*	255.5	51.2 (20.6%)	58.5 (23.4%)	15.6%	26	0	0
OB+EB+S2R+TITLE*	55.2	10.7 (19.6%)	12.0 (22.2%)	14.2%	20	2	4
OB+S2R+TITLE+CODE*	92.8	25.6 (28.3%)	27.1 (29.8%)	7.7%	17	4	5
OB+S2R*	255.5	51.2 (20.6%)	54.5 (21.9%)	7.7%	21	2	3
S2R+TITLE*	256.2	51.8 (20.7%)	54.4 (21.7%)	6.4%	17	2	7
EB+S2R+TITLE	55.3	10.8 (19.7%)	11.1 (20.7%)	5.5%	18	2	6
OB+EB	93.6	20.4 (22.2%)	21.0 (22.7%)	4.6%	13	3	10
OB+TITLE+CODE	139.3	39.2 (29.0%)	40.1 (29.2%)	2.8%	13	2	11
S2R+TITLE+CODE	93.1	25.8 (28.4%)	25.9 (28.3%)	2.7%	12	3	11
OB+EB+TITLE	93.6	20.4 (22.2%)	20.6 (22.1%)	1.9%	11	3	12
EB+S2R	55.3	10.8 (19.7%)	10.7 (19.6%)	1.8%	12	3	11
OB+TITLE	458.3	99.8 (22.4%)	99.8 (22.3%)	0.9%	13	0	13
OB+S2R+CODE	92.8	25.6 (28.3%)	25.3 (27.9%)	0.4%	10	2	14
TITLE	474.2	104.1 (22.6%)	99.3 (21.3%)	-3.7%	9	0	17
OB	458.3	99.8 (22.4%)	94.7 (21.2%)	-4.5%	5	4	17
OB+CODE	139.3	39.2 (29.0%)	37.2 (27.3%)	-4.6%	6	0	20
TITLE+CODE	145.0	40.9 (29.1%)	37.7 (26.5%)	-7.1%	4	2	20
OB+EB+S2R+CODE	16.7	5.7 (34.2%)	5.1 (30.0%)	-12.2%	1	11	14
EB+TITLE	98.0	20.8 (21.6%)	18.0 (18.6%)	-13.4%	2	1	23
OB+EB+S2R+TITLE+CODE	16.7	5.7 (34.2%)	4.9 (28.7%)	-16.0%	1	11	14
OB+EB+TITLE+CODE	25.9	9.0 (34.2%)	7.7 (27.8%)	-20.0%	3	2	21
EB+S2R+CODE	16.7	5.7 (34.2%)	4.4 (26.7%)	-21.0%	0	8	18
OB+EB+CODE	25.9	9.0 (34.2%)	7.5 (27.2%)	-21.5%	3	1	22
EB+S2R+TITLE+CODE	16.7	5.7 (34.2%)	4.5 (26.4%)	-22.3%	1	3	22
S2R+CODE	93.1	25.8 (28.4%)	19.3 (21.0%)	-22.8%	3	0	23
EB+TITLE+CODE	27.9	9.0 (31.6%)	7.0 (23.8%)	-24.5%	1	2	23
EB	98.0	20.8 (21.6%)	14.9 (15.5%)	-28.3%	0	0	26
S2R	254.1	51.0 (20.6%)	33.9 (13.6%)	-32.1%	0	0	26
EB+CODE	27.9	9.0 (31.6%)	6.3 (20.7%)	-34.5%	1	1	24
CODE	142.6	40.5 (29.4%)	24.1 (17.1%)	-40.7%	0	0	26

Average # of queries and HITS@N values across FDS and the 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). Strategies sorted by average HITS@N improvement (Improv). All strategies marked with * achieve a statistically-significant higher HITS@N, compared to no reformulation (Mann-Whitney, p-value< 5%). Number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

Table 6.18: Categorization of each reformulation strategy when using BRTracer.

		Effectiveness			
		VE	**SE**	**SI**	**VI**
Applicability	**H**		O+T (0.9%)	T (-3.7%) O (-4.5%)	
	M		O+S+T (15.6%) O+S (7.7%) S+T (6.4%)		S (-32.1%)
	S		O+T+C (2.8%)	O+C (-4.6%) T+C (-7.1%)	C (-40.7%)
	L		O+E+S (16.5%) O+E+S+T (14.2%) O+S+T+C (7.7%) E+S+T (5.5%) O+E (4.6%) S+T+C (2.7%) O+E+T (1.9%) E+S (1.8%) O+S+C (0.4%)	O+E+S+C (-12.2%) E+T (-13.4%) O+E+S+T+C (-16.0%) O+E+T+C (-20.0%)	E+S+C (-21.0%) O+E+C (-21.5%) E+S+T+C (-22.3%) S+C (-22.8%) E+T+C (-24.5%) E (-28.3%) E+C (-34.5%)

In parenthesis, average HITS@N improvement across FDS and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Somewhat Ineffective (**SI**), and Very Ineffective (**VI**). The strategies in **green** belong to the *effective* category and the strategies in **red** to the *ineffective* category.
Information types: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

and it is the only strategy of all that consistently improves HITS@N for all thresholds. This strategy also leads to 12.8%/14.2% avg. MRR/MAP improvement, across all thresholds N and FDS. Conversely, out of the 31 strategies, 17 of them lead to HITS@N deterioration, and nine of them fall in the *very-ineffective* category, including S2R and CODE, which have the highest impact since they are *somewhat-* and *moderately-applicable*, respectively.

We conclude that OB+S2R+TITLE is the best reformulation strategy when using BRTracer, since it is *effective*, *moderately-applicable*, and it *consistently* retrieves more buggy code documents than no reformulation across all 26 thresholds N. The results indicate that BRTracer is more robust to noisy queries, compared to Lucene and Lobster, since only six strategies achieve higher HITS@N (vs. no reformulation) with statistical significance. Remember that BRTracer is an extension of BugLocator, consequently, BRTracer's robustness comes from BugLocator.

6.4.5 Performance for Locus

Tables 6.19 and 6.20 show the results obtained for Locus across the 26 thresholds N and FDS (*i.e.*, file-level granularity). Eleven strategies are *effective*, *i.e.*, they improve HITS@N by 0.1% to 36.1% (on average) with respect to no reformulation. The most effective strategies belong to the *low-applicability* category, OB+EB+S2R being the strategy that achieves the highest avg. improvement in terms of HITS@N (*i.e.*, 36.1%). This strategy consistently improves HITS@N for 24 thresholds, while deteriorating it for only one. All *highly-applicable* strategies are *effective* with statistical significance (Mann-Whitney, p-value$<$ 5%) and fall in the *somewhat-effective* category (with 2.1% - 15.5% avg. HITS@N, and 31.1%/45% - 30.7%/43.9% avg. MRR/MAP improvement). However, only OB+TITLE is *effective* for all 26 thresholds N. Among the *moderately-applicable* strategies, OB+S2R+TITLE is the only one that leads to a statistically significant HITS@N improvement (*i.e.*, 10.2% on average) for 25 thresholds N. This strategy also achieves 23.2%/25.7% avg. MRR/MAP improvement. Conversely, the 20 remaining strategies retrieve the buggy code artifacts within top-N for fewer queries than when using no reformulation (*i.e.*, they are *ineffective*). Eight of them are *very-ineffective* because they lead to more than 20.6% HITS@N deterioration, and consistently deteriorate HITS@N for a large amount of thresholds (*i.e.*, 18, 25, or 26 - see Table 6.19). S2R and CODE belong to this subset and are the strategies with the most negative impact in practice, given its applicability level (*moderate* and *somewhat*, respectively).

We conclude that OB+TITLE is the best reformulation strategy when using Locus, since it is *effective*, *highly-applicable*, and it *consistently* retrieves the buggy code documents within the top-N results for more cases across all 26 thresholds N (compared to no reformulation).

Table 6.19: TRBL performance of each reformulation strategy when using Locus.

Reformulation strategy	**# of queries**	**HITS@N**			**Thresh.**		
		No reform.	**Reform.**	**Improv.**	**E**	**N**	**I**
OB+EB+S2R*	26.9	7.3 (26.2%)	9.2 (34.5%)	36.1%	24	1	1
OB+EB+S2R+TITLE*	26.9	7.3 (26.2%)	8.9 (33.6%)	33.3%	23	1	2
OB+EB+TITLE*	40.6	13.6 (33.3%)	16.8 (41.6%)	25.4%	24	1	1
OB+EB*	40.6	13.6 (33.3%)	16.5 (41.2%)	24.2%	24	0	2
OB+TITLE*	190.7	60.0 (32.1%)	69.3 (37.0%)	15.5%	26	0	0
OB+S2R+TITLE*	124.4	37.0 (30.4%)	40.7 (33.4%)	10.2%	25	1	0
EB+S2R+TITLE	26.9	7.3 (26.2%)	7.6 (27.8%)	9.1%	14	5	7
TITLE*	199.6	62.7 (32.0%)	64.8 (33.1%)	3.5%	20	2	4
OB*	190.7	60.0 (32.1%)	61.2 (32.8%)	2.1%	17	5	4
OB+TITLE+CODE	92.5	29.4 (32.9%)	29.4 (32.5%)	1.1%	11	2	13
S2R+TITLE	125.6	37.2 (30.3%)	37.5 (30.1%)	0.1%	11	2	13
OB+EB+TITLE+CODE	16.5	7.3 (43.7%)	7.0 (43.4%)	-0.3%	6	14	6
EB+TITLE	42.1	13.9 (32.9%)	13.7 (32.4%)	-1.4%	8	4	14
OB+EB+CODE	16.5	7.3 (43.7%)	6.7 (42.5%)	-2.1%	6	14	6
OB+EB+S2R+CODE	10.5	4.1 (34.1%)	3.8 (32.4%)	-3.7%	1	19	6
OB+S2R	124.4	37.0 (30.4%)	35.4 (28.9%)	-4.3%	7	2	17
OB+EB+S2R+TITLE+CODE	10.5	4.1 (34.1%)	3.7 (32.0%)	-4.5%	1	18	7
EB+S2R+TITLE+CODE	10.5	4.1 (34.1%)	3.6 (31.2%)	-6.2%	0	18	8
OB+CODE	92.5	29.4 (32.9%)	26.8 (29.6%)	-7.8%	6	2	18
OB+S2R+TITLE+CODE	63.4	20.3 (32.9%)	18.1 (29.5%)	-10.0%	3	2	21
OB+S2R+CODE	63.4	20.3 (32.9%)	17.0 (27.7%)	-15.4%	1	1	24
EB+TITLE+CODE	17.5	7.3 (40.8%)	6.0 (34.0%)	-16.3%	0	7	19
TITLE+CODE	98.1	30.5 (32.1%)	24.5 (25.3%)	-18.6%	2	3	21
EB+S2R	26.9	7.3 (26.2%)	5.5 (20.3%)	-21.3%	0	3	23
S2R+TITLE+CODE	64.4	20.3 (32.4%)	15.6 (24.9%)	-22.6%	1	0	25
EB+S2R+CODE	10.5	4.1 (34.1%)	3.1 (25.1%)	-29.7%	0	8	18
S2R	125.6	37.2 (30.3%)	23.4 (19.1%)	-36.5%	0	0	26
EB+CODE	17.5	7.3 (40.8%)	4.2 (22.7%)	-44.4%	0	0	26
S2R+CODE	64.4	20.3 (32.4%)	11.0 (17.6%)	-45.0%	0	0	26
CODE	98.1	30.5 (32.1%)	16.1 (16.9%)	-46.7%	0	0	26
EB	42.1	13.9 (32.9%)	7.6 (17.5%)	-46.9%	0	0	26

Average # of queries and HITS@N values across FDS and the 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). Strategies sorted by average HITS@N improvement (Improv). All strategies marked with * achieve a statistically-significant higher HITS@N, compared to no reformulation (Mann-Whitney, p-value< 5%). Number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

Table 6.20: Categorization of each reformulation strategy when using Locus.

Applicability	Effectiveness: VE	SE	SI	VI
H		O+T (15.5%) T (3.5%) O (2.1%)		
M		O+S+T (10.2%) S+T (0.1%)	O+S (-4.3%)	S (-36.5%)
S		O+T+C (1.1%)	O+C (-7.8%) T+C (-18.6%)	C (-46.7%)
L	O+E+S (36.1%) O+E+S+T (33.3%) O+E+T (25.4%) O+E (24.2%)	E+S+T (9.1%)	O+E+T+C (-0.3%) E+T (-1.4%) O+E+C (-2.1%) O+E+S+C (-3.7%) O+E+S+T+C (-4.5%) E+S+T+C (-6.2%) O+S+T+C (-10.0%) O+S+C (-15.4%) E+T+C (-16.3%)	E+S (-21.3%) S+T+C (-22.6%) E+S+C (-29.7%) E+C (-44.4%) S+C (-45.0%) E (-46.9%)

In parenthesis, average HITS@N improvement across FDS and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Somewhat Ineffective (**SI**), and Very Ineffective (**VI**). The strategies in **green** belong to the *effective* category and the strategies in **red** to the *ineffective* category.
Information types: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

6.4.6 Analysis across Code Granularities

We analyze the performance of the reformulation strategies across the three code granularities for Lucene, which is the only TRBL approach among the five that we could use with all granularities. Tables 6.21 and 6.22 reveal that the strategies have a different performance across code granularities.

Regarding class-level granularity (*i.e.*, CDS), twelve reformulation strategies achieve a 22.3% - 50.2% avg. HITS@N improvement with respect to no reformulation (*i.e.*, they are *very-effective*), and nine more strategies achieve 0.5% - 20.9% avg. HITS@N improvement (*i.e.*, they are *somewhat-effective*). The remaining 10 strategies lead to HITS@N deterioration, hence, they are *ineffective*. Lucene performs best when using the *moderately-*

Table 6.21: TRBL performance improvement across datasets when using Lucene.

Reformulation strategy	**CDS**		**FDS**		**MDS**		**All data sets**	
	|Q|	**Improv**	**|Q|**	**Improv**	**|Q|**	**Improv**	**|Q|**	**Improv**
OB+EB+TITLE	37.4	22.3%	89.6	34.2%	28.1	50.0%	155.2	30.3%
OB+TITLE	170.8	44.0%	472.7	26.4%	120.1	11.7%	763.6	29.3%
OB+S2R+TITLE	97.5	50.2%	263.3	20.8%	48.1	34.8%	409.0	29.2%
OB+EB+S2R+TITLE	21.4	7.1%	51.6	34.8%	17.3	90.2%	90.3	28.6%
OB+EB	37.4	19.4%	89.6	30.9%	28.1	35.6%	155.2	26.1%
OB+TITLE+CODE	80.4	40.3%	151.2	14.3%	53.5	39.4%	285.1	25.6%
OB+EB+S2R	21.4	8.7%	51.6	28.8%	17.3	86.3%	90.3	24.9%
OB+S2R	97.5	49.2%	263.3	10.9%	48.1	30.8%	409.0	22.4%
TITLE	172.6	31.5%	488.1	16.6%	122.8	14.2%	783.5	20.1%
S2R+TITLE	98.0	46.7%	263.8	4.1%	48.1	47.2%	409.9	19.1%
EB+TITLE	38.9	29.4%	94.3	14.2%	28.1	18.2%	161.3	17.3%
OB	169.8	31.7%	472.7	13.5%	120.1	-4.0%	762.6	16.2%
OB+CODE	80.4	27.0%	151.2	3.1%	53.5	34.6%	285.1	14.7%
EB+S2R+TITLE	21.9	-6.7%	51.6	15.5%	17.3	131.2%	90.8	14.4%
TITLE+CODE	80.4	31.1%	156.8	-4.1%	55.3	36.0%	292.5	12.4%
OB+EB+TITLE+CODE	17.6	5.2%	25.7	8.7%	12.0	21.3%	55.3	6.7%
OB+S2R+TITLE+CODE	50.5	17.3%	99.1	2.4%	27.2	35.2%	176.8	6.2%
OB+EB+CODE	17.6	4.5%	25.7	2.3%	12.0	20.5%	55.3	3.3%
OB+S2R+CODE	50.5	20.9%	99.1	-5.4%	27.2	34.2%	176.8	3.2%
S2R+TITLE+CODE	50.5	1.9%	99.2	-7.5%	27.2	35.7%	176.9	-4.1%
OB+EB+S2R+TITLE+CODE	13.0	-30.1%	16.6	8.5%	7.6	-1.3%	37.1	-9.8%
EB+S2R	21.9	0.5%	51.6	-13.9%	17.3	-13.9%	90.8	-10.6%
EB	37.5	24.9%	94.3	-18.2%	27.1	-51.4%	158.9	-10.6%
OB+EB+S2R+CODE	13.0	-30.1%	16.6	0.1%	7.6	-2.2%	37.1	-15.1%
EB+TITLE+CODE	17.6	-17.5%	27.7	-22.6%	12.0	1.3%	57.3	-20.0%
S2R+CODE	50.5	-32.3%	99.2	-26.3%	27.2	11.6%	176.9	-27.9%
EB+CODE	17.6	-24.8%	27.7	-34.0%	12.0	-13.7%	57.3	-29.3%
S2R	98.0	-17.6%	262.4	-34.0%	48.1	-46.5%	408.5	-30.6%
CODE	76.2	-39.5%	153.4	-44.9%	54.8	-1.4%	284.5	-37.2%
EB+S2R+TITLE+CODE	13.0	-55.8%	16.6	-33.2%	7.6	-1.3%	37.1	-41.5%
EB+S2R+CODE	13.0	-59.0%	16.6	-46.4%	7.6	-2.2%	37.1	-49.8%

|Q|: Number of reduced queries. Average values across thresholds N={5, 6, 7, ..., 30}.

Strategies sorted by avg. HITS@N improvement for all data sets.

applicable strategies OB+S2R+TITLE, OB+S2R, and S2R+TITLE (*i.e.*, 50.2%, 49.2%, and 46.7% avg. HITS@N improvement, respectively), and when using the *highly-applicable* strategies OB+TITLE, OB, and TITLE (*i.e.*, 44%, 31.7%, and 31.5% avg. HITS@N improvement, respectively). These six strategies consistently improve HITS@N for all 26 thresholds (see Table 6.22 - CDS columns). The worst strategies are EB+S2R+CODE and EB+S2R+TITLE+CODE, which consistently fail to retrieve the buggy code documents within the top-N results for 24 thresholds N. These two and five more strategies belong to the *very-ineffective* category. S2R and CODE are the strategies with the highest negative impact in practice, according to their level of applicability (*i.e.*, *moderate* and *somewhat*, respectively) and performance (*i.e.*, they lead to more than 15% HITS@N deterioration, and consistently achieve deterioration for 20 and 26 thresholds, respectively). We consider OB+TITLE as the best reformulation strategy for Lucene when retrieving buggy classes.

In the case of file-level granularity (*i.e.*, FDS), we found that five strategies achieve more than 21.4% average HITS@N improvement, namely OB+EB+S2R+TITLE, OB+EB+TITLE, OB+EB, OB+EB+S2R, and OB+TITLE (*i.e.*, 34.8%, 34.2%, 30.9%, 28.8%, and 26.4% average improvement, respectively), hence, they belong to the *very-effective* category and they consistently improve HITS@N for all thresholds (see Table 6.22 - FDS columns). Among these, OB+TILE is the only *highly-applicable* strategy. Fourteen more strategies lead to HITS@N improvement by 0.1% - 20.8%, on average, including all *moderately-applicable* strategies, except S2R. The remaining 12 strategies are *ineffective*. Among these, seven lead to more than 20.6% HITS@N deterioration (*i.e.*, they are *very-ineffective*), including S2R and CODE, which consistently fail to retrieve the buggy code documents for all thresholds. We conclude that OB+TITLE is the best reformulation strategy for Lucene when retrieving buggy code files.

As for the method-level granularity (*i.e.*, MDS), 14 strategies are *very-effective* (*i.e.*, their HITS@N improvement is greater than 21.4%). Among these, five strategies achieve more

Table 6.22: *Consistency* results across datasets when using Lucene.

Reformulation Strategy	**CDS**				**FDS**				**MDS**			
	E	**N**	**I**	**H@N**	**E**	**N**	**I**	**H@N**	**E**	**N**	**I**	H@N
OB+EB+TITLE	25	1	0	22.3%	26	0	0	34.2%	22	1	3	50.0%
OB+TITLE	26	0	0	44.0%	26	0	0	26.4%	19	1	6	11.7%
OB+S2R+TITLE	26	0	0	50.2%	26	0	0	20.8%	21	5	0	34.8%
OB+EB+S2R+TITLE	11	7	8	7.1%	26	0	0	34.8%	13	12	1	90.2%
OB+EB	24	2	0	19.4%	26	0	0	30.9%	19	4	3	35.6%
OB+TITLE+CODE	25	0	1	40.3%	25	1	0	14.3%	25	1	0	39.4%
OB+EB+S2R	11	7	8	8.7%	26	0	0	28.8%	13	10	3	86.3%
OB+S2R	26	0	0	49.2%	25	1	0	10.9%	20	3	3	30.8%
TITLE	26	0	0	31.5%	24	0	2	16.6%	19	1	6	14.2%
S2R+TITLE	26	0	0	46.7%	13	3	10	4.1%	22	1	3	47.2%
EB+TITLE	26	0	0	29.4%	22	2	2	14.2%	14	4	8	18.2%
OB	26	0	0	31.7%	26	0	0	13.5%	7	4	15	-4.0%
OB+CODE	20	3	3	27.0%	16	5	5	3.1%	24	0	2	34.6%
EB+S2R+TITLE	6	4	16	-6.7%	14	4	8	15.5%	13	9	4	131.2%
TITLE+CODE	21	2	3	31.1%	3	6	17	-4.1%	25	1	0	36.0%
OB+EB+TITLE+CODE	10	6	10	5.2%	14	9	3	8.7%	8	16	2	21.3%
OB+S2R+TITLE+CODE	9	5	12	17.3%	17	2	7	2.4%	17	5	4	35.2%
OB+EB+CODE	10	6	10	4.5%	10	9	7	2.3%	7	17	2	20.5%
OB+S2R+CODE	13	2	11	20.9%	9	4	13	-5.4%	17	4	5	34.2%
S2R+TITLE+CODE	9	0	17	1.9%	4	9	13	-7.5%	17	5	4	35.7%
OB+EB+S2R+TITLE+CODE	0	8	18	-30.1%	7	10	9	8.5%	0	25	1	-1.3%
EB+S2R	8	3	15	0.5%	5	2	19	-13.9%	2	14	10	-13.9%
EB	24	1	1	24.9%	0	1	25	-18.2%	1	2	23	-51.4%
OB+EB+S2R+CODE	0	8	18	-30.1%	5	10	11	0.1%	0	24	2	-2.2%
EB+TITLE+CODE	5	5	16	-17.5%	0	3	23	-22.6%	2	19	5	1.3%
S2R+CODE	4	0	22	-32.3%	0	0	26	-26.3%	10	8	8	11.6%
EB+CODE	1	5	20	-24.8%	0	0	26	-34.0%	0	17	9	-13.7%
S2R	5	1	20	-17.6%	0	0	26	-34.0%	0	1	25	-46.5%
CODE	0	0	26	-39.5%	0	0	26	-44.9%	4	7	15	-1.4%
EB+S2R+TITLE+CODE	0	2	24	-55.8%	0	1	25	-33.2%	0	25	1	-1.3%
EB+S2R+CODE	0	2	24	-59.0%	0	0	26	-46.4%	0	24	2	-2.2%

Number of thresholds N (out of 26) for which each reformulation strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**) when using Lucene on each data set. **H@N** is the average HITS@N improvement across 26 thresholds N. Strategies sorted by average HITS@N improvement across all data sets.

than 40% average HITS@N improvement, namely EB+S2R+TITLE, OB+EB+S2R+TITLE, OB+EB+S2R, OB+EB+TITLE, and S2R+TITLE (*i.e.*, 131.2%, 90.2%, 86.3%, 50.0%, and 47.2% avg. HITS@N improvement, respectively), S2R+TITLE being the only *moderately-applicable*. However, from these five strategies, only OB+EB+TITLE and S2R+TITLE improve HITS@N for most thresholds N (*i.e.*, 22 N - see Table 6.22), while deteriorating it for few cases (*i.e.*, 3 N). The strategies OB+CODE and TITLE+CODE are the only ones that lead to improvement for nearly all thresholds (*i.e.*, 25 N). Seven more strategies achieve 1.3% - 21.3% average HITS@N improvement (*i.e.*, they are *somewhat-effective*). The remaining 10 strategies do not lead to any improvement, and among these, S2R and EB are the most ineffective ones, which lead to deterioration for a large number of thresholds (*i.e.*, 25 and 23 N values, respectively). We conclude that S2R+TITLE is the best reformulation strategy for Lucene when retrieving buggy methods.

We observed that the *effective* CODE-based strategies are more common for MDS than for CDS and FDS (10 vs. 8 and 7 strategies, respectively), which indicates that the CODE information in bug reports is more effective for retrieving methods than for retrieving files or classes. We manually analyzed a subset of the MDS queries and their respective buggy code methods (*i.e.*, the gold set) and found that, indeed, code snippets are commonly found in the MDS bug reports and they contain many relevant terms present in the buggy code. This is more common on the queries from the Defects4J systems [128], *i.e.*, Lang, Joda-Time, and Math, than for other systems. Note that the Defects4J bugs were originally collected for software testing research and have three main characteristics [128]: (1) they are clearly related to the source code in the version control system; (2) they are reproducible; and (3) their fixes were done independently of other code changes. This explains (in part) why the code in the Defects4J queries is likely to contain relevant terms with respect to TRBL.

All *highly-* and *moderately-applicable* strategies, except OB and S2R, are among the strategies that achieve the highest HITS@N improvement across the three code granularities.

Among these, OB+TITLE and S2R+TITLE are the best-performing strategies, the former for CDS and FDS, and the latter for MDS (*i.e.*, 44%, 26.4%, and 47.2% avg. HITS@N improvement, respectively). These results provide evidence of how effective it is to combine the TITLE with OB or S2Rs for reformulating the initial queries.

Note that the top-3 most *effective* strategies for MDS achieve a high avg. HITS@N improvement, *i.e.*, greater than 85%, which is substantially higher than the best improvement rates achieved for CDS and FDS (*i.e.*, 50.2% and 34.8%, respectively). These differences come from the large improvement that the MDS strategies achieve for a subset of the thresholds N. For example, EB+S2R+TITLE's avg. HITS@N improvement is 300% for N={13, 21, 22, 23, 24, 25, 26} and 100% for N={12, 27, 28, 29, 30}. These values lead to a high overall average improvement for this strategy. For the remaining thresholds, HITS@N improvement values are similar to the ones from CDS and FDS. Finally, note that for Java systems, we can consider class- and file-level granularities to be somewhat similar. The performance for Lucene across the CDS and FDS data sets indicates that the corpus granularity does not seriously impact the successful reformulation techniques.

6.4.7 Overall Performance

In general, fewer strategies lead to TRBL improvement in terms of HITS@N for BugLocator, BRTracer, and Locus than for Lucene and Lobster (2, 14, and 11 vs. 19 and 25 strategies, respectively). The results indicate that BugLocator, BRTracer, and Locus are less sensitive to noisy queries than Lucene and Lobster, yet the reformulation strategies still lead to TRBL improvement for all of them (*i.e.*, buggy code retrieval in top-N for more cases than without reformulation).

Summarizing across the TRBL techniques, code granularities, and thresholds N (see Tables 6.23, 6.24, and 6.25), the results show that 18 query reformulation strategies achieve improvement in terms of HITS@N, MRR, and MAP. Among these 18, all *highly-applicable* strategies (*i.e.*, OB, TITLE, and OB+TITLE) improve TRBL by 16.6% - 25.6% HITS@N, and by 48.9% - 73.9% (51.6% - 69.8%) MRR (MAP), on average. OB+TITLE falls in the *very-effective* category, and TITLE and OB in the *somewhat-effective* category (see Table 6.25). OB+TITLE is the second best strategy (after OB+S2R+TITLE, see below) in terms of the total number of thresholds N for which there is HITS@N improvement, *i.e.*, 97 thresholds N (out of 130 total, on aggregate across TRBL approaches - see Table 6.23), versus 31 thresholds N for which there is HITS@N deterioration. This strategy also retrieves the buggy code documents within the top-N results for 25.6% more queries (on average), compared to no reformulation and achieves 58.6%/60.6% avg. MRR/MAP improvement.

In addition, among the *effective* strategies, three *moderately-applicable* (*i.e.*, OB+S2R+ TITLE, OB+S2R, and S2R+TITLE) stand out, since they achieve between 22.6% and 31.4% avg. HITS@N improvement, and between 42.3% (49.1%) and 54.4% (57.3%) avg. MRR/MAP improvement. All three strategies are *very-effective*, and OB+S2R+TITLE is the best strategy in terms of avg. HITS@N improvement (*i.e.*, 31.4%) and the number of thresholds N for which there is HITS@N improvement, *i.e.*, 108 thresholds N (out of 130, on aggregate) versus 15 thresholds N for which there is HITS@N deterioration.

The remaining 12 *effective* strategies are less applicable and achieve between 2% and 41.7% improvement in terms of HITS@N, and between 13.3% (9.3%) and 201.1% (148.8%) MRR/MAP improvement, on average. OB+EB+TITLE is the strategy that performs best in terms of HITS@N (*i.e.*, 41.7% avg. improvement), but at the same time, its applicability is rather *low*. All strategies with positive HITS@N improvement, excluding OB+EB+CODE and OB+S2R+CODE, achieve a statistically significant HITS@N improvement, compared to no reformulation (Mann-Whitney, p-value$<$ 5%).

Overall, OB+S2R+TITLE is the *moderately-applicable* reformulation strategy that achieves the highest TRBL performance in terms of HITS@N across TRBL techniques, as it leads to the retrieval of the buggy code artifacts within the top-N results (N={5, 6, ..., 30}) for 28.2%

Table 6.23: TRBL performance when using all five TRBL techniques.

Reformulation strategy	**# of queries**	**HITS@N**			**Thresh.**		
		No reform.	**Reform.**	**Improv.**	**E**	**N**	**I**
OB+EB+TITLE	86.3	20.7 (26.4%)	23.5 (39.2%)	41.7%	79	13	38
OB+EB	86.3	20.7 (26.4%)	22.9 (38.8%)	39.8%	81	11	38
OB+S2R+TITLE	228.5	47.2 (22.3%)	55.1 (28.2%)	31.4%	108	7	15
EB+TITLE	90.1	21.2 (24.9%)	21.2 (31.8%)	28.2%	53	13	64
OB+TITLE+CODE	141.8	35.0 (25.7%)	39.4 (30.9%)	25.8%	90	4	36
OB+TITLE	399.9	86.3 (22.7%)	98.2 (27.9%)	25.6%	97	2	31
S2R+TITLE	229.5	47.6 (22.4%)	50.8 (26.4%)	23.2%	78	7	45
OB+S2R	228.5	47.2 (22.3%)	51.0 (26.1%)	22.6%	77	7	46
TITLE	412.9	89.7 (22.7%)	93.1 (26.2%)	18.8%	81	2	47
OB+CODE	141.8	35.0 (25.7%)	36.4 (29.0%)	18.3%	76	4	50
OB	399.7	86.3 (22.7%)	89.6 (25.7%)	16.6%	74	9	47
OB+EB+S2R	59.8	13.1 (22.5%)	14.5 (25.7%)	14.2%	68	32	30
OB+EB+S2R+TITLE	59.8	13.1 (22.5%)	14.6 (25.7%)	14.0%	68	31	31
TITLE+CODE	147.0	36.4 (25.7%)	35.6 (27.5%)	12.3%	56	6	68
OB+EB+TITLE+CODE	28.8	8.9 (34.8%)	8.8 (40.5%)	11.9%	49	32	49
OB+EB+CODE	28.8	8.9 (34.8%)	8.4 (39.5%)	8.6%	47	31	52
OB+S2R+TITLE+CODE	96.9	23.5 (26.7%)	23.6 (28.4%)	7.5%	58	23	49
OB+S2R+CODE	96.9	23.5 (26.7%)	22.4 (27.0%)	2.0%	43	17	70
S2R+TITLE+CODE	97.2	23.6 (26.7%)	21.6 (26.4%)	-0.3%	40	13	77
EB+TITLE+CODE	30.3	8.9 (33.1%)	7.4 (35.0%)	-0.8%	25	20	85
EB+S2R+TITLE	59.6	13.2 (22.5%)	12.9 (22.4%)	-0.9%	46	37	47
OB+EB+S2R+TITLE+CODE	21.0	5.8 (27.5%)	5.1 (24.1%)	-10.2%	9	67	54
EB	89.6	21.0 (24.9%)	15.2 (21.9%)	-10.2%	20	10	100
OB+EB+S2R+CODE	21.0	5.8 (27.5%)	5.0 (23.8%)	-12.2%	8	66	56
S2R	228.4	47.2 (22.3%)	30.7 (17.0%)	-18.9%	17	9	104
EB+S2R	59.6	13.2 (22.5%)	10.5 (18.0%)	-19.2%	18	33	79
S2R+CODE	97.2	23.6 (26.7%)	16.4 (21.1%)	-20.5%	22	7	101
EB+S2R+TITLE+CODE	21.2	5.9 (27.8%)	4.2 (21.1%)	-23.7%	1	52	77
CODE	143.7	35.8 (26.0%)	21.7 (18.1%)	-27.1%	17	8	105
EB+CODE	29.9	8.7 (32.6%)	5.8 (24.0%)	-27.2%	5	23	102
EB+S2R+CODE	21.2	5.9 (27.8%)	3.8 (18.7%)	-32.7%	0	44	86

Average # of queries and HITS@N values the 3 data sets/granularities, 5 TRBL techniques, and 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). The strategies are sorted by average HITS@N improvement (Improv). Aggregated number of thresholds for which each strategy is Effective (E), Neutral (N), and Ineffective (I), across all TRBL techniques. The total number of thresholds (*i.e.*, E + N + I) is 5 (techniques) x 26 (thresholds) = 130. All strategies with positive improvement, excluding OB+EB+CODE and OB+S2R+CODE, achieve a statistically- significant higher HITS@N, compared to no reformulation (Mann-Whitney, *p*-value< 5%).

Table 6.24: MRR/MAP results for all five TRBL techniques.

Reformulation strategy	**MRR**			**MAP**		
	No ref.	**Ref.**	**Improv.**	**No ref.**	**Ref.**	**Improv.**
OB+EB+TITLE	8.4%	19.7%	196.1%	6.6%	14.6%	148.4%
OB+EB	8.4%	19.9%	201.1%	6.6%	14.5%	148.8%
OB+S2R+TITLE	7.3%	10.0%	48.9%	5.6%	8.2%	55.1%
EB+TITLE	8.0%	15.8%	173.5%	6.3%	12.2%	133.9%
OB+TITLE+CODE	8.2%	14.2%	101.0%	6.3%	11.3%	110.4%
OB+TITLE	7.3%	11.0%	58.6%	5.7%	8.7%	60.6%
S2R+TITLE	7.3%	10.2%	54.4%	5.7%	8.2%	57.3%
OB+S2R	7.3%	9.5%	42.3%	5.6%	7.8%	49.1%
TITLE	7.4%	11.9%	73.9%	5.7%	9.2%	69.8%
OB+CODE	8.2%	12.8%	88.0%	6.3%	10.1%	99.2%
OB	7.4%	10.2%	48.9%	5.7%	8.1%	51.6%
OB+EB+S2R	7.3%	8.3%	23.1%	5.6%	6.3%	21.9%
OB+EB+S2R+TITLE	7.3%	7.8%	13.3%	5.6%	6.0%	13.4%
TITLE+CODE	8.2%	13.0%	85.5%	6.3%	10.1%	90.9%
OB+EB+TITLE+CODE	11.2%	18.1%	97.7%	8.7%	15.5%	101.8%
OB+EB+CODE	11.2%	17.8%	95.3%	8.7%	15.3%	99.4%
OB+S2R+TITLE+CODE	8.8%	10.1%	16.2%	6.7%	7.7%	17.8%
OB+S2R+CODE	8.8%	9.3%	6.7%	6.7%	7.2%	9.3%
S2R+TITLE+CODE	8.8%	9.6%	10.3%	6.7%	7.3%	11.0%
EB+TITLE+CODE	10.7%	16.1%	80.0%	8.4%	14.1%	83.8%
EB+S2R+TITLE	7.3%	7.8%	45.9%	5.6%	5.7%	20.9%
OB+EB+S2R+TITLE+CODE	9.6%	9.4%	89.7%	7.1%	6.9%	29.0%
EB	8.1%	12.9%	130.2%	6.3%	10.2%	101.7%
OB+EB+S2R+CODE	9.6%	9.2%	86.6%	7.1%	6.8%	26.8%
S2R	7.4%	7.0%	-1.1%	5.7%	5.6%	0.4%
EB+S2R	7.3%	6.4%	25.8%	5.6%	4.7%	2.4%
S2R+CODE	8.8%	7.6%	-13.4%	6.7%	5.9%	-11.3%
EB+S2R+TITLE+CODE	9.7%	8.1%	68.7%	7.2%	5.9%	8.6%
CODE	8.4%	8.9%	33.8%	6.5%	6.7%	37.2%
EB+CODE	10.6%	13.9%	60.5%	8.3%	12.6%	66.0%
EB+S2R+CODE	9.7%	7.0%	56.1%	7.2%	5.0%	-3.5%

Average values across the 3 data sets, 5 TRBL techniques, and 26 thresholds N, when using the reformulation strategies (Reform) vs. no reformulation (No reform). The strategies are sorted by avg. HITS@N improvement (*i.e.*, same order as in Table 6.23). All strategies with positive HITS@N improvement, including S2R+TITLE+CODE, achieve a statistically-significant higher MRR/MAP, compared to no reformulation (Mann-Whitney, p-value$< 5\%$).

Table 6.25: Categorization of each strategy when using all five TRBL techniques.

		Effectiveness			
		VE	**SE**	**SI**	**VI**
Applicability	**H**	O+T (25.6%)	T (18.8%) O (16.6%)		
	M	O+S+T (31.4%) S+T (23.2%) O+S (22.6%)		S (-18.9%)	
	S	O+T+C (25.8%)	O+C (18.3%) T+C (12.3%)		C (-27.1%)
	L	O+E+T (41.7%) O+E (39.8%) E+T (28.2%)	O+E+S (14.2%) O+E+S+T (14.0%) O+E+T+C (11.9%) O+E+C (8.6%) O+S+T+C (7.5%) O+S+C (2.0%)	S+T+C (-0.3%) E+T+C (-0.8%) E+S+T (-0.9%) O+E+S+T+C (-10.2%) E (-10.2%) O+E+S+C (-12.2%) E+S (-19.2%) S+C (-20.5%)	E+S+T+C (-23.7%) E+C (-27.2%) E+S+C (-32.7%)

In parenthesis, average HITS@N improvement across the 3 data sets, 5 TRBL techniques, and 26 thresholds N.
Strategies sorted by avg. HITS@N improvement for each *Applicability-Effectiveness* category.
Applicability categories: High (**H**), Moderate (**M**), Somewhat (**S**), and Low (**L**).
Effectiveness categories: Very Effective (**VE**), Somewhat Effective (**SE**), Somewhat Ineffective (**SI**),
and Very Ineffective (**VI**). The strategies in **green** belong to the *effective* category and the strategies in **red**
to the *ineffective* category. *Information types*: OB (**O**), EB (**E**), S2Rs (**S**), TITLE (**T**), and CODE (**C**).

of the queries, on average, which is 31.4% more queries than when using no reformulation at all, where only 22.3% of the queries return the buggy code on the top of the result list. The OB+TITLE strategy achieves comparable (yet lower) results, as it retrieves the buggy code for 25.6% more queries (on average) than without reformulation, while being more applicable in an actual usage scenario - in fact, it is the best *highly-applicable* strategy. Both strategies consistently retrieve the buggy code documents within top-N for more queries (than when using no reformulation) across thresholds N.

We consider OB+TITLE as the best strategy across all TRBL techniques and code granularities, as it is *very-effective*, *highly-applicable*, and *consistent* across different thresholds. Combining the TITLE and OB with the S2Rs from the bug report (if present) leads to higher TRBL effectiveness and comparable consistency, yet it is less applicable in an actual usage scenario. We conclude that among the five types of information from bug reports (*i.e.*, the TITLE, OB, EB, S2Rs, and CODE), the TITLE, OB, and S2Rs are the most effective and practical for improving TRBL in the context of query reformulation. This means that developers should use the terms used in the TITLE, and the ones describing OB and S2Rs (when present) to reformulate an initial query and expect to find the buggy code artifacts in the top of the list for more cases (*i.e.*, between 25.6% and 31.4%, on average), compared to the cases when no reformulation is used.

6.4.8 Discussion

We observed that the summary provided in the bug report TITLE usually contains key terms about the context of the software bug, which are helpful for retrieval (according to the results). The OB usually describes details about the bug, which include specific terms that help narrow down the search space of code documents. In several cases, we found that the title is a succinct description of the observed behavior, which is later expanded in the bug report description. Combining the TITLE and OB for reformulation increases the weight of relevant terms, thus leading to higher retrieval performance. We also observed that the S2Rs usually adds key terms related to the problem, *i.e.*, it gives additional context for retrieval. However, these terms may hinder TRBL if they are not specific enough to the problem or contain extra terms that are present in many documents from the corpus.

To illustrate these observations, consider the bug report #102778 from Eclipse and its respective buggy code file `CodeSnippetParser.java` (see Figure 6.3). The title describes a problem related to "enhanced for statements" in "scrapbook pages". The S2Rs and the CODE snippet provide the information for triggering the problem. Note that they do not state the problem, but provide additional context about it (*i.e.*, "creating a java project" with the given code snippet in a "scrapbook page"). Finally, in the last sentence of the report, the problem is explicitly described (*i.e.*, a "syntax error" thrown by the system – this is the OB). Note that the steps to reproduce are not quite specific to the problem,

Bug report title:
Scrapbook page doesn't work with enhanced for $\boxed{\text{statement}}$ [TITLE]

Bug report description:
Using 3.1, $\boxed{\text{create}}$ a new java project. [S2Rs]
Add a new scrapbook page that $\boxed{\text{contains}}$ this $\boxed{\text{source}}$: [S2Rs]

```
int[] tab = new int[] {1, 2, 3, 4, 5, 6, 7, 8, 9 };
int sum = 0;
for (int i : tab) {
  sum += i;
}
sum [CODE]
```

You get an $\boxed{\text{error}}$ about $\boxed{\text{syntax}}$ $\boxed{\text{error}}$. [OB]

Buggy code file: `CodeSnippetParser.java`

```
[...]
public class CodeSnippetParser extends Parser implements EvaluationConstants {
    [...]
    int lastStatement = -1; // end of last top level statement
    [...]
    /**
    * Creates a new code snippet parser.
    */
    public CodeSnippetParser(ProblemReporter problemReporter,
                            [...] ) {
        [...]
        this.reportOnlyOneSyntaxError = true;
        this.javadocParser.checkDocComment = false;
    }
    [...]
    protected CompilationUnitDeclaration endParse(int act) {
        [...]
        // otherwise it contained the type, [...]
        int start = unitResult.problems[i].getSourceStart();
        [...]
    }
    [...]
    protected void reportSyntaxErrors(boolean isDietParse, int oldFirstToken) {
        [...]
        super.reportSyntaxErrors(isDietParse, oldFirstToken);
    }
    [...]
}
```

Bug report found at `https://bugs.eclipse.org/bugs/show_bug.cgi?id=102778`

The boxed terms represent the terms shared between the report and the file. Each sentence in the report is marked according to its corresponding information type: [TITLE], [CODE], [OB], or [S2Rs].

Figure 6.3: Bug report #102778 from Eclipse and its corresponding buggy file.

i.e., creating a java project that contains some source code is a common task in Eclipse. In this case, the S2Rs contain terms (*e.g.*, “project” or “contain”) that are likely to be present in multiple files within Eclipse, even though, some of them appear in the buggy file (*i.e.*, “create” and “source”).

Using the full bug report from Figure 6.3, as input query to Lucene, would retrieve the buggy file in the 179th position of the result list. Clearly, the query is *low-quality*[10]. Assume the developer inspects the top-5 documents, then reformulates and executes the query, and inspects the next top-5 documents[11]. Using the TITLE, OB, and S2Rs **alone** as reformulation strategies lead to retrieving the file in positions 43, 74, and 2,888, respectively. Using the CODE **alone** fails to retrieve the buggy file because the code snippet does not share any terms with the file. Similar results are obtained for BugLocator, BRTracer, and Locus. These results indicate that the TITLE and OB contain the most useful terms for retrieval (*i.e.*, “statement”, “syntax”, and “error”) and fewer noisy terms than the other parts of the report. Conversely, the terms from the S2Rs hinder retrieval. We found that the shared S2R terms with the buggy file (*i.e.*, “create”, “contain”, and “source”) appear in more than 3,600 files, while the TITLE and OB terms appear in no more than 326 files. Also, the term “project” appears in more than 1,600 documents. In this case, the S2R terms are not discriminatory for TRBL (*i.e.*, they are noisy). Using OB+TITLE, S2R+TITLE, OB+S2R, and OB+S2R+TITLE as reformulation strategies lead to retrieving the buggy file in positions 4, 314, 633, and 60, respectively. These results confirm the usefulness of OB and TITLE for TRBL (*i.e.*, the buggy code is retrieved on position 4). Note that the S2Rs, when combined with OB or TITLE, deteriorates the rank of the buggy file.

Consider the example in Figure 6.4, regarding bug report #4330 from ArgoUML and its respective buggy class `TabToDo`. In this case, the OB states that there is an “exception”

[10]The query is *low-quality* for the other three file-level TRBL techniques as well.

[11]The position of the buggy file, after excluding the first top-5 documents would be 174.

Bug report title:
Exception in ”Send $\boxed{\text{email}}$ to $\boxed{\text{expert}}$” [OB,TITLE]

Bug report description:
Steps to reproduce:
- select an active critic from the $\boxed{\text{ToDoPane}}$ [S2Rs]
- press the ”send $\boxed{\text{email}}$ to $\boxed{\text{expert}}$” button [S2Rs]

The following exception is thrown (stack from the console, no pop-ups appear): [OB]

Exception in thread ”AWT-EventQueue-0” java.lang.NullPointerException
at org.$\boxed{\text{argouml}}$.ui.cmd.$\boxed{\text{ActionEmailExpert}}$.$\boxed{\text{action}}$Performed($\boxed{\text{ActionEmailExpert}}$.java:57)

[...]

Buggy code class: `TabToDo`

```
[...]
public class TabToDo extends AbstractArgoJPanel implements TabToDoTarget {
    [...]
    private static UndoableAction actionEmailExpert = new ActionEmailExpert();
    [...]
    /**
    * The constructor.
    * Is only called thanks to its listing in the org/argouml/argo.ini file.
    */
    public TabToDo() {
        [...]
        JToolBar toolBar = new ToolBar(SwingConstants.VERTICAL);
        toolBar.add(actionNewToDoItem);
        toolBar.add(actionResolve);
        toolBar.add(actionEmailExpert);
        [...]
        splitPane = new BorderSplitPane();
        add(splitPane, BorderLayout.CENTER);
        setTarget(null);
    }
    public void setTree(ToDoPane tdp) {
        if (getOrientation().equals(Horizontal.getInstance())) {
            splitPane.add(tdp, BorderSplitPane.WEST);
        } else {
            splitPane.add(tdp, BorderSplitPane.NORTH);
        }
    }
    [...]
}
```

Bug report found at `http://argouml.tigris.org/issues/show_bug.cgi?id=4330`

The boxed terms represent the terms shared between the report and the class. Each sentence in the report is marked according to its corresponding information type: [TITLE], [OB], or [S2Rs].

Figure 6.4: Bug report #4330 from ArgoUML and its corresponding buggy class.

that produces the reported "stack (trace)". The TITLE describes part of the OB (*i.e.*, the exception) and the feature being used (*i.e.*, "send email to expert"), which is also described in the S2Rs. The S2Rs provides additional information/context related to the feature (*i.e.*, selecting an "active critic" in "ToDoPane"). When using the full bug report as initial query to Lucene, the buggy class is retrieved in the 334th position. When the query is reformulated by using the OB and TITLE, the class is retrieved in the 5th position (after removing the first top-5 irrelevant documents). The significant improvement is because many terms from other parts of the bug report (especially from the stack trace) appear in other documents of the corpus. Also, the terms "email" and "expert", present in the TITLE, appear frequently in the buggy class, hence, they are highly relevant. When the S2Rs are also retained in the reformulated query, the buggy class ranks in the 3rd position (after removing the first top-5 irrelevant documents). In this case, three terms are added to the query, namely "ToDoPane", "email" and "expert". The first term is a new term in the query and is highly relevant. While the other two are already present in the query, their frequency/weight gets increased, thus improving the ranking. This case illustrates how the S2Rs contain specific terms about the problem and are useful for TRBL.

Although we consider combining the OB and TITLE (as well as the S2Rs, when present in the bug report) as the most effective and practical strategy to reformulate queries, note that their combination with EB is highly effective as well. For all TRBL techniques, except BugLocator, the EB, when combined with the OB, TITLE or S2Rs, achieves the best effectiveness, and in many cases, it achieves comparable consistency across different thresholds. The only shortcoming of combining the EB with other information is that it is not frequently found in bug reports. In any case, we recommend developers to use the expected behavior (when available) together with the OB, TITLE, and/or S2Rs. According to the results, by using this strategy, developers can expect to find the buggy code artifacts in the top of the list for more cases than with no reformulation (*e.g.*, 47.7% on average, when combined with

OB and TITLE, across different techniques, granularities, and thresholds). We observed that when using the EB, the terms retained from the OB (as well as from other information types) get weighted, thus leading to higher TRBL performance. This indicates how similar the OB and EB are.

It is important to note that the S2Rs and CODE alone (*i.e.*, when they are not combined with any of the other information types) consistently deteriorate HITS@N with respect to no reformulation. In many cases (*e.g.*, the Eclipse case in Figure 6.3), the S2Rs include terms that refer to several features of the system (*i.e.*, it gives a broad problem context), which can diverge the retrieval engine from the specific buggy code. In other cases, the S2Rs can refer to higher layers of the systems' architecture (*e.g.*, the GUI layer) instead of referring to lower layers, which are the buggy ones in many cases. In the future, we will combine the reformulation strategies with code dependency analysis to trace the buggy code across layers. While in many cases, CODE snippets use few code artifacts, we observed that in many others, they refer to many classes or methods. Usually, these latter cases correspond to large code snippets, which help communicate the bug better to the developers. However, since they contain many code references, their discriminatory power is low (for text retrieval), which leads to retrieving many non-buggy documents. In addition, as seen in Figure 6.3, the CODE may not share any terms at all with the buggy documents.

We observed that in many cases, the relevant terms from the OB, S2Rs, and TITTLE are present in other parts of the bug report. We believe that the weight of these terms can be boosted according to how frequently they appear in the full bug report. In our future work, we will investigate this method for improving the performance of the reformulation strategies.

6.4.9 Successful vs. Unsuccessful Queries

During query reformulation, there is always a trade-off, as some queries become *successful* while others become *unsuccessful* with respect to no reformulation. A good reformulation strategy would lead to more successful queries (*i.e.*, retrieving buggy code artifacts in top-N) than unsuccessful queries (*i.e.*, not retrieving buggy code artifacts in top-N), compared to the initial queries. We aim to better understand the trade-offs for the best reformulation strategy: OB+TITLE. We also analyze the case when OB+TITLE is combined with the S2Rs: OB+S2R+TITLE.

We refer to all the queries that retrieve code artifacts in top-N as *successful queries*, and to those that do not retrieve code artifacts as *unsuccessful queries*. Ideally, a reformulation strategy would preserve the successful queries (*i.e.*, an initial *successful* query, which reformulated remains *successful*, *a.k.a. successful* → *successful*), while converting all (or at least some) of the initially-unsuccessful queries into successful ones (*i.e.*, *unsuccessful* → *successful*). In other terms, we want to avoid a situation when *successful queries* turn *unsuccessful* (*i.e.*, *successful* → *unsuccessful*) via the reformulation.

Table 6.26 shows that OB+TITLE and OB+S2R+TITLE transform about the same proportion of unsuccessful queries into successful ones (*i.e.*, approx. 11% of the queries, on average). However, OB+TITLE converts slightly more successful queries into unsuccessful ones compared to OB+S2R+TITLE (*i.e.*, 6% vs 4.2% on average, respectively), which is less desirable. Both strategies preserve nearly the same proportion (*i.e.*, approx. 17%) of the successful queries, and OB+S2R+TITLE preserves slightly more unsuccessful queries than OB+TITLE (*i.e.*, 67.8% vs 66.1%, respectively), which is less desirable. The small differences of successful and unsuccessful queries before and after reformulation supports our conclusion in that both strategies achieve comparable TRBL performance. Finally, approach-wise, note that for the case of OB+S2R+TITLE, Lucene and Lobster are the approaches with the highest proportions of *unsuccessful* → *successful*, which are substantially higher than the proportions for BugLocator and BRTracer. For the case of OB+TITLE, Lucene, Lobster, and Locus achieve the highest proportions of *unsuccessful* → *successful* queries, which are higher than the proportions for BugLocator and BRTracer. These results further show the robustness of BugLocator and BRTracer with respect to noisy queries.

Bug report title:
too large first $\boxed{\text{step}}$ with embedded $\boxed{\text{Runge}}$-$\boxed{\text{Kutta}}$ $\boxed{\text{integrators}}$ (Dormand-Prince 8(5,3) ...) [OB,TITLE]

Bug report description:
Adaptive $\boxed{\text{step}}$ $\boxed{\text{size}}$ $\boxed{\text{integrators}}$ $\boxed{\text{compute}}$ the first $\boxed{\text{step}}$ $\boxed{\text{size}}$ by themselves if it is not provided. For embedded $\boxed{\text{Runge}}$-$\boxed{\text{Kutta}}$ type, this $\boxed{\text{step}}$ $\boxed{\text{size}}$ is not $\boxed{\text{checked}}$ against the $\boxed{\text{integration}}$ range, so if the $\boxed{\text{integration}}$ range is extremely short, this $\boxed{\text{step}}$ $\boxed{\text{size}}$ may $\boxed{\text{evaluate}}$ the function out of the range (and in fact it tries afterward to go back, and fails to stop). [OB]
Gragg-Bulirsch-Stoer $\boxed{\text{integrators}}$ do not have this problem, the $\boxed{\text{step}}$ $\boxed{\text{size}}$ is $\boxed{\text{checked}}$ and truncated if needed.

Buggy code method: `EmbeddedRungeKuttaIntegrator:integrate`

```
/** {@inheritDoc} */
@Override
public void integrate(final ExpandableStatefulODE equations,
                      final double t)
                      throws MathIllegalStateException,
                      [...] {
   sanityChecks(equations, t);
   [...]
   // set up an interpolator sharing the integrator arrays
   final RungeKuttaStepInterpolator interpolator =
         (RungeKuttaStepInterpolator) prototype.copy();
   [...]
    stepSize = hNew;
    // next stages
    for (int k = 1; k < stages; ++k) {
        for (int j = 0; j < y0.length; ++j) {
            [...]
            yTmp[j] = y[j] + stepSize * sum;
        }
        computeDerivatives(stepStart + c[k-1]
                           * stepSize, yTmp, yDotK[k]);
    }
   [...]
   if (fsal) {
       // save the last evaluation for the next step
       System.arraycopy(yDotTmp, 0, yDotK[0], 0, y0.length);
   }
   [...]
}
```

Bug report found at `https://issues.apache.org/jira/browse/MATH-727`

The boxed terms represent the terms shared between the report and the method. Each sentence in the report is marked according to its corresponding information type: [TITLE] or [OB].

Figure 6.5: Bug report #727 from Math and its corresponding buggy method.

Table 6.26: Proportion of Successful and Unsuccessful Queries.

(a) OB+TITLE

TRBL technique	**|Q|**	**U → S**	**S → U**	**S → S**	**U → U**
Lucene	763.6	13.9%	7.3%	16.0%	62.7%
Lobster	34.6	17.3%	2.0%	17.0%	63.6%
BugLocator	552.4	5.3%	6.2%	10.5%	78.0%
BRTracer	458.3	7.7%	7.8%	14.6%	69.9%
Locus	190.7	11.7%	6.7%	25.4%	56.2%
Average		**11.2%**	**6.0%**	**16.7%**	**66.1%**

(b) OB+S2R+TITLE

TRBL technique	**|Q|**	**U → S**	**S → U**	**S → S**	**U → U**
Lucene	409.0	12.7%	6.6%	16.2%	64.4%
Lobster	19.9	20.0%	0.4%	19.3%	60.4%
BugLocator	309.1	4.1%	4.4%	11.0%	80.4%
BRTracer	255.5	7.6%	4.7%	15.8%	71.8%
Locus	124.4	7.6%	4.6%	25.8%	62.0%
Average		**10.4%**	**4.2%**	**17.6%**	**67.8%**

Average proportion of Successful (**S**) and Unsuccessful (**U**) queries before reformulation that turned Unsuccessful (**U**) and Successful (**S**) after reformulation, when using each reformulation strategy. Average values across the 3 data sets and 26 thresholds N.

Figure 6.5 illustrates a *successful* → *successful* case when using Lucene. The full bug report #727 (from Math), used as input query to Lucene, fails to return the buggy method within the top-5 results (*i.e.*, N=5), *i.e.*, the method is ranked in the 7th position. Reformulating the query with the TITLE and the OB from the bug report[12] leads to retrieving the buggy method in the 13th position, *i.e.*, 6 positions down the result list. Figure 6.5 reveals that all the sentences in the bug report contain shared terms with the buggy method, and the terms "step", "size", "integrat(or/tion)", "Runge", and "Kuttap" are the most relevant ones, since they appear frequently in the buggy method. When the query is reformulated, the only relevant term that is completely removed is "compute". While the most relevant

[12]The reformulation results in the query: "too large first step ... (Dormand-Prince 8(5,3) ...) For embedded Runge-Kutta type, this step size ... and fails to stop)."

terms are not removed by the reformulation, they appear less frequently in the reformulated query (*i.e.*, their term frequency decreases), thus reducing their weight and finally hindering the retrieval of the buggy method. This is another example of how increasing the weight of the terms appearing in the OB or TITLE, based how frequent they appear in other parts of the bug report, may improve TRBL.

Another *successful* → *successful* example is the bug report #1152 from Tika [47], whose buggy class is `ChmLzxBlock`. When the full bug report is used as input query to Lucene, the buggy class is retrieved in the 6th position. When the query is reformulated, by using the OB (*i.e.*, "... Java process stuck"), the TITLE (*i.e.*, "Process loops infinitely... "), and the S2Rs (*i.e.*, "By parsing the attachment CHM file..."), the class is retrieved in the 11th position. The reason for the deterioration is the removal of the terms corresponding to the stack trace included in the report, which contains the terms of the buggy class name. The reformulation strategy removes this content since it is not natural language written by the users, hence does not correspond to OB. Note that the initial query is not *low-quality* when using Lobster. This is because Lobster uses the stack traces within bug reports to boost the classes that appear in the traces as well as their dependencies (*i.e.*, related classes). As part of our future work, we will investigate ways to incorporate (parts of the) stack traces into the reformulation.

6.5 Threats to Validity

We discuss the threats that could affect the validity of our empirical evaluation.

The main threat to *construct validity* concerns the criteria used to determine if a query is successful or unsuccessful within the proposed scenario for bug localization (see Section 6.2). In our experimental setting, the buggy code artifacts are known for each query/bug report. We determined the success of a query by measuring the rank of these artifacts in the list produced by the TRBL techniques when using the query as input to them. A query is deemed successful if any of the buggy code artifacts is found within the top-N results (i.e., their rank is less than or equal to N), otherwise the query is considered unsuccessful. In a real case scenario, the developer does not know the buggy code artifacts beforehand, and determining the success of a query implies manually inspecting the returned code candidates, which may be non-trivial. The metrics used in the evaluation, in particular HITS@N, which is based on the rank of the first buggy document found in the result list, were used as a proxy to measure the effort spend by a developer when inspecting the code candidates. Although, this is a widely-used experimental setting in TRBL research, it might not resemble a realistic scenario for bug localization. In our future work, we will address this threat to validity by conducting empirical studies with developers to determine the usefulness of reformulating the initial queries via the proposed reformulation strategies.

Another threat to *construct validity* is the subjectivity introduced in the labeled set of bug reports when manually identifying the OB, EB, and S2Rs, as each bug report was coded by a single coder. We made this choice in order to maximize the number of queries used in our evaluation. Also, our past experience when we had multiple coders per bug report revealed high agreement between coders [88, 83]. In order to reduce subjectivity, we used the set of common coding criteria that we defined in our prior work [88, 83]. We also conducted training sessions with the coders, which included examples and discussion of ambiguous phrases in the bug reports. The impact of bug coding from different coders on code retrieval will be investigated in our future work.

In order to mitigate threats to the *conclusion validity*, we compared the performance of the initial and reduced queries using HITS@N, MRR, and MAP, metrics widely used in TRBL research [220, 245, 167, 225]. We focused our evaluation primarily on HITS@N. We argue that this metric is best for assessing query reformulation for TRBL as, in practice, developers would likely inspect the top N candidate code artifacts only, before switching to another bug localization method (*e.g.*, navigating code dependencies). Also, HITS@N is more intuitive and easier to interpret than MRR and MAP. We categorized the strategies using three dimensions, namely *effectiveness*, *applicability*, and *consistency*, which allowed us to determine the best strategies across TRBL techniques and granularities. We also analyzed the trade-offs of our reformulation strategies, to further strengthen our conclusions. We defined two categories of queries (*i.e.*, *successful* and *unsuccessful*) and analyzed the transition of the queries between categories before and after reformulation. Similar analyses have been used in prior query reformulation research [83, 116].

The *internal validity* of our evaluation is affected by our TRBL data sets and approaches. Based on data previously used in TRBL studies [245, 225, 167, 161, 83, 142], we built three data sets at different granularity levels (*i.e.*, method-, class-, and file-level). These data sets contain bug reports/queries and code corpora that correspond to distinct Java software systems. While we observed variation in results across data sets (*i.e.*, code granularity) and TRBL approaches, the common denominator in all treatments was our query reformulation strategies, which we consider the main factor in the observed improvements. We also used five state-of-the-art TRBL techniques proposed by prior research. As mentioned before, the differences in performance we observed for the original implementation of Buglocator and our implementation may impact the results, but we consider the impact minimal, and using the other four approaches confirms that the successful reformulations work with different approaches. Finally, our query sample contains a small subset of duplicated queries across the three code granularities and projects versions. The duplication stems from the independent data collection process performed by the data owners of the original data sources. These queries can be treated as different queries because they are likely to perform differently across granularities and project versions. In any case, given the small proportion of these queries in our sample (*i.e.*, 4.3% total), we consider that their impact in the results is minimal.

We addressed the *external validity* of our empirical evaluation by using 1,221 *low-quality* queries from 248 versions of 30 different software systems that span different domains and software types. We used nearly as three times more queries and nine more software projects than in our prior work on OB-based query reformulation [83] to strengthen the generalizability of our conclusions. Finally, we used five TRBL techniques, namely Lucene [117], Lobster [167], BugLocator [245], BRTracer [225], and Locus [224]. Investigating the effectiveness of our reformulation strategies with other TRBL techniques is part of our future work.

6.6 Conclusions

We proposed a set of reformulation strategies based on the structure of bug descriptions. These strategies can be employed when using the full bug reports as initial queries fails to retrieve the buggy code artifacts within the top retrieved results (*i.e.*, these bug descriptions result in *low-quality* queries). Our hypothesis was that the TITLE of the bug reports, the observed behavior (OB), expected behavior (EB), and steps to reproduce (S2Rs), as well as the code snippets (CODE) in the bug description contain relevant information with respect to TRBL, while other parts of the description include irrelevant terms that act as noise for code retrieval. From the combination of these five types of content, we defined 31 query reformulation strategies that are based on the user selecting the TITLE, OB, EB, S2R, or CODE parts of the bug description. We used the defined strategies to reformulate 1,221 *low-quality* queries, which were executed using five state-of-the-art TRBL approaches on data of three code granularity levels (*i.e.*, file, class, and method). We assessed the ability of the reformulation strategies to retrieve the buggy code artifact(s) within the top-N returned candidates for 26 different thresholds (N={5, 6, 7, ..., 30}) in comparison with no reformulation, when excluding the first N irrelevant results produced by the initial queries.

The results indicate that combining the TITLE and the OB from the bug descriptions is the best reformulation strategy across the five TRBL approaches and three code granularities, as it leads to retrieving the buggy code artifacts within the top-N results for 25.6% more queries (on average) than without query reformulation. This strategy is *highly-applicable* and *highly-consistent* across different thresholds N. In addition, combining the OB and TITLE with the S2Rs, when provided in the bug reports, leads to better retrieval performance (*i.e.*, for 31.4% more queries with respect to no reformulation) and comparable consistency, yet it is applicable in fewer cases. Likewise, using the EB (when available) along with the OB and TITLE leads to better performance (*i.e.*, 41.7% more queries with respect to no reformulation) and comparable consistency. However, the shortcoming of using this strategy is its low applicability, given that the EB is not frequently found in bug reports.

We also found that three of the TRBL approaches we experimented with (*i.e.*, BugLocator, BRTracer, and Locus) are less sensitive to noisy queries than the other two (*i.e.*, Lucene and Lobster), while all benefit from the best query reformulation strategies we defined. The results bear evidence in support of our hypothesis about the effectiveness of the structure of bug descriptions on TRBL. Our reformulation strategies are simple to use, do not depend on any information outside the bug report, and demand minimal effort from the developer, *i.e.*, simply select the TITLE and the sentences describing the OB and the S2Rs (when available).

As future work, we plan to evaluate the proposed reformulation strategies with additional TRBL approaches. In addition, we will conduct a sensitivity analysis of the reformulation strategies with respect to fuzzy selection of the OB, EB, and S2Rs by different users. While we believe that the proposed reformulation strategies are easy to use, as they only require a copy-paste operation from the user, we need empirical evidence to support this. Our future research will be directed towards finding such evidence. In addition, we will focus on investigating combined reformulation strategies, that is, not only query reduction, which may lead to even better results. Finally, expanding the evaluation on more TRBL data and queries is also planned.

6.7 Acknowledgments

This chapter was published at the Empirical Software Engineering journal [85]. This publication is an extended version of the paper published and presented [83] at the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17). These manuscripts were published in collaboration with Juan Manuel Florez from The University of Texas at Dallas. Oscar Chaparro and Juan Manuel Florez contributed to the design, implementation, and evaluation of the proposed approach.

CHAPTER 7
QUERY REFORMULATION FOR DUPLICATE BUG REPORT DETECTION[1]

7.1 Introduction

Duplicate bug report detection techniques have two aspects in common: (1) most of them rely on some form of text retrieval (TR) or text-based classifiers, where a query is formulated from the full textual description in the new bug report (*i.e.*, title + description) and old bug reports are retrieved (or classified) as duplicate candidates. Some techniques use simple TR models, while others use more complex ones. The common trend in improving such approaches is adding additional information, such as stack traces and other information recorded in the reports (see Section 2.4). Other approaches also consider information from other sources and/or employ techniques to learn from past duplicate sets. The common premise here is that there are language commonalities between bug reports corresponding to the same bugs [82]. However, even the most sophisticated duplicate detection approaches are far from perfect. Among the ones we reviewed in Section 2.4, the highest reported recall rates@10 are around 80%-90% [168, 57, 144, 209, 223]. This means that in 10%-20% of the cases, these techniques fail to return existing duplicates in the top-10 of the retrieved list of bug reports; (2) all these approaches assume a rather limited usage scenario. The tools use the new bug report as a query and then the user inspects the ranked list of retrieved bug reports to check if any are duplicates of the new bug report. At some point, if a duplicate is not found, the user chooses to stop and mark the bug as *new* or tries some other approach.

We contend that duplicate bug report detection is different from other text retrieval applications in software engineering, and tool support should address the differences. For

[1]© 2019 IEEE. Reprinted with permission from Oscar Chaparro, Juan Manuel Florez, Unnati Singh, and Andrian Marcus, 'Reformulating Queries for Duplicate Bug Report Detection", Proceedings of the 26th International Conference on Software Analysis, Evolution, and Reengineering (SANER'19).

example, in applications for other tasks, such as feature/bug localization [157, 105], traceability link recovery [100], or impact analysis [176, 145], the users have a specific information need, such as finding a relevant part of the code. They would formulate a query (sometimes with tool support) and retrieve relevant documents from the code corpus. Then, the users assess the relevance of the retrieved code documents and may reformulate the query to retrieve others. They may also use some of the retrieved documents as a starting point for navigating the code. The users eventually *stop only when their information need is met* (*i.e.*, they found the code artifact of interest), and they can continue solving their main task.

In contrast, during duplicate bug report detection (*a.k.a. duplicate detection* or *DD*), the user stops when she finds a duplicate report or *when she is confident enough that there are no duplicates*. In other words, users may stop without finding any relevant result, as there is no specific information need that must be met, in form of a retrieved artifact, which is then used to solve the task at hand (*e.g.*, fixing the bug). Not retrieving any duplicates is a likely outcome. Our main goal is to increase the user confidence when she makes the decision to stop. We propose that tool support should return a number, say N, of bug reports, rather than rank the entire search space. More importantly, when a duplicate is not found among the N retrieved candidates, the user should be able to refine the query and retrieve N more, before making a final decision. The expectation is that the reformulated query is better at retrieving the duplicates (if any) in top N than the original query is at retrieving them in top 2N, which would yield higher confidence in the retrieval.

Furthermore, in the other text retrieval applications, when the relevant documents are not found, users rely on their specific information need, the retrieved results, and the software knowledge, for reformulating queries. Differently, during bug report duplicate detection, when no duplicates are returned, the lack of a specific information need and the knowledge about the code or the retrieved results are unlikely to help with query reformulation. The user is also unlikely to have much specific information about the (potentially) thousands of previous reports, which she could leverage for reformulation. Hence, the major challenge is to have a query reformulation strategy that is independent of such factors and would help retrieve duplicates in more cases.

7.2 Query Reduction Approach

We rethink tool-supported duplicate bug report detection as a two-step process, using the entire new bug report as a query in the first step (*i.e.*, the initial query), for retrieving N bug reports, and a reformulated query in the second step, for retrieving additional N bug reports, if needed. We propose and evaluate three reformulation strategies for improving duplicate bug report detection. The reformulation strategies are independent of the underlying detection approach and they do not depend on the returned results or any information from other external sources. In other words, they are easy to use by any potential user and should work with any existing duplicate bug report detection tool based on bug reports.

Our approach is based on the observation that most bug reports have an inherent structure, consisting of the bug title (BT), the observed behavior (OB), the expected behavior (EB), and the steps to reproduce the noted bug (S2Rs) [249, 98, 88, 83]. While many bug reports may lack EB and S2Rs, our previous research found that OB is present in ~93% of the bug reports [88, 98]. Our previous work also found that OB contains salient information that helps locate the corresponding bugs in the code, more than other parts of the bug report [83]. We conjecture that *the information contained in the OB may result in better retrieval of duplicate bug reports.* Intuitively, we expect that OB encodes information unique to the reported bug, while other parts of the bug reports would have elements potentially common to many other bugs. With that in mind, our reformulation strategy is a query reduction approach, where the new query is obtained by retaining only the OB from a bug report (*i.e.*, from the title and description) and removing the rest of the textual bug description. We call this reformulation strategy *OB-DD*.

The bug title also exhibits some of the same properties that OB does. BT is present in all bug reports, hence it can be easily leveraged for query reformulation. Previous research noted that bug reporters create the bug title as a summary of the entire report, meaning that they often select the words that best describe the bug [150, 136]. In addition, nearly all existing duplicate detection techniques rely specifically on the bug title as a feature for retrieval or classification (see Section 7.4). Hence, we consider an additional query reduction strategy, where the initial query (*i.e.*, the entire bug report) is reduced to its BT only. We call this reformulation strategy *BT-DD*.

While the bug title may contain words that describe the OB, we conjecture that OB and BT each have unique terms that are important for retrieval. Therefore, we also consider the hybrid reformulation strategy, which reduces the initial query to the OB *and* BT only. We call this combined strategy *OBT-DD*.

The three reformulation strategies can all be employed with any duplicate bug detection tool that relies on the new bug report (and potentially other information) as query, and the bug report contains a part describing the software's observed behavior. If no duplicates are retrieved within the top-N results, when using the entire bug report as an initial query, then the user should select the OB and/or BT and remove the rest of the textual bug description. Elements such as stack traces or code snippets may be retained if the tool uses them. Then, the user can run the new query and retrieve N more bug reports.

7.3 Empirical Evaluation

We performed an empirical evaluation of the proposed reformulation strategies. The evaluation aims at answering the following research questions:

***RQ*$_1$**: *Do the three query reformulation strategies help duplicate detection approaches retrieve more duplicate bug reports than without query reformulation?*

RQ_2: *Which of the three query reformulation strategies retrieve more duplicate bug reports?*

We used a Lucene-based approach (Section 7.3.1) to detect duplicates for a large set of bug reports (Sections 7.3.2 and 7.3.3). Then, for a subset of the bug reports for which the tool failed to retrieve duplicates in top N (Section 7.3.4), we used the three strategies to reformulate the reports (Sections 7.3.5 and 7.3.6) and assessed how many more duplicates are retrieved among the next N candidates (Section 7.3.6). Section 7.3.7 presents and discusses the evaluation results.

7.3.1 Duplicate Bug Report Detector

We selected a DD approach based on Lucene [117], which was used in previous research for retrieving duplicate bug reports [72, 82] and also for retrieving duplicate Stack Overflow posts [82]. Since the OB-based query reformulation strategies do not depend on the underlying duplicate detection approach, we expect the reformulations to be effective for other, more complex duplicate detection approaches as well. However, such an investigation is subject of future research.

Lucene [117] is a retrieval technique implemented in the open source library of the same name [14], which combines the standard information retrieval Boolean model and the Vector Space Model (based on TF-IDF [197]) to compute the similarity between a new bug report (*i.e.*, the query) and an existing report. Lucene is a technique that relies only on textual information to retrieve a ranked list of candidate duplicate reports. Typically, a Lucene query is created by concatenating the bug report's title and description, including any information embedded in these sources (*e.g.*, code snippets). In our evaluation, we used Apache Lucene v5.3.0 [14] with the default similarity measure and parameters (see [20] and [21] for details).

7.3.2 Bug Report Data Set

We use data that corresponds to 449,901 bug reports (*a.k.a.* issues) from 20 open source projects (see Table 7.1). This data was constructed based on two data sets used in prior duplicate detection research: the one used by Sun *et al.* [209] (*a.k.a. SDS*) and the one provided by Chaparro *et al.* [82] (*a.k.a. CDS*).

We used the bug reports for three projects from SDS (*i.e.*, Eclipse, Mozilla Firefox, and OpenOffice). The original data in SDS is in the format used by Sun *et al.*'s tool (*i.e.*, REP), which is essentially a set of real-valued vectors corresponding to *n*-grams extracted from the bug reports. Since the reformulation strategies require the bug description, we downloaded the full text and extra meta-information (*e.g.*, bug type, version, *etc.*) of the bug reports used in SDS (from the issue trackers), rather than using the original data set made available by Sun *et al.*

As for CDS, we selected 17 (of 115) projects with the largest number of bug reports. From the projects' issue trackers, we downloaded the same bug reports information as for SDS. Our data set includes all bug reports submitted to the issue trackers of each project, at the time of creation of CDS (*i.e.*, May 2016) and SDS (*i.e.*, Dec. 2007), except for Firefox and OpenOffice. For these projects, our data set contains only the subset of bug reports provided by Sun *et al.* [209]. Firefox' bug reports are from 2010 and OpenOffice's reports are from 2008 to 2010 only. The CDS projects use Jira [19] issue tracker, while the SDS projects use Bugzilla [16]. Our data set spans different software types that range from desktop applications (*e.g.*, Eclipse) to frameworks/libraries (*e.g.*, Struts and PDFBox), in a variety of domains, such as web browsing (Firefox), office productivity (OpenOffice), mobile/web development (*e.g.*, Cordova and MyFaces), distributed computing (*e.g.*, Hadoop and Spark), databases (*e.g.*, Cassandra or Hive), development tools (*e.g.*, Groovy and Maven), *etc.*

Table 7.1: Statistics of the duplicate detection data set.

Project	**# of queries**	**Total # of bug reports**	**# of duplicate buckets**
Accumulo	79	4,106	72
Ambari	117	14,763	100
ActiveMQ	120	5,936	104
Cassandra	334	11,011	273
Cordova	247	10,208	188
Continuum	89	2,722	68
Drill	186	4,305	155
Eclipse	28,518	209,056	16,833
Groovy	115	7,486	97
Hadoop	231	10,645	194
Hbase	97	15,114	93
Hive	315	12,854	268
Maven	275	4,758	177
M. Firefox	7,585	75,653	4,510
MyFaces	79	3,657	55
OpenOffice	3,019	31,138	1,708
PDFBox	155	3,207	106
Spark	430	12,676	359
Wicket	149	6,077	127
Struts	86	4,529	75
Total	**42,226**	**449,901**	**25,562**

7.3.3 Queries and Ground Truth

We used the duplicate references between bug reports in the issue tracker to determine the set of queries and their respective duplicate reports. To do so, we first identified buckets of duplicate reports in the data, using direct and transitive duplicate references in the issue tracker (similar to Sadat *et al.*'s approach [194]). For example, if A is a duplicate of B, which is a duplicate of C, with C being the oldest (*i.e.*, submitted first in the issue tracker) and A the youngest (*i.e.*, submitted last), then the respective bucket is {A, B, C}. If A is a duplicate of C, and B is a duplicate of C, then we form the same bucket {A, B, C}.

We sorted the bug reports in each bucket chronologically by creation date (in our example: {C, B, A}), and for each bucket of size n, we created $n-1$ queries, each having as ground truth the ones that are older from the bucket. The oldest bug report in each bucket is called the *master* bug report [209] and it is the only one that does not reference any past duplicate, hence it is not considered as a query. In the above example, C is the *master*, while A and B are used as queries, with {B, C} and {C} as ground truth, respectively. We created the ground truth for each query, as described in the above example, which includes the list of corresponding past duplicate reports.

Overall, our data sets contain 25,562 duplicate buckets, from which 42,226 queries were created by using the title and description of the bug reports (see Table 7.1). All bug reports in the corpus and all queries were normalized by using standard preprocessing operations. First, we identified the words in the bug report text, and performed code identifier splitting based on the camel case and underscore formats (*e.g.*, *CodeIdentifier* or *code_identifier* would split into *code* and *identifier*). Then, we removed words that are unlikely to contribute to retrieval, namely, special characters (*e.g.*, *#* or *$*), numbers, common English stop words (*e.g.*, *about*, *because*, *the*, *etc.*), Java keywords (*e.g.*, *for*, *while*, *at*, *with*, *like*, *etc.*), and words shorter than three characters. Finally, we applied Porter's algorithm to reduce the words to their root form [175].

It is important to note the differences between the number of queries for the SDS projects presented by Sun *et al.* [209] and the ones presented in Table 7.1. Before discussing such differences, we must point out that each query in the SDS data set uses the *master* bug report as the only duplicate in the ground truth. We claim that the ground truth based on all past duplicates for a query is better suited for evaluating DD approaches, because, when used in practice, users expect to retrieve any of the duplicate bug reports and not necessarily the master bug report only. We use this approach in our evaluation.

We compared the list of queries in SDS and our data set and found 2,741 differences in the following three categories:

1. Queries that are not present in SDS but are present in our data set. We found 1,824 queries that reference duplicate reports transitively or directly, which are missing in the SDS data set.

2. Queries that are not present in our data set but are present in SDS. We found 521 queries that do not reference any duplicate reports at all (by following the query generation approach described above). However, the SDS data set (incorrectly) includes these queries.

3. Queries that are present in both our data set and SDS, but the ground truth (*i.e.*, the *master* report) is different. We found 396 queries for which the issue tracker reports a different master bug report than the one in SDS.

We manually inspected a subset of the reports in each category and confirmed the differences. The replication package contains the full list of queries with differences [87].

In summary, our data set includes: (1) a set of queries generated from the bug reports submitted on the projects' issue tracker; (2) the past duplicate bug reports for each query, which represent the ground truth; and (3) the entire set of existing bug reports which represents the document search space for DD. The full set of stop words, bug reports, queries, and ground truth, is available in our replication package [87].

7.3.4 Low-quality Queries

Our reformulation approach follows the scenario in which the triager issues the *initial query* (using the full text of a bug report) and inspects the top-N candidates returned by the DD technique at hand (*e.g.*, Lucene). If none of the candidates are deemed duplicated (by inspecting their title and/or description), the triager reformulates the query (via the reformulation strategies) and inspects additional N candidates. In this scenario, the user would inspect a total of 2N candidates. Large N values (say 30 and beyond) would mean that our approach is impractical because, in the worst-case scenario, it would imply inspecting 60 results total, which could demand a significant effort from the user. It is likely that the triager would not assess more than 30 reports. Very small N values (say less than 5) would imply an unrealistic scenario. If the triager finds at least one duplicate report within the top-5 results, then she does not need reformulation. We contend that inspecting 5 to 10 documents (i.e., 10 to 20 documents total, following reformulation) is a realistic scenario for DD. In other words, if a query retrieves the buggy code in top-5/10, then it is likely that no reformulation is needed. Similar thresholds have been used in prior DD research [209, 193, 120, 183]. Since there is no specific research on user behavior during query reformulation for duplicate bug report detection, we do not want to limit the evaluation only to the thresholds we consider most realistic. Hence, we include results for the threshold set N={5, 6, 7, ..., 30}, which amounts to 26 thresholds total.

Our reformulation strategies focus on queries that fail to retrieve the duplicate reports within the top-N results (*i.e.*, *low-quality* queries). Therefore, in order to determine the set of *low-quality* queries, we executed Lucene with the initial queries (generated from the entire text of the bug report's title and description) and checked if none of the duplicates (among the set of previously-reported reports for each query [183]) were retrieved in the top-N results, for N={5, 6, 7, ..., 30}.

Lucene fails at retrieving duplicates for 50.6%, 43.9%, 40.2%, 37.5%, 35.5%, and 33.9% of the queries, within the top-5, -10, -15, -20, -25, and -30 results, respectively (see Table 7.2). These numbers mean that a large number of queries require reformulation (between 14,3k and 21,3k queries, for N=30 and 5, respectively). More sophisticated duplicate detectors may achieve better retrieval [168, 57, 144, 209, 223], but there is still a large percentage of queries that fail to retrieve duplicates.

Table 7.2: Number of *low-quality* queries for N={5, 10, 15, 20}.

Project	**Top-5**	**Top-10**	**Top-15**	**Top-20**
Accumulo	31 (39.2%)	26 (32.9%)	22 (27.8%)	21 (26.6%)
Ambari	25 (21.4%)	21 (17.9%)	21 (17.9%)	19 (16.2%)
ActiveMQ	54 (45.0%)	44 (36.7%)	42 (35.0%)	37 (30.8%)
Cassandra	194 (58.1%)	176 (52.7%)	164 (49.1%)	154 (46.1%)
Cordova	106 (42.9%)	90 (36.4%)	82 (33.2%)	76 (30.8%)
Continuum	31 (34.8%)	28 (31.5%)	27 (30.3%)	24 (27.0%)
Drill	102 (54.8%)	90 (48.4%)	81 (43.5%)	72 (38.7%)
Eclipse	14,7k (51.5%)	12,9k (45.2%)	11,8k (41.5%)	11,1k (38.9%)
Groovy	48 (41.7%)	46 (40.0%)	41 (35.7%)	38 (33.0%)
Hadoop	79 (34.2%)	68 (29.4%)	63 (27.3%)	58 (25.1%)
Hbase	38 (39.2%)	33 (34.0%)	27 (27.8%)	26 (26.8%)
Hive	160 (50.8%)	131 (41.6%)	123 (39.0%)	113 (35.9%)
Maven	156 (56.7%)	131 (47.6%)	117 (42.5%)	106 (38.5%)
M. Firefox	3,7k (49.3%)	3,2k (42.2%)	2,9k (38.4%)	2,7k (35.7%)
MyFaces	38 (48.1%)	30 (38.0%)	27 (34.2%)	25 (31.6%)
OpenOffice	1,5k (48.3%)	1,2k (39.6%)	1,1k (35.4%)	968 (32.1%)
PDFBox	84 (54.2%)	74 (47.7%)	71 (45.8%)	63 (40.6%)
Spark	212 (49.3%)	170 (39.5%)	151 (35.1%)	138 (32.1%)
Wicket	69 (46.3%)	61 (40.9%)	57 (38.3%)	47 (31.5%)
Struts	35 (40.7%)	28 (32.6%)	24 (27.9%)	24 (27.9%)
Total	**21,3k (50.6%)**	**18,5k (43.9%)**	**17k (40.2%)**	**15,8k (37.5%)**

All Top-N results, for N={5, 6, ..., 30}, are available in the replication package [87].

7.3.5 Observed Behavior Identification

In order to answer our research questions, we need to identify the terms corresponding to the system's observed behavior (OB) for the bug reports that require reformulation (*i.e.*, for the *low-quality* queries), just as a potential user would do. In contrast, the bug title is found in a separate field within the bug report and its identification is trivial.

We randomly sampled 749 bug reports for which Lucene fails to retrieve duplicates within the top-5 results (see Table 7.3). The sample includes reports for each project, and excludes reports referring to new features and enhancements (*i.e.*, non-bugs) – see the replication package for the full list of manually excluded reports [87]. The amount of sampled reports represents 3.5% of the 21,346 *low-quality* queries for N=5. The query set also contains a subsample of the queries that fail to retrieve the duplicates in top-N for N={6, ..., 30}. Having a relatively small percentage of reports in our overall sample comes from the large query set for Eclipse, Firefox, and OpenOffice. Excluding these projects, the sample would represent 30% of the *low-quality* queries for N=5.

Two PhD and one master students (*a.k.a.* coders) conducted qualitative text *coding* [199] on all 749 bug reports. The reports were distributed among the coders in such a way that each report was coded by one coder. The coders had to select any part of the text (*e.g.*, words, clauses, or sentences) in the reports' title and description that corresponded to the OB. This task was performed using the text annotation tool BRAT [15].

We summarize the main criteria used by the coders to tag the OB in the bug report title and description (the full list can be found in the replication package [87]):

- The coding focused only on natural language content written by the users, ignoring code snippets, stack traces, or program logs. However, the natural language referencing this information may indicate OB and was allowed for coding. An example of this case is: "*When I click the File menu, I get the following error and stack trace: ...*".

- Internal system behavior, described by the reporters, was also allowed for coding, for example: "*The open() method in the class FileMenu reads the options from the file...*".

- Descriptions of graphical user interface issues can be considered as OB, for example: *"The menu's color is too light, it should be darker"*.

- Uninformative sentences, such as "*The File menu does not work*" are insufficient to be considered OB. There must be a clear description of the software's OB, for example: "*The File menu doesn't open when I click on it*".

- Explanations of code attached to the bug reports are not considered OB, for example: *"The attached code defines the openMenu() method, which iterates on the options..."*.

Table 7.3: Number of sampled/coded bug reports and reduced queries.

Project	**# of coded BRs**[a]	**# of BRs with OB**[b]	**# of reduced queries**
Accumulo	27 (87.1%)	25 (92.6%)	25
Ambari	7 (28.0%)	7 (100%)	7
ActiveMQ	35 (64.8%)	35 (100%)	35
Cassandra	18 (9.3%)	18 (100%)	18
Cordova	29 (27.4%)	29 (100%)	29
Continuum	29 (93.5%)	29 (100%)	29
Drill	34 (33.3%)	34 (100%)	34
Eclipse	96 (0.7%)	93 (96.9%)	93
Groovy	32 (66.7%)	31 (96.9%)	31
Hadoop	26 (32.9%)	22 (84.6%)	22
Hbase	27 (71.1%)	27 (100%)	27
Hive	17 (10.6%)	17 (100%)	17
Maven	35 (22.4%)	32 (91.4%)	32
M. Firefox	102 (2.7%)	99 (97.1%)	99
MyFaces	31 (81.6%)	30 (96.8%)	30
OpenOffice	113 (7.8%)	111 (98.2%)	110
PDFBox	16 (19.0%)	15 (93.8%)	15
Spark	14 (6.6%)	13 (92.9%)	13
Wicket	32 (46.4%)	32 (100%)	32
Struts	29 (82.9%)	28 (96.6%)	28
Total	**749 (3.5%)**	**727 (97.1%)**	**726**

[a]Proportions with respect to the total # of *low-quality* queries for each project (for N=5). [b]Proportions with respect to the total # of coded BRs.

Overall, 727 (*i.e.*, 97.1%) of the tagged bug reports describe an OB (see Table 7.3). The OB coding required significant manual effort for all 749 reports, however, in an actual usage scenario, a user only needs to select the OB terms from a single report, which takes seconds.

Figures 7.1 and 7.2 show examples of coded bug report descriptions, where their OB part is highlighted in yellow. In practical scenarios, the user would only select the highlighted text and use it for reformulation (*i.e.*, in the case of OB-DD). Note that the OB may be described in non-contiguous parts of the text, including in parts of the title.

We made the choice of having one coder for each bug report in order to maximize the number of queries used in the evaluation. However, our past research that uses multiple coders per bug report reveals high agreement between coders (*i.e.*, +80% *kappa* [83]), hence, we expect minimal differences between single- and multiple-coders-based coding. In any case, our future work will investigate and assess the robustness of OB-DD and OBT-DD with respect to these differences.

7.3.6 Query Reduction Strategies and Metrics

We reformulate each *initial query*, by using each reformulation strategy, as follows:

- For OB-DD, we concatenate the parts of the text corresponding to OB only, from the bug report title and description, and remove the rest of the textual description.
- For BT-DD, we consider the bug title only and remove the rest of the textual description.
- For OBT-DD, we retain the title as is, and the OB parts from the bug report, while removing the rest of the textual description.

We reformulated all 727 bug reports that describe an OB. We call these queries *reduced queries*. However, after preprocessing, one OpenOffice query (from bug report #113872) became empty because its OB contained stop words, numbers, and special characters only. Hence, in total, we obtained 726 reduced queries (see Table 7.3). For these 726 queries, we also used BT-DD and OBT-DD to produce reduced queries. For each initial query, we generated *three* reduced queries.

We executed each one of the 726 *initial* and *reduced queries* with the Lucene-based duplicate detector. We measured the performance of duplicate retrieval using *Recall Rate@N*, *i.e.*, the proportion of queries for which a DD approach returns at least one duplicate report within the top-N candidates. This is one of the most commonly used measures in past duplicate detection research (see Section 7.4) and is ideal for assessing the performance of

Bug report title:
Heading lines do not show in MsWord export

Bug report description:
When exporting a document to MsWord, in my case in order to submit it to the LuLu website, the 'Heading Lines' ('Righe d'intestazione' in the Italian version I use), so painstakingly inserted, do not show in the Word export.

Figure 7.1: OB content in in OpenOffice's bug report #104924.

Bug report title:
Security Wizard: Delete ATS call should be non-blocking

Bug report description:
Before Start All Services web-client triggers Delete ATS call. This call should not be marked with jquery ajax async flag as false. Doing so makes the call blocking and page becomes unresponsive untill call returns

Figure 7.2: OB content in Ambari's bug report #4968.

DD techniques as, in practice, users would likely inspect no more than top-N results before stopping the DD search (*i.e.*, when they find a duplicate or they are confident enough that there are no duplicates). For the sake of completeness, we also measured the performance of the strategies based on MRR and MAP, which are other common metrics used in DD research (see Section 7.4). The replication package contains the full MRR/MAP results [87].

As mentioned before, our empirical evaluation mimics an actual usage scenario, where the user issues the initial query and inspects the N returned bug reports (step #1). If she does not find a duplicate, then she makes a choice whether to retrieve additional N bug reports with the same query or use any strategy to reformulate the query (*i.e.*, OB-DD, BT-DD, or OBT-DD) and then retrieve additional N bug reports (step #2). In both cases (reformulation and no reformulation in step #2), the N bug reports originally returned in top N (in step #1) are removed from the ranked list and then *Recall Rate@N* is computed for the initial query and the reformulated query as well. We repeat this process for all queries, for N={5, 6, ..., 30}. In the end, if the *Recall Rate@N* for the reformulated queries is higher than the one for the initial queries, we can conclude that the reformulation is the better strategy. Conversely, if the measures are the other way around, we can conclude that it is not worth reformulating the query, as there is no gain over just simply investigating N more results returned by the initial query.

We measured the magnitude of improvement for the *Recall Rate@N* measurements, by computing the change percentage of the metric before (M_b) and after reformulation (M_a), *i.e.*, Improvement(M) = $(M_a - M_b)/M_b$. We aim at maximizing the improvement, avoiding negative values, which would mean deterioration rather than improvement.

We assessed the statistical significance of our measures using the Mann-Whitney test [122], a paired non-parametric test that does not assume normal distributions (as in our case). This method was used to test if a measure M, when applying a reformulation strategy (M_a), is higher than when using no reformulation (M_b). We carried out the test on the paired *Recall Rate@N* values that we collected across the 26 thresholds. We define the null hypothesis as $H_0 : M_b \geq M_a$, and the alternative hypothesis as $H_1 : M_b < M_a$. We applied the test with a 95% confidence level, thus rejecting the null hypothesis, in favor of the alternative, if p-value $< 5\%$.

We also estimated the magnitude of the difference between *Recall Rate@N* measures, by using Cliff's Delta (d), a non-parametric measure of the *effect size* for ordinal data that does not assume normal distributions [92, 113]. We applied this measure with a 95% confidence level. The *effect size* (*i.e.*, *Recall Rate@N* difference) is interpreted as *negligible* if $|d| < 0.147$, *small* if $0.147 \leq |d| < 0.33$, *medium* if $0.33 \leq |d| < 0.474$, and *large* if $|d| \geq 0.474$ [192, 113].

7.3.7 Results and Discussion

We present and discuss the evaluation results for the three query reformulation strategies. For space limitations, we present the results for N={5, 10, 15, 20, 15, 30} only and focus our analysis on Recall Rate@N. The results for the full threshold set, including the MRR/MAP measurements, are available in the online replication package [87].

Table 7.4: Recall Rate@N results for each reformulation strategy.

N	$\|Q\|$	No reformulation	Reformulation			Improvement		
			OB-DD	BT-DD	OB-DD	OB-DD	BT-DD	OB-DD
5	726	90 (12.4%)	139 (19.1%)	125 (17.2%)	150 (20.7%)	54.4%	38.9%	66.7%
10	636	92 (14.5%)	129 (20.3%)	130 (20.4%)	151 (23.7%)	40.2%	41.3%	64.1%
15	594	92 (15.5%)	137 (23.1%)	141 (23.7%)	154 (25.9%)	48.9%	53.3%	67.4%
20	544	78 (14.3%)	135 (24.8%)	137 (25.2%)	153 (28.1%)	73.1%	75.6%	96.2%
25	528	92 (17.4%)	135 (25.6%)	147 (27.8%)	155 (29.4%)	46.7%	59.8%	68.5%
30	502	87 (17.3%)	136 (27.1%)	141 (28.1%)	155 (30.9%)	56.3%	62.1%	78.2%
Avg	**583**	**87 (15.1%)**	**136 (23.6%)**	**139 (24.1%)**	**155 (26.8%)**	**56.6%**	**59.6%**	**78.0%**

The average values are across the 26 thresholds (*i.e.*, N={5, 6, ..., 30}).

Duplicate Detection Performance. Table 7.4 shows the Recall Rate@N (RR@N) results for the initial and reduced queries when using OB-DD, BT-DD, and OBT-DD. Remember that we remove the original top-N results returned by the initial queries and compare the ability of both the initial and reduced queries on retrieving the duplicates within the next top-N results. This means that for each N, we have a different number of queries (see Table 7.4). Note that we compare the three strategies on the same set of bug reports used for creating the initial and reduced queries.

When using the initial queries (*i.e.*, no reformulation), Lucene retrieves the duplicates in top N for 15.1% of the queries, on average. When reformulating the queries via OB-DD, Lucene retrieves at least one duplicate report for 23.6% of the queries, on average. This means that 8.5% more queries (on average) return duplicates when using OB-DD, compared to no reformulation, which represents a 56.6% improvement. When reducing the queries based on BT (*i.e.*, BT-DD), Lucene retrieves at least one duplicate for 24.1% of the queries, on average, which represents 9% more queries with retrieved duplicates (*i.e.*, 59.6% improvement). The highest performance is achieved when combining both the OB and BT for reducing the queries, as this strategy leads to 26.8% of the queries (on average) to retrieve duplicates in top N: 11.7% more queries or 78% improvement, on average, with respect to no reformulation. The three reformulation strategies achieve RR@N improvement (and 63.4%-317.9% MRR/MAP improvement, see the replication package [87]), compared to no reformulation, for each of the 26 thresholds N. The RR@N achieved by all three strategies is statistically significant higher than the one achieved by the initial queries, across the 26 thresholds N (Mann-Whitney, p-value $< 5\%$). All strategies achieve a *large* RR@N improvement according to our *effect size* analysis based on Cliff's delta ($|d| \geq 0.474$) – see the replication package for the full statistical test results [87].

We answer our first research question (**RQ1**) positively, as the RR@N indicates that using all reformulation strategies (*i.e.*, OB-DD, BT-DD, and OBT-DD) lead to the retrieval of more duplicates bug reports than with no reformulation. As for our second research question (**RQ2**), we found that OBT-DD achieves a statistically significant higher RR@N compared to OB-DD and BT-DD (Mann-Whitney, p-value $< 5\%$), and the RR@N improvement is *large* ($|d| = 0.536$) and *medium* ($|d| = 0.411$), respectively. We also found that BT-DD's RR@N is statistically significant higher than OB-DD's RR@N (Mann-Whitney, p-value $<$ 5%). However, this RR@N improvement is *negligible* ($|d| = 0.129$). The results indicate that OB-DD and BT-DD achieve comparable performance. We conclude that OBT-DD leads to the best detection performance, in terms of number of queries with retrieved duplicates, compared to using OB-DD and BT-DD.

The results also reveal an interesting issue. We expected that Lucene would retrieve more duplicates (with or without reformulation) as N increases. As we report in Table 7.4, Lucene retrieves almost the same number of duplicates across thresholds N. To better understand this phenomenon and the detection improvements, we analyze the results in more detail.

Successful vs. Unsuccessful Queries. During query reformulation, there is always a trade-off: some queries become *successful* while others become *unsuccessful*. A good reformulation strategy would lead to more *successful* queries (*i.e.*, retrieve duplicates) than *unsuccessful* queries (*i.e.*, fail to retrieve duplicates), compared to the initial ones. All three reformulation strategies achieve that, but we aim to understand better the trade-offs.

We refer to all the queries that retrieve duplicates in top N as *successful queries*, and to those that do not retrieve duplicates as *unsuccessful queries*. The ideal reformulation strategy would preserve the *successful queries* (*i.e.*, an initial *successful* query, which reformulated remains *successful*, *a.k.a.* *successful* → *successful*), while converting all (or at least some) of the *unsuccessful* initial queries into *successful* ones (*i.e.*, *unsuccessful* → *successful*). In other words, we want to avoid the situation when *successful* queries turn *unsuccessful* (*i.e.*, *successful* → *unsuccessful*) via the reformulation.

Table 7.5 shows that OB-DD and BT-DD transform about the same number of *successful* queries into *unsuccessful* ones, on average (*i.e.*, approx. 45 and 48, out of 87 *good* queries, respectively). However, BT-DD converts more *unsuccessful* queries into *successful* ones (on average), compared to OB-DD (*i.e.*, 99 vs. 93 queries, respectively). OB-DD preserves more of the *successful* queries as *successful* than BT-DD does, on average (*i.e.*, 43 vs 39, respectively), while preserving more of the *unsuccessful* queries as *unsuccessful* (*i.e.*, 403 vs 397, respectively), which is undesirable. These results support BT-DD's higher performance against OB-DD. As for OBT-DD, Table 7.5 reveals that it outperforms the other two strategies in all the aspects. Specifically, 39 of 87 *successful* queries turn *unsuccessful*, while 107 *unsuccessful* queries become *successful*. OBT-DD preserves 48 of the *successful* queries as *successful*, and 390 of the *unsuccessful* queries as *unsuccessful*, on average.

The *successful* → *unsuccessful* cases need further analysis. Being query reduction strategies, the assumption is that OB-, BT-, and OBT-DD eliminate parts of the bug report (different than BT and OB) that should not be removed. Identifying these parts helps us improve the strategies in the future.

Analysis of Successful → Unsuccessful Queries. We analyzed the set of *successful* queries that turned *unsuccessful* for N=5 when using the reformulation strategies. This set corresponds to 55 unique queries/reports. From this set, 30 queries (*i.e.*, 54.5%) do not

Table 7.5: Number of Successful and Unsuccessful Queries.

(a) OB-DD

N	**U $\rightarrow$ S**	**S $\rightarrow$ U**	**S $\rightarrow$ S**	**U $\rightarrow$ U**
5	89 (12.3%)	40 (5.5%)	50 (6.9%)	547 (75.3%)
10	88 (13.8%)	51 (8.0%)	41 (6.4%)	456 (71.7%)
15	92 (15.5%)	47 (7.9%)	45 (7.6%)	410 (69.0%)
20	96 (17.6%)	39 (7.2%)	39 (7.2%)	370 (68.0%)
25	93 (17.6%)	50 (9.5%)	42 (8.0%)	343 (65.0%)
30	97 (19.3%)	48 (9.6%)	39 (7.8%)	318 (63.3%)
Avg.*	**93 (16.3%)**	**45 (7.7%)**	**43 (7.4%)**	**403 (68.7%)**

(b) BT-DD

N	**U $\rightarrow$ S**	**S $\rightarrow$ U**	**S $\rightarrow$ S**	**U $\rightarrow$ U**
5	80 (11.0%)	45 (6.2%)	45 (6.2%)	556 (76.6%)
10	96 (15.1%)	58 (9.1%)	34 (5.3%)	448 (70.4%)
15	101 (17.0%)	52 (8.8%)	40 (6.7%)	401 (67.5%)
20	100 (18.4%)	41 (7.5%)	37 (6.8%)	366 (67.3%)
25	104 (19.7%)	49 (9.3%)	43 (8.1%)	332 (62.9%)
30	101 (20.1%)	47 (9.4%)	40 (8.0%)	314 (62.5%)
Avg.*	**99 (17.3%)**	**48 (8.3%)**	**39 (6.8%)**	**397 (67.6%)**

(c) OBT-DD

N	**U $\rightarrow$ S**	**S $\rightarrow$ U**	**S $\rightarrow$ S**	**U $\rightarrow$ U**
5	100 (13.8%)	40 (5.5%)	50 (6.9%)	536 (73.8%)
10	103 (16.2%)	44 (6.9%)	48 (7.5%)	441 (69.3%)
15	100 (16.8%)	38 (6.4%)	54 (9.1%)	402 (67.7%)
20	109 (20.0%)	34 (6.3%)	44 (8.1%)	357 (65.6%)
25	108 (20.5%)	45 (8.5%)	47 (8.9%)	328 (62.1%)
30	111 (22.1%)	43 (8.6%)	44 (8.8%)	304 (60.6%)
Avg.*	**107 (18.5%)**	**39 (6.8%)**	**48 (8.3%)**	**390 (66.4%)**

Number and proportion of Successful (**S**) and Unsuccessful (**U**) queries before reformulation that turned Unsuccessful (**U**) and Successful (**S**) after reformulation.

* Average values across the 26 thresholds (*i.e.*, N={5, 6, ..., 30}).

Percentage values with respect to the # of queries for each N, from Table 7.4.

retrieve duplicates with any of the reformulation strategies, which means that the strategies removed important information from the bug report, leading to non-retrieved duplicates. Thirteen queries (*i.e.*, 23.6%) retrieve duplicates only with OB-DD and OBT-DD, which means that for these cases, the OB is required for better duplicate retrieval. Eight (*i.e.*, 14.5%) retrieve duplicates when using BT-DD only, two (*i.e.*, 4.6%) when using BT-DD and OBT-DD, and two more (*i.e.*, 4.6%) when using OB-DD only.

We manually analyzed the 30 queries that fail to retrieve duplicates with any of the strategies. Among these, we found that the main source of important terms for retrieval (besides OB and BT) is the steps to reproduce the bug (*i.e.*, S2Rs).

For 12 *successful* queries, key S2R terms from the bug report were removed by the reformulation strategies. These terms are also present in the duplicate reports, hence, removing them from the initial query leads to non-retrieved duplicates. For example, the Firefox bug report #619430 [18] duplicates the report #611991 [17]. In the former one (*i.e.*, the query), the OB is described by the sentence *"font size menu drop down is not displayed"*, which is repeated three times in the bug description. The terms *font*, *size*, and *menu* are relevant because they appear in the duplicate report. However, phrases such as *"Under preferences"* or *"under content"*, present only in S2R from report #619430 and shared with the duplicate report, were removed, which led to not retrieving the duplicate.

Overall, we expected S2R terms to be common among bugs. For example, *"open a fil*e" may be a necessary step for replicating many bugs. However, we observed that the S2Rs often contains terms that better indicate what the bug is about and includes key terms for retrieval. We also hypothesized that the bugs that can be reproduced in more than one way, would benefit from removing the distinct S2Rs in the reports, while preserving the OB. However, we did not find any of these cases in the analyzed reports.

For several of the analyzed queries, we found that sources, such as code snippets (in eight bug reports), stack traces (in six reports), and the software's expected behavior (in three reports), contained relevant terms that should not be removed by the reformulation. We hypothesize that code snippets are good candidates to be preserved in the reformulated queries. Our future work will investigate ways of leveraging code snippets in combination with the BT and OB/S2R descriptions for reformulating queries.

Finally, in seven of the analyzed queries, we also observed that the frequency of some terms, present in the OB and/or BT and relevant for retrieval, decreases when reformulating the query with the strategies. Given that Lucene is based on term frequencies, we conjecture that increasing the frequency of these terms may lead to better retrieval. In the future, we will increase the frequency of OB/BT terms based on their entire document frequency.

7.3.8 Threats to Validity

The main threat to the *construct validity* of the evaluation is the subjectivity introduced in the coded set of bug reports, as each bug report was coded by a single coder. We made this choice to maximize the number of coded bug reports. Since our past research revealed high agreement (*i.e.*, +80% *kappa* [83]) when each report is coded by multiple coders, we expect minimal differences in the results when using multiple coders.

In order to mitigate threats to the *conclusion validity*, we compared the performance of the initial and reduced queries using Recall Rate@N and MRR/MAP, metrics widely used in DD research [130, 78]. We also analyzed the trade-offs of the reformulation strategies, to further strengthen our conclusions. As in prior research [83, 116], we defined two query categories (*i.e.*, *successful* and *unsuccessful*) and analyzed the query transitions between categories before and after reformulation.

The *internal validity* of our evaluation is mainly affected by our selected data set. We built such a data set based on two existing ones, previously used in duplicate detection research [82, 209]. This data set contains bug reports with duplicates, used as queries, with their set of duplicate reports in the corresponding duplicate buckets. While these buckets were extracted from the projects' issue tracker, this does not guarantee that all reports in a bucket are duplicates, either because of incorrect references in the issue tracker or the way we created the buckets (see Section 7.3.3). To mitigate this issue, we recreated the queries from the bug reports used in the SDS data set by Sun *et al.* [209]. We identified inconsistencies between the data set we constructed and SDS, based on the same bug reports. We manually checked a subset of the inconsistencies in favor of our data set. Given the small size of the inconsistent buckets, compared to the size of SDS, we believe that the impact on the results is minimal. As for the CDS data set by Chaparro *et al.* [83], we used it as is.

We addressed the *external validity* of our evaluation by using a large set of bug reports from the issue trackers of 20 open source projects that span different domains and software types. We used one ranking-based duplicate detector, based on Lucene. In the future, we will investigate how the reformulation strategies help other DD approaches on retrieving more duplicates. Those techniques that use BT as a separate feature may benefit less from BT-DD, but we believe that OB-DD and OBT-DD would still lead to better retrieval.

7.4 Related Work

Duplicate Bug Report Detection. Duplicate bug report detection (DD) has received significant attention in the past years and dozens of approaches have been proposed using a variety of text retrieval, natural language processing, and machine learning techniques. See Chapter 2 for more details.

The proposed reformulation strategies are designed to work with most DD approaches, but the evaluation we present in this chapter focuses on ranking approaches, which use Text Retrieval. Evaluating the strategies on the other approaches is subject of future work.

Most issue trackers provide two fields that capture the natural language bug description reported by the user, *i.e.*, *title* and *description*. All the approaches reviewed in Section 2.4 combine the textual information found in both fields [67], with a few exceptions. Two approaches use the report's title only [57, 177] and Lerch *et al.* [144] use only a part of the description, specifically, stack traces. As OB-DD and OBT-DD are based on the description, they would not work with these approaches. The approaches from Amoui *et al.* [57] and Prifti *et al.* [177] are similar to BT-DD, which only use the bug title. However, they are DD techniques rather than query reformulation approaches.

Related to our work, Amoui *et al.* [57] used the observed results (*a.k.a.* OB) and the steps to reproduce (S2Rs) from the bug report as a source of extra information for retrieving duplicates at BlackBerry. This information is found inside a field called *first_email*, which is unique to BlackBerry's issue tracker and includes the OB and S2Rs by convention. However, this is not the case in most bug reports submitted to open-source projects [129, 83]. Most issue trackers used by these projects do not enforce the existence of this information in the bug report's textual description [83]. Amoui *et al.*'s research does not study the effect of these sources of information on duplicate detection or focus on query reformulation.

Empirical evaluations of DD techniques have used various metrics, depending on the approach type. Ranking approaches are usually evaluated using standard Information Retrieval measures, such as precision, recall, and MAP [70, 72, 119, 120, 140, 144, 149, 183, 184, 209, 57]. These metrics are suitable when all duplicate bug reports in the corpus are considered useful for retrieval since it is possible for a new bug report to have more than one duplicate. However, many researchers reckon that it is sufficient to find a single duplicate, since the triagers' goal is to mark the new report as duplicate. In this case, MRR and Recall Rate@N are usually used [57, 72, 144, 168, 177, 193, 210, 183, 184, 209, 211, 223, 244]. We contend that Recall Rate@N is better suited for evaluating DD approaches, since the triager is likely to inspect the top-N results only (rather than the entire ranking), hence, it is the measure we mainly use in our evaluation (see Section 7.3). For binary and decision-making approaches, accuracy and true positive/negative rates are typically used [53, 119, 140, 216].

Notable is the work by Hindle *et al.* [120], who propose a continuous querying approach. While not focusing on query reformulation as we do, in Hindle *et al.*'s approach, queries are generated as the user types the bug description. Each new word will trigger the creation of a new query, which consists of the new word and all the words typed before. It is possible that, as the user enters the bug description, a query would be formulated in such a way that roughly corresponds to the OB. For example, if she starts describing the OB in a contiguous sequence of words. In that case, the query would be similar to our reformulated query. However, differently to that approach, we consider already-submitted bug reports as queries and we do not assume that the OB description is sequential.

Finally, it is worth noting that the proposed reformulation approach addresses the cases when duplicates are hard to retrieve. For example, for the new reports that *just-in-time* retrieval tools in issue trackers fail to identify as duplicates [184].

Query Reformulation for Duplicate Detection. Duplicate detection techniques offer no guidance on what to do when a duplicate is not retrieved within the top-N candidates. We argue that *query reformulation* can be applied in such situations to increase the triager's confidence in the retrieval. Query reformulation has not been applied to the problem of detecting duplicates of bug reports. However, researchers have investigated the impact of query reformulation on other software engineering tasks based on text retrieval methods, especially on code search and concept location (see Section 2.6).

Query reduction is the category that our approach falls into [90]. Our OB-DD reformulation strategy is motivated by our prior work that proposed a query reformulation approach for *low-quality* queries in bug localization based on OB [83, 86]. While the reformulation strategies are similar, their applications and evaluations are quite different. In bug localization, the buggy code element is guaranteed to exist in the code corpus, and the developer must find it eventually. However, in duplicate detection, a duplicate may or may not exist in the bug report corpus, and at one point the developer must choose to stop looking through the list of candidates. These are two distinct scenarios that must be evaluated differently.

7.5 Conclusions

We argue that duplicate bug report detection approaches, based on text retrieval, should be viewed as a two-step process. The process should allow for query reformulation, which may lead to the retrieval of more duplicates or higher user confidence when none are retrieved. The challenge is that any reformulation strategy should be tool agnostic and independent of the user's knowledge.

We hypothesized that the description of the software's observed behavior (*i.e.*, OB) in bug reports and the bug report title (BT) contain relevant information that is bug specific, while other parts of the description include less specific information (*i.e.*, noise) with respect to duplicate retrieval. Based on this observation, we defined three query reformulation techniques that are based on the user selecting the OB part of the bug description and/or the BT. The reformulation strategies are simple, they do not depend on any information outside the bug report, and they can be applied in more than 97% of the cases. Better yet, they retrieve more duplicates (*i.e.*, for 23.6%-26.8% of the cases, on average, equivalent to a 56.6%-78% avg. improvement) than inspecting more candidate reports retrieved by the initial query. We conclude that triagers should use both the BT and OB from the bug report to reformulate the initial *low-quality* query and expect to find duplicates for more cases than without reformulation. The results bear evidence in support of our conjecture about the OB, BT, and the proposed two-step paradigm for duplicate detection, at least when using Lucene as a duplicate bug report detector.

On the other hand, the results also revealed that relevant information is present in the steps to reproduce (*i.e.*, S2Rs) and code snippets, which may improve duplicate retrieval. This observation guides our future research plans. Specifically, we will evaluate additional reformulation strategies that would combine information from the BT, OB, S2Rs, and code snippets. Any of such strategies would still meet the main challenge of being independent of the underlying duplicate detector and other information external to the bug report. With that in mind, we plan to also evaluate the proposed reformulation strategies with duplicate detectors more sophisticated than Lucene. Finally, our future work will focus on automatically reducing queries based on specific bug descriptions.

7.6 Acknowledgments

This chapter was published and presented at the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19). The paper published in this conference [86] is a collaborative effort with Juan Manuel Florez and Unnati Singh from The University of Texas at Dallas. Unnati Singh, Oscar Chaparro, and Juan Manuel Florez contributed to the design, implementation, and evaluation of the proposed approach.

CHAPTER 8
CONCLUSIONS AND FINAL REMARKS

Bug descriptions/reports are the main source of information for developers to triage and fix the bugs in the software. Unfortunately, bug descriptions are often unclear, incomplete, and/or ambiguous, so much so that developers are unable to replicate the problems let alone fix the bugs in the code. Current bug reporting technology (*i.e.*, issue/bug tracking systems), which is mostly passive and does not verify the information provided by the reporters, helps little in collecting high-quality bug information and improving the quality of bug descriptions.

In this dissertation, we aimed to improve the quality of bug descriptions by automatically identifying, verifying, and processing the most relevant information in bug descriptions (*i.e.*, the *observed behavior*, the *expected behavior*, and the *steps to reproduce* the bug). At the same time, we aimed to improve the accuracy of automated bug localization and duplicate bug report detection techniques, by reformulating low-quality queries using such information. We proposed automated techniques that combine software analysis, natural language processing, and machine learning for the automated analysis of bug descriptions. These techniques (1) provide feedback to reporters, useful to improve the quality of bug descriptions, and (2) reformulate queries, effective for retrieving buggy code artifacts and duplicate bug reports.

The specific contributions of this dissertation are as follows:

1. We documented a catalog of 154 discourse patterns that reporters recurrently use to express the *observed behavior*, the *expected behavior*, and the *steps to reproduce* in bug descriptions. This catalog is the result of a qualitative analysis, based on an open coding methodology, of a large set of bug descriptions from nine software projects. The discourse patterns indicate that bug descriptions present textual regularities that enable the automatic analysis of such documents.

2. We proposed and evaluated DeMIBuD, an approach that leverages discourse patterns and additional textual information, to automatically detect missing expected

behavior and steps to reproduce in bug descriptions. DeMIBuD is based on natural language processing and machine learning techniques, and was evaluated on a large set of bug reports from nine software projects. The results reveal that DeMIBuD accurately detects missing bug information, even when different data is used for training. DeMIBuD can be deployed in new projects without the need of retraining, to alert reporters about missing information when they are submitting their bug descriptions.

3. We proposed and evaluated Euler, an approach that automatically identifies the steps to reproduce in bug descriptions, assesses the quality of each step, and provides actionable feedback to reporters about problems with the steps to reproduce. Euler performs automated quality assessment based on neural sequence labeling of natural language, dynamic software analysis, graph modeling, and textual matching. External evaluators judged the feedback produced by Euler on bug reports from Android apps. The results indicate that Euler accurately identifies the steps from the bug descriptions and accurately infers the steps missed by the reporter. According to the evaluators, Euler's feedback is correct in most of the cases.

4. We proposed and evaluated a set of query reformulation strategies to improve text-retrieval-based bug localization and duplicate bug report detection. These strategies combine and use the content corresponding to the observed behavior, the expected behavior, and the steps to reproduce, as input queries to state-of-the-art text retrieval engines. We used the proposed strategies to reformulate a large set of (initial) queries, generated from entire bug descriptions, which fail to retrieve the buggy code artifacts or duplicate bug reports. The results indicate that the strategies are more effective than using no reformulation. The reformulation strategies are highly applicable, easy to use, and they do not require external or additional information to the one already reported in the bug description.

The work in this dissertation achieves important milestones towards improving the quality of bug descriptions and supporting developers on bug resolution tasks that rely on such descriptions. At the same time, this work reveals challenges and opportunities of future research towards the *automated management of bug reports*.

The major challenge for future work is transforming passive bug reporting (*i.e.*, the current paradigm) into active and automated bug reporting and management. Given the current advances in natural language processing and generation as well as other areas in artificial intelligence, interactive conversation systems (*e.g.*, chatbots or virtual assistants) represent the most logical step towards the next generation of issue/bug trackers. The expectation for a conversation system is to automatically interpret bug-related information that the user may report (*e.g.*, the observed behavior, expected behavior, steps to reproduce, system input, *etc.*), either textually, verbally or graphically (*e.g.*, via screenshots or videos). The system will generate feedback and formulate questions to collect complete, clear, and correct bug-related information, considering the context of the reporter and information from existing bug reports, while supporting developers towards better and automated bug triage, reproduction, localization, and fixing. In other words, the goal of this system will be to *automate the full process of bug report management*.

Designing and evaluating an interactive and intelligent bug report management system (BRMS) is challenging, given the variety of reporters, types of bugs, and software systems that exist. Different software users have different bug reporting expertise and knowledge about the software system being used. Some users are external to the software project, and some others are internal (*e.g.*, developers). These factors impact the quality of the information provided, and the way a BRMS should interact with the users. At the same, some type of bugs, such as performance or security issues, are likely to require more details from the user, and deeper analysis by the BRMS, compared to other bug types, such as those related to functional problems. The challenge for a BRMS is also supporting different software platforms, for example, mobile, web-based, and desktop applications as well as libraries and other types of software.

Realizing a BRMS will require extensive research on designing and combining techniques from software analysis, natural language processing, machine learning, and human-computer interaction. The results of this dissertation will help realize the development of a BRMS. One specific challenge in our future work is the definition of an actionable model for assessing the quality of bug reports, which is currently inexistent in the body of knowledge and practice of the field. In this dissertation, we took the first step in this direction by defining a taxonomy of problems in the steps to reproduce and by proposing an approach to identify them. Another challenge, from the bug reproduction perspective, is to automatically relate and verify the steps to reproduce, the observed behavior, the expected behavior, and system input information found in the bug reports, especially for implicit bugs that do not lead to errors or crashes. For example, automatically comparing the expected behavior against the system's observed behavior is challenging because these can take different forms, depending on the software system (*e.g.*, a desktop system vs. a library) or functionalities within the system (*i.e.*, generating a data report vs. receiving user input). Another challenge is the integration of different solutions (*e.g.*, duplicate bug report detection, bug reproduction, and bug localization approaches), together with a conversation agent for (1) interpreting the information reported by the user, (2) determining the quality of this information, (2) analyzing existing software data from different sources (*e.g.*, issue or code repositories), and (4) generating questions and feedback in natural language. Finally, the evaluation of a BRMS will require extensive data collection and tagging (*e.g.*, reporter-developer conversations in existing bug reports) and controlled user studies. The goal is to design a system that is effective, accurate, and useful in managing bug reports.

REFERENCES

[1] "An open letter to GitHub from the maintainers of open source projects," 2016. [Online]. Available: https://github.com/dear-github/dear-github

[2] "Apache Web Server," 2017. [Online]. Available: http://httpd.apache.org/

[3] "Docker," 2017. [Online]. Available: https://www.docker.com/

[4] "Eclipse," 2017. [Online]. Available: http://www.eclipse.org

[5] "Facebook," 2017. [Online]. Available: https://www.facebook.com/

[6] "GnuCash's bug report #256," 2017. [Online]. Available: https://tinyurl.com/y3df92g7

[7] "Hibernate," 2017. [Online]. Available: http://hibernate.org/

[8] "LibreOffice," 2017. [Online]. Available: https://www.libreoffice.org/

[9] "Mozilla Firefox," 2017. [Online]. Available: https://www.mozilla.org/en-US/firefox/

[10] "Mozilla Support: Contributors Guide to Writing a Good Bug," 2017. [Online]. Available: https://goo.gl/ykkrUJ

[11] "OpenMRS," 2017. [Online]. Available: http://openmrs.org

[12] "Perl interface to Snowball stemmers," 2017. [Online]. Available: https://tinyurl.com/98lk9t

[13] "Wordpress Android," 2017. [Online]. Available: https://apps.wordpress.com/mobile/

[14] "Apache Lucene," 2018. [Online]. Available: https://lucene.apache.org/

[15] "BRAT," 2018. [Online]. Available: https://brat.nlplab.org/

[16] "Bugzilla," 2018. [Online]. Available: https://www.bugzilla.org/

[17] "Firefox bug report #611991," 2018. [Online]. Available: https://tinyurl.com/y3r9ljnb

[18] "Firefox bug report #619430," 2018. [Online]. Available: https://tinyurl.com/y492nva2

[19] "Jira," 2018. [Online]. Available: https://tinyurl.com/l5c9ctc

[20] "Lucene's DefaultSimilarity Javadoc," 2018. [Online]. Available: https://tinyurl.com/y84vawy5

[21] "Lucene's TFIDFSimilarity Javadoc," 2018. [Online]. Available: https://tinyurl.com/ybhqqrqm

[22] "Aard Dictionary," 2019. [Online]. Available: https://tinyurl.com/mwpxshz

[23] "Aard Dictionary's bug report #104," 2019. [Online]. Available: https://tinyurl.com/y3xhlky3

[24] "Aard Dictionary's bug report #81," 2019. [Online]. Available: https://tinyurl.com/y3xvqf3j

[25] "AppSee," 2019. [Online]. Available: https://www.appsee.com

[26] "BugClipper," 2019. [Online]. Available: http://bugclipper.com

[27] "Droid Weight," 2019. [Online]. Available: https://tinyurl.com/lxazk36

[28] "GitHub's Issues," 2019. [Online]. Available: https://help.github.com/articles/about-issues/

[29] "GnuCash," 2019. [Online]. Available: https://tinyurl.com/ku9dqq8

[30] "GnuCash's bug report #471," 2019. [Online]. Available: https://tinyurl.com/y6luonwp

[31] "GnuCash's bug report #616," 2019. [Online]. Available: https://tinyurl.com/y5edsasv

[32] "GnuCash's bug report #620," 2019. [Online]. Available: https://tinyurl.com/y3pw69ac

[33] "GnuCash's bug report #701," 2019. [Online]. Available: https://tinyurl.com/y4e4ny9a

[34] "Google Chrome's feedback feature," 2019. [Online]. Available: https://support.google.com/chrome/answer/95315

[35] "Instabug," 2019. [Online]. Available: https://instabug.com

[36] "Launchpad's bug tracker," 2019. [Online]. Available: https://bugs.launchpad.net/

[37] "Microsoft Office's feedback feature," 2019. [Online]. Available: https://tinyurl.com/y6p2huna

[38] "Mileage," 2019. [Online]. Available: https://tinyurl.com/cw3uttu

[39] "Mileage's bug report #53," 2019. [Online]. Available: https://tinyurl.com/y6mo92cm

[40] “Qualtrics online survey system,” https://www.qualtrics.com/research-core/survey-software/, 2019.

[41] “Schedule,” 2019. [Online]. Available: https://tinyurl.com/bsw89ud

[42] “Schedule’s bug report #154,” 2019. [Online]. Available: https://tinyurl.com/y3pg92fr

[43] “Schedule’s bug report #169,” 2019. [Online]. Available: https://tinyurl.com/y46l44vr

[44] “StORMeD island parser,” 2019. [Online]. Available: https://stormed.inf.usi.ch/

[45] “TensorFlow’s issues,” 2019. [Online]. Available: https://github.com/tensorflow/tensorflow/issues

[46] “TestFairy,” 2019. [Online]. Available: https://testfairy.com

[47] “Tika’s bug report #1152,” 2019. [Online]. Available: https://issues.apache.org/jira/browse/TIKA-1152

[48] “A Time Tracker,” 2019. [Online]. Available: https://tinyurl.com/lt4ztgp

[49] “A Time Tracker’s bug report #1,” 2019. [Online]. Available: https://tinyurl.com/y4skjrp6

[50] “A Time Tracker’s bug report #10,” 2019. [Online]. Available: https://tinyurl.com/y4a698hb

[51] “A Time Tracker’s bug report #35,” 2019. [Online]. Available: https://tinyurl.com/y3tvylgs

[52] “Trac,” 2019. [Online]. Available: https://trac.edgewall.org/

[53] K. Aggarwal, F. Timbers, T. Rutgers, A. Hindle, E. Stroulia, and R. Greiner, “Detecting duplicate bug reports with software engineering domain knowledge,” *Journal of Software: Evolution and Process*, vol. 29, no. 3, 2017.

[54] K. Aijmer and A.-B. Stenström, *Discourse patterns in spoken and written corpora.* John Benjamins Publishing, 2004, vol. 120.

[55] N. Ali, A. Sabane, Y.-G. Gueheneuc, and G. Antoniol, “Improving Bug Location Using Binary Class Relationships,” in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM’12)*, 2012, pp. 174–183.

[56] J. Allan, J. Callan, K. Collins-Thompson, B. Croft, F. Feng, D. Fisher, J. Lafferty, L. Larkey, T. N. Truong, P. Ogilvie *et al.*, “The lemur toolkit for language modeling and information retrieval. The Lemur Project,” 2003.

[57] M. Amoui, N. Kaushik, A. Al-Dabbagh, L. Tahvildari, S. Li, and W. Liu, "Search-Based Duplicate Defect Detection: An Industrial Experience," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 173–182.

[58] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests," in *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, 2008, pp. 304–318.

[59] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse Technology eXchange (ETX'05)*, 2005, pp. 35–39.

[60] V. Arnaoudova, S. Haiduc, A. Marcus, and G. Antoniol, "The Use of Text Retrieval and Natural Language Processing in Software Engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 898–899.

[61] A. Bacchelli, A. Cleve, M. Lanza, and A. Mocci, "Extracting Structured Data from Natural Language Documents with Island Parsing," in *Proceedings of the International Conference on Automated Software Engineering (ASE'11)*, 2011, pp. 476–479.

[62] Y.-M. Baek and D.-H. Bae, "Automated Model-based Android GUI Testing Using Multi-level GUI Comparison Criteria," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 238–249.

[63] S. K. Bajracharya and C. V. Lopes, "Analyzing and Mining a Code Search Engine Usage Log," *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 424–466, 2012.

[64] C. F. Baker, C. J. Fillmore, and J. B. Lowe, "The Berkeley FrameNet Project," in *Proceedings of the Annual Meeting of the Association for Computational Linguistics and International Conference on Computational Linguistics*, 1998, pp. 86–90.

[65] B. Bassett and N. A. Kraft, "Structural information based term weighting in text retrieval for feature location," in *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, 2013, pp. 133–141.

[66] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What Makes a Good Bug Report?" in *Proceedings of the 16th International Symposium on Foundations of Software Engineering (FSE'08)*, 2008, pp. 308–318.

[67] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful ... really?" in *Proceedings of the International Conference on Software Maintenance (ICSM'08)*, 2008, pp. 337–345.

[68] J. Bhatia, T. D. Breaux, and F. Schaub, "Mining Privacy Goals from Privacy Policies Using Hybridized Task Recomposition," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, pp. 1–24, 2016.

[69] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent Dirichlet Allocation," *Journal of Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

[70] V. Boisselle and B. Adams, "The Impact of Cross-Distribution Bug Duplicates, Empirical Study on Debian and Ubuntu," in *Proceedings of the 15th International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 131–140.

[71] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching Word Vectors with Subword Information," *Transactions of the Association for Computational Linguistics*, vol. 5, pp. 135–146, 2017.

[72] M. Borg, P. Runeson, and A. Ardö, "Recovering from a decade: a systematic mapping of information retrieval approaches to software traceability," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1565–1616, 2014.

[73] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann, "Information Needs in Bug Reports: Improving Cooperation Between Developers and Users," in *Proceedings of the Conference on Computer Supported Cooperative Work (CSCW'10)*, 2010, pp. 301–310.

[74] J. Burstein, D. Marcu, S. Andreyev, and M. Chodorow, "Towards Automatic Classification of Discourse Elements in Essays," in *Proceedings of the Annual Meeting on Association for Computational Linguistics*, 2001, pp. 98–105.

[75] J. Burstein, D. Marcu, and K. Knight, "Finding the WRITE stuff: automatic identification of discourse structure in student essays," *IEEE Intelligent Systems*, vol. 18, no. 1, pp. 32–39, 2003.

[76] C. Carpineto and G. Romano, "A Survey of Automatic Query Expansion in Information Retrieval," *Computing Surveys*, vol. 44, no. 1, p. 1, 2012.

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

[78] Y. C. Cavalcanti, P. A. da Mota Silveira Neto, I. d. C. Machado, T. F. Vale, E. S. de Almeida, and S. R. d. L. Meira, "Challenges and Opportunities for Software Change Request Repositories: A Systematic Mapping Study," *Journal of Software: Evolution and Process*, vol. 26, no. 7, pp. 620–653, 2014.

[79] Y. C. Cavalcanti, P. A. d. M. S. Neto, D. Lucrédio, T. Vale, E. S. de Almeida, and S. R. de Lemos Meira, "The bug report duplication problem: an exploratory study," *Software Quality Journal*, vol. 21, no. 1, pp. 39–66, 2013.

[80] O. Chaparro, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the Quality of the Steps to Reproduce in Bug Reports," in *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19)*, 2019, (to appear).

[81] ——, "Replication package: Assessing the Quality of the Steps to Reproduce in Bug Reports," 2019. [Online]. Available: https://seers.utdallas.edu/projects/s2r-quality

[82] O. Chaparro, J. M. Florez, and A. Marcus, "On the Vocabulary Agreement in Software Issue Descriptions," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, 2016, pp. 448–452.

[83] ——, "Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 376–387.

[84] ——, "Replication package: Using Bug Descriptions to Reformulate Queries during Text-Retrieval-based Bug Localization," 2019. [Online]. Available: https://seers.utdallas.edu/projects/emse-query-reformulation/

[85] ——, "Using Bug Descriptions to Reformulate Queries during Text-Retrieval-based Bug Localization," *Empirical Software Engineering*, pp. 1–61, 2019, (online first).

[86] O. Chaparro, J. M. Florez, U. Singh, and A. Marcus, "Reformulating Queries for Duplicate Bug Report Detection," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER'2019)*, 2019, pp. 218–229.

[87] ——, "Replication package: Reformulating Queries for Duplicate Bug Report Detection," 2019. [Online]. Available: https://seers.utdallas.edu/projects/ob-query-reformulation-duplicates/

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

[89] ——, "Replication package: Detecting Missing Information In Bug Descriptions," 2017. [Online]. Available: https://seers.utdallas.edu/projects/missing-info-in-bugs/

[90] O. Chaparro and A. Marcus, "On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance," in *Proceedings of the 38th International Conference on Software Engineering (ICSE'16)*, 2016, pp. 716–718.

[91] P. K. Chilana, A. J. Ko, and J. O. Wobbrock, "Understanding Expressions of Unwanted Behaviors in Open Bug Reporting," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'10)*, 2010, pp. 203–206.

[92] N. Cliff, *Ordinal methods for behavioral data analysis.* Psychology Press, 2014.

[93] J. Cohen, "A Coefficient of Agreement for Nominal Scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[94] A. Conneau, G. Kruszewski, G. Lample, L. Barrault, and M. Baroni, “What you can cram into a single vector: Probing sentence embeddings for linguistic properties,” *CoRR*, vol. abs/1805.01070, 2018.

[95] K. Damevski, D. Shepherd, and L. Pollock, “A Field Study of How Developers Locate Features in Source Code,” *Empirical Software Engineering*, vol. 21, no. 2, pp. 724–747, 2016.

[96] T. Dao, L. Zhang, and N. Meng, “How does execution information help with information-retrieval based bug localization?” in *Proceedings of the International Conference on Program Comprehension (ICPC’17)*, 2017, pp. 241–250.

[97] J. L. Davidson, N. Mohan, and C. Jensen, “Coping with duplicate bug reports in free/open source software projects,” in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC’11)*, 2011, pp. 101–108.

[98] S. Davies and M. Roper, “What’s in a bug report?” in *Proceedings of the 8th International Symposium on Empirical Software Engineering and Measurement (ESEM’14)*, 2014, p. 26.

[99] S. Davies, M. Roper, and M. Wood, “Using bug report similarity to enhance bug localisation,” in *Proceedings of the Working Conference on Reverse Engineering (WCRE’12)*, 2012, pp. 125–134.

[100] A. De Lucia, A. Marcus, R. Oliveto, and D. Poshyvanyk, “Information Retrieval Methods for Automated Traceability Recovery,” in *Software and Systems Traceability*, J. Cleland-Huang, O. Gotel, and A. Zisman, Eds. Springer, 2012, pp. 71–98.

[101] A. Di Sorbo, S. Panichella, C. A. Visaggio, M. Di Penta, G. Canfora, and H. C. Gall, “Development Emails Content Analyzer: Intention Mining in Developer Discussions,” in *Proceedings of the International Conference on Automated Software Engineering (ASE’15)*, 2015, pp. 12–23.

[102] T. Dietrich, J. Cleland-Huang, and Y. Shin, “Learning effective query transformations for enhanced requirements trace retrieval,” in *Proceedings of the International Conference on Automated Software Engineering (ASE’13)*, 2013, pp. 586–591.

[103] T. Dilshener, M. Wermelinger, and Y. Yu, “Locating Bugs Without Looking Back,” in *Proceedings of the International Conference on Mining Software Repositories (MSR’16)*, 2016, pp. 286–290.

[104] B. Dit, D. Poshyvanyk, and A. Marcus, “Measuring the semantic similarity of comments in bug reports,” in *Proceedings of the 1st International Workshop on Semantic Technologies in System Maintenance (STSM’08)*, 2008, pp. 265–280.

[105] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature Location in Source code: A Taxonomy and Survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2012.

[106] G. Dougherty, *Pattern Recognition and Classification: An Introduction.* Springer Science & Business Media, 2012.

[107] B. P. Eddy, N. A. Kraft, and J. Gray, "Impact of structural weighting on a latent Dirichlet allocation–based feature location technique," *Journal of Software: Evolution and Process*, vol. 30, no. 1, p. e1892, 2018.

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

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

[110] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in IR-based concept location," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 351–360.

[111] X. Ge, D. C. Shepherd, K. Damevski, and E. Murphy-Hill, "Design and evaluation of a multi-recommendation system for local code search," *Journal of Visual Languages & Computing*, 2016.

[112] M. Gibiec, A. Czauderna, and J. Cleland-Huang, "Towards Mining Replacement Queries for Hard-to-retrieve Traces," in *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 245–254.

[113] R. J. Grissom and J. J. Kim, *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum Associates Publishers, 2005.

[114] J. Guo, M. Gibiec, and J. Cleland-Huang, "Tackling the term-mismatch problem in automated trace retrieval," *Empirical Software Engineering*, pp. 1–40, 2016.

[115] P. J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy, "Characterizing and predicting which bugs get fixed: an empirical study of Microsoft Windows," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, 2010, pp. 495–504.

[116] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic Query Reformulations for Text Retrieval in Software Engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 842–851.

[117] E. Hatcher and O. Gospodnetic, *Lucene in Action.* Manning Publications, 2004.

[118] E. Hill, M. Roldan-Vega, J. A. Fails, and G. Mallet, "NL-based query refinement and contextualized code search results: A user study," in *Proceedings of the Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, 2014, pp. 34–43.

[119] A. Hindle, A. Alipour, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection and ranking," *Empirical Software Engineering*, vol. 21, no. 2, pp. 368–410, 2016.

[120] A. Hindle and C. Onuczko, "Preventing duplicate bug reports by continuously querying bug reports," *Empirical Software Engineering*, vol. 24, no. 2, pp. 902–936, 2019.

[121] T. V. Hoang, R. J. Oentaryo, T. B. Le, and D. Lo, "Network-Clustered Multi-Modal Bug Localization," *IEEE Transactions on Software Engineering*, 2018, (to appear).

[122] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods.* John Wiley & Sons, 2013, vol. 751.

[123] P. Hooimeijer and W. Weimer, "Modeling Bug Report Quality," in *Proceedings of the 22nd International Conference on Automated Software Engineering (ASE'07)*, 2007, pp. 34–43.

[124] Z. Huang, W. Xu, and K. Yu, "Bidirectional LSTM-CRF Models for Sequence Tagging," *CoRR*, vol. abs/1508.01991, 2015.

[125] T. Joachims, "Making large-Scale SVM Learning Practical," Universität Dortmund, LS VIII-Report, LS8-Report 24, 1998.

[126] ——, "Text categorization with Support Vector Machines: Learning with many relevant features," in *Proceedings of the European Conference on Machine Learning*, 1998, pp. 137–142.

[127] ——, "Training Linear SVMs in Linear Time," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 2006, pp. 217–226.

[128] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*, 2014, pp. 437–440.

[129] S. Just, R. Premraj, and T. Zimmermann, "Towards the next generation of bug tracking systems," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing*, 2008, pp. 82–85.

[130] L. Kang, "Automated Duplicate Bug Reports Detection - An Experiment at Axis Communication AB," Master's thesis, 2017.

[131] G. Karagöz and H. Sözer, "Reproducing failures based on semiformal failure scenario descriptions," *Software Quality Journal*, vol. 25, no. 1, pp. 111–129, 2017.

[132] K. Kevic and T. Fritz, "Automatic Search Term Identification for Change Tasks," in *Proceedings of the International Conference on Software Engineering (ICSE'14)*, 2014, pp. 468–471.

[133] N. Klein, C. S. Corley, and N. A. Kraft, "New Features for Duplicate Bug Detection," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'14)*. ACM, 2014, pp. 324–327.

[134] A. J. Ko and P. K. Chilana, "How Power Users Help and Hinder Open Bug Reporting," in *Proceedings of the Conference on Human Factors in Computing Systems (CHI'10)*, 2010, pp. 1665–1674.

[135] ——, "Design, Discussion, and Dissent in Open Bug Reports," in *Proceedings of the iConference (iConference'11)*, 2011, pp. 106–113.

[136] A. J. Ko, B. A. Myers, and D. H. Chau, "A Linguistic Analysis of How People Describe Software Problems," in *Proceedings of the Symposium on Visual Languages and Human-Centric Computing (VL/HCC'06)*, 2006, pp. 127–134.

[137] K. Krippendorff, *Content analysis: An introduction to its methodology.* Sage, 2004.

[138] G. Lample, M. Ballesteros, S. Subramanian, K. Kawakami, and C. Dyer, "Neural Architectures for Named Entity Recognition," in *Proceedings of North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT'16)*, 2016, pp. 260–270.

[139] E. I. Laukkanen and M. V. Mäntylä, "Survey Reproduction of Defect Reporting in Industrial Software Development," in *Proceedings of the International Symposium on Empirical Software Engineering and Measurement*, 2011, pp. 197–206.

[140] A. Lazar, S. Ritchey, and B. Sharif, "Generating Duplicate Bug Datasets," in *Proceedings of the 11th Working Conference on Mining Software Repositories (MSR'11)*, 2014, pp. 392–395.

[141] T.-D. B. Le, F. Thung, and D. Lo, "Predicting effectiveness of ir-based bug localization techniques," in *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE'14)*, 2014, pp. 335–345.

[142] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4BL: Reproducibility Study on the Performance of IR-based Bug Localization," in *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA'18)*, 2018, pp. 61–72.

[143] O. A. L. Lemos, A. Carvalho de Paula, H. Sajnani, and C. V. Lopes, “Can the Use of Types and Query Expansion Help Improve Large-scale Code Search?” in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM’15)*, 2015, pp. 41–50.

[144] J. Lerch and M. Mezini, “Finding Duplicates of Your Yet Unwritten Bug Report,” in *Proceedings of the 17th European Conference on Software Maintenance and Reengineering (CSMR’13)*, 2013, pp. 69–78.

[145] B. Li, X. Sun, H. Leung, and S. Zhang, “A survey of code-based change impact analysis techniques,” *Software Testing, Verification and Reliability*, vol. 23, no. 8, pp. 613–646, 2013.

[146] Z. Li, T. Wang, Y. Zhang, Y. Zhan, and G. Yin, “Query Reformulation by Leveraging Crowd Wisdom for Scenario-based Software Search,” in *Proceedings of the Asia-Pacific Symposium on Internetware (Internetware’16)*, 2016, pp. 36–44.

[147] M.-J. Lin, C.-Z. Yang, C.-Y. Lee, and C.-C. Chen, “Enhancements for Duplication Detection in Bug Reports with Manifold Correlation Features,” *Journal of Systems and Software*, vol. 121, pp. 223–233, 2016.

[148] E. Linstead and P. Baldi, “Mining the coherence of GNOME bug reports with statistical topic models,” in *Proceedings of the 6th International Working Conference on Mining Software Repositories (MSR’09)*, 2009, pp. 99–102.

[149] K. Liu, H. B. K. Tan, and H. Zhang, “Has This Bug Been Reported?” in *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE’13)*, 2013, pp. 82–91.

[150] R. Lotufo, Z. Malik, and K. Czarnecki, “Modelling the ’hurried’ bug report reading process to summarize bug reports,” *Empirical Software Engineering*, vol. 20, no. 2, pp. 516–548, 2014.

[151] X. A. Lu and R. B. Keefer, “Query expansion/reduction and its impact on retrieval effectiveness,” *NIST Special Publication*, pp. 231–231, 1995.

[152] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, “CodeHow: Effective Code Search Based on API Understanding and Extended Boolean Model (E),” in *Proceedings of the International Conference on Automated Software Engineering (ASE’15)*, 2015, pp. 260–270.

[153] X. Ma and E. Hovy, “End-to-end Sequence Labeling via Bi-directional LSTM-CNNs-CRF,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, ser. ACL’16, vol. 1, 2016, pp. 1064–1074.

[154] W. Maalej and M. P. Robillard, “Patterns of Knowledge in API Reference Documentation,” *IEEE Transactions on Software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[155] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "AUSUM: Approach for Unsupervised Bug Report Summarization," in *Proceedings of the International Symposium on the Foundations of Software Engineering (FSE'12)*, 2012, pp. 11:1–11:11.

[156] C. D. Manning, M. Surdeanu, J. Bauer, J. R. Finkel, S. Bethard, and D. McClosky, "The Stanford CoreNLP Natural Language Processing Toolkit," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics*, 2014, pp. 55–60.

[157] A. Marcus and S. Haiduc, "Text Retrieval Approaches for Concept Location in Source Code," in *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7171, pp. 126–158.

[158] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An Information Retrieval Approach to Concept Location in Source Code," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'04)*, 2004, pp. 214–223.

[159] W. Martin, F. Sarro, Y. Jia, Y. Zhang, and M. Harman, "A Survey of App Store Analysis for Software Engineering," *IEEE Transactions on Software Engineering*, vol. 43, no. 9, pp. 817–847, 2017.

[160] M. B. Miles, A. M. Huberman, and J. Saldaña, *Qualitative Data Analysis: A Methods Sourcebook*, 3rd ed. SAGE Publications, Inc, Apr 2013.

[161] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. De Lucia, "Predicting Query Quality for Applications of Text Retrieval to Software Engineering Tasks," *Transactions on Software Engineering and Methodology*, vol. 26, no. 1, p. 3, 2017.

[162] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are Bug Reports Enough for Text Retrieval-based Bug Localization?" in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 410–421.

[163] K. Moran, M. Linares-Váquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically Discovering, Reporting and Reproducing Android Application Crashes," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'16)*, 2016, pp. 33–44.

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

[165] K. Moran, M. Linares-Vasquez, C. Bernal-Cardenas, C. Vendome, and D. Poshyvanyk, "CrashScope: A Practical Tool for Automated Testing of Android Applications," in *Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE'17)*, 2017, pp. 15–18.

[166] L. Moreno, W. Bandara, S. Haiduc, and A. Marcus, "On the Relationship between the Vocabulary of Bug Reports and Source Code," in *Proceedings of the International Conference on Software Maintenance (ICSM'13)*, 2013, pp. 452–455.

[167] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the Use of Stack Traces to Improve Text Retrieval-Based Bug Localization," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 151–160.

[168] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proceedings of the 27th International Conference on Automated Software Engineering (ASE'12)*, 2012, pp. 70–79.

[169] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query Expansion Based on Crowd Knowledge for Code Search," *IEEE Transactions on Services Computing*, vol. 9, no. 5, pp. 771–783, 2016.

[170] A. N. Oppenheim, *Questionnaire Design, Interviewing and Attitude Measurement.* Pinter Publishers, 1992.

[171] S. Panichella, A. Di Sorbo, E. Guzman, C. A. Visaggio, G. Canfora, and H. C. Gall, "ARdoc: App Reviews Development Oriented Classifier," in *Proceedings of the International Symposium on Foundations of Software Engineering (FSE'16)*, 2016, pp. 1023–1027.

[172] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering Information Explaining API Types Using Text Classification," in *Proceedings of the International Conference on Software Engineering (ICSE'15)*, 2015, pp. 869–879.

[173] L. Polanyi, *The handbook of discourse analysis.* Wiley-Blackwell, 2003, vol. 18, ch. The Linguistic Structure of Discourse, p. 265.

[174] L. Ponzanelli, A. Mocci, and M. Lanza, "StORMeD: Stack Overflow ready made data," in *Proceedings of the 12th Working Conference on Mining Software Repositories (MSR'15)*, 2015, pp. 474–477.

[175] M. F. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.

[176] D. Poshyvanyk, A. Marcus, R. Ferenc, and T. Gyimóthy, "Using information retrieval based coupling measures for impact analysis," *Empirical Software Engineering*, vol. 14, no. 1, pp. 5–32, 2009.

[177] T. Prifti, S. Banerjee, and B. Cukic, "Detecting Bug Duplicate Reports Through Local References," in *Proceedings of the 7th International Conference on Predictive Models in Software Engineering (PROMISE'11)*, 2011, pp. 8:1–8:9.

[178] M. M. Rahman, J. Barson, S. Paul, J. Kayani, F. A. Lois, S. F. Quezada, C. Parnin, K. T. Stolee, and B. Ray, “Evaluating How Developers Use General-purpose Web-search for Code Retrieval,” in *Proceedings of the 15th International Conference on Mining Software Repositories (MSR'18)*, 2018, pp. 465–475.

[179] M. M. Rahman and C. K. Roy, “QUICKAR: Automatic query reformulation for concept location using crowdsourced knowledge,” in *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 220–225.

[180] ——, “Improved query reformulation for concept location using CodeRank and document structures,” in *Proceedings of the International Conference on Automated Software Engineering (ASE'17)*, 2017, pp. 428–439.

[181] ——, “STRICT: Information Retrieval Based Search Term Identification for Concept Location,” in *Proceeding of the Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, 2017, pp. 79–90.

[182] ——, “Improving IR-Based Bug Localization with Context-Aware Query Reformulation,” in *Proceedings of the 26th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'18)*, 2018, pp. 621–632.

[183] M. S. Rakha, C.-P. Bezemer, and A. E. Hassan, “Revisiting the Performance Evaluation of Automated Approaches for the Retrieval of Duplicate Issue Reports,” *IEEE Transactions on Software Engineering*, vol. 44, no. 12, pp. 1245–1268, 2018.

[184] ——, “Revisiting the performance of automated approaches for the retrieval of duplicate reports in issue tracking systems that perform just-in-time duplicate retrieval,” *Empirical Software Engineering*, vol. 23, no. 5, pp. 2597–2621, 2018.

[185] L. A. Ramshaw and M. P. Marcus, “Text chunking using transformation-based learning,” in *Natural language processing using very large corpora.* Springer, 1999, pp. 157–176.

[186] S. Rastkar, G. C. Murphy, and G. Murray, “Automatic Summarization of Bug Reports,” *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 366–380, 2014.

[187] M. Rath, D. Lo, and P. Mäder, “Analyzing Requirements and Traceability Information to Improve Bug Localization,” in *Proceedings of the Working Conference on Mining Software Repositories (MSR'18).* ACM, 2018.

[188] P. C. Rigby and M. P. Robillard, “Discovering Essential Code Elements in Informal Documentation,” in *Proceedings of the International Conference on Software Engineering (ICSE'12)*, 2013, pp. 832–841.

[189] H. Rocha, M. T. Valente, H. Maques-Neto, and G. Murphy, “An Empirical Study on Recommendations of Similar Bugs,” in *Proceedings of the 23rd International Conference on Software Analysis, Evolution and Reengineering (SANER'16)*, 2016, pp. 1–10.

[190] P. Rodeghero, D. Huo, T. Ding, C. McMillan, and M. Gethers, “An empirical study on how expert knowledge affects bug reports,” *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 542–564, 2016.

[191] M. Roldan-Vega, G. Mallet, E. Hill, and J. A. Fails, “CONQUER: A Tool for NL-Based Query Refinement and Contextualizing Code Search Results.” in *Proceedings of the International Conference on Software Maintenance (ICSM’13)*, 2013, pp. 512–515.

[192] J. Romano, J. D. Kromrey, J. Coraggio, and J. Skowronek, “Appropriate statistics for ordinal level data: Should we really be using t-test and Cohen’s d for evaluating group differences on the NSSE and other surveys,” in *Annual meeting of the Florida Association of Institutional Research*, 2006, pp. 1–33.

[193] P. Runeson, M. Alexandersson, and O. Nyholm, “Detection of duplicate defect reports using natural language processing,” in *Proceedings of the 29th International Conference on Software Engineering (ICSE’07)*, 2007, pp. 499–510.

[194] M. Sadat, A. B. Bener, and A. V. Miranskyy, “Rediscovery Datasets: Connecting Duplicate Reports,” in *Proceedings of the 14th International Conference on Mining Software Repositories (MSR’17)*, 2017, pp. 527–530.

[195] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” in *Proceedings of the International Conference on Automated Software Engineering (ASE’13)*, 2013, pp. 345–355.

[196] S. K. Sahoo, J. Criswell, and V. Adve, “An empirical study of reported bugs in server software with implications for automated bug diagnosis,” in *Proceedings of the International Conference on Software Engineering (ICSE’10)*, 2010, pp. 485–494.

[197] G. Salton, A. Wong, and C. S. Yang, “A vector space model for automatic indexing,” *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[198] T. D. Sasso, A. Mocci, and M. Lanza, “What Makes a Satisficing Bug Report?” in *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS’16)*, 2016, pp. 164–174.

[199] C. B. Seaman, “Qualitative Methods in Empirical Studies of Software Engineering,” *IEEE Transactions on Software Engineering*, vol. 25, no. 4, pp. 557–572, 1999.

[200] D. Shepherd, Z. P. Fry, E. Hill, L. Pollock, and K. Vijay-Shanker, “Using Natural Language Program Analysis to Locate and Understand Action-oriented Concerns,” in *Proceedings of the International Conference on Aspect-oriented Software Development (AOSD’07)*, 2007, pp. 212–224.

[201] Z. Shi, J. Keung, K. E. Bennin, and X. Zhang, “Comparing Learning to Rank Techniques in Hybrid Bug Localization,” *Applied Soft Computing*, vol. 62, pp. 636–648, 2018.

[202] S. E. Sim, M. Umarji, S. Ratanotayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Transactions on Software Engineering and Methodology*, vol. 21, no. 1, p. 4, 2011.

[203] B. Sisman, S. A. Akbar, and A. C. Kak, "Exploiting Spatial Code Proximity and Order for Improved Source Code Retrieval for Bug Localization," *Journal of Software: Evolution and Process*, vol. 29, no. 1, p. e1805, 2016.

[204] B. Sisman and A. C. Kak, "Incorporating version histories in Information Retrieval based bug localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, 2012, pp. 50–59.

[205] ——, "Assisting Code Search with Automatic Query Reformulation for Bug Localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 309–318.

[206] R. Slavin, X. Wang, M. B. Hosseini, J. Hester, R. Krishnan, J. Bhatia, T. D. Breaux, and J. Niu, "Toward a Framework for Detecting Privacy Policy Violations in Android Application Code," in *Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 25–36.

[207] D. Spencer, *Card sorting: Designing usable categories.* Rosenfeld Media, 2009.

[208] J. Starke, C. Luce, and J. Sillito, "Searching and skimming: An exploratory study," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 157–166.

[209] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proceedings of the 26th International Conference on Automated Software Engineering (ASE'11)*, 2011, pp. 253–262.

[210] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proceedings of the 32nd International Conference on Software Engineering (ICSE'10)*, 2010, pp. 45–54.

[211] A. Sureka and P. Jalote, "Detecting Duplicate Bug Report Using Character N-Gram-Based Features," in *Proceedings of the Asia Pacific Software Engineering Conference (APSEC'10)*, 2010, pp. 366–374.

[212] A. Takahashi, N. Sae-Lim, S. Hayashi, and M. Saeki, "A Preliminary Study on Using Code Smells to Improve Bug Localization," in *Proceedings of the International Conference on Program Comprehension (ICPC'18).* ACM, 2018, p. 4.

[213] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical Software Engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[214] F. Thung, P. S. Kochhar, and D. Lo, "DupFinder: Integrated Tool Support for Duplicate Bug Report Detection," in *Proceedings of the 29th International Conference on Automated Software Engineering (ASE'14)*, 2014, pp. 871–874.

[215] F. Thung, D. Lo, and L. Jiang, "Automatic Defect Categorization," in *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*, 2012, pp. 205–214.

[216] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proceedings of the 16th European Conference on Software Maintenance and Reengineering (CSMR'12)*, 2012, pp. 385–390.

[217] C. Treude, F. Figueira Filho, and U. Kulesza, "Summarizing and Measuring Development Activity," in *Proceedings of the 10th Joint Meeting on Foundations of Software Engineering (FSE'15)*, 2015, pp. 625–636.

[218] A. J. Viera, J. M. Garrett *et al.*, "Understanding interobserver agreement: the kappa statistic," *Family medicine*, vol. 37, no. 5, pp. 360–363, 2005.

[219] J. Wang, M. Li, S. Wang, T. Menzies, and Q. Wang, "Images don't lie: Duplicate crowdtesting reports detection with screenshot information," *Information and Software Technology*, vol. 110, pp. 139 – 155, 2019.

[220] S. Wang and D. Lo, "Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization," in *Proceedings of the 22nd International Conference on Program Comprehension (ICPC'14)*, 2014, pp. 53–63.

[221] ——, "AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, pp. 921–942, 2016.

[222] S. Wang, D. Lo, and L. Jiang, "Active Code Search: Incorporating User Feedback to Improve Code Search Relevance," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE'14)*, 2014, pp. 677–682.

[223] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proceedings of the 30th international conference on Software engineering (ICSE'08)*, 2008, pp. 461–470.

[224] M. Wen, R. Wu, and S. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 262–273.

[225] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 181–190.

[226] Y. Xiao, J. Keung, K. E. Bennin, and Q. Mi, "Improving bug localization with word embedding and enhanced convolutional neural networks," *Information and Software Technology*, vol. 105, pp. 17 – 29, 2019.

[227] B. Xu, D. Lo, X. Xia, A. Sureka, and S. Li, "EFSPredictor: Predicting Configuration Bugs with Ensemble Feature Selection," in *Proceedings of the Asia-Pacific Software Engineering Conference (ASPEC'15)*, 2015, pp. 206–213.

[228] J. Yang, S. Liang, and Y. Zhang, "Design Challenges and Misconceptions in Neural Sequence Labeling," in *Proceedings of the 27th International Conference on Computational Linguistics (COLING'18)*, 2018, pp. 3879–3889.

[229] J. Yang and Y. Zhang, "NCRF++: An Open-source Neural Sequence Labeling Toolkit," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (ACL'18)*, 2018.

[230] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to Extract API Mentions from Informal Natural Language Discussions," in *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, 2016, pp. 389–399.

[231] X. Ye, R. Bunescu, and C. Liu, "Mapping Bug Reports to Relevant Files: A Ranking Model, a Fine-Grained Benchmark, and Feature Evaluation," *IEEE Transactions on Software Engineering*, vol. 42, no. 4, pp. 379–402, 2016.

[232] X. Ye, F. Fang, J. Wu, R. Bunescu, and C. Liu, "Bug Report Classification Using LSTM Architecture for More Accurate Software Defect Locating," in *Proceedings of the 17th IEEE International Conference on Machine Learning and Applications (ICMLA'18)*, 2018, pp. 1438–1445.

[233] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[234] R. N. Zaeem, M. R. Prasad, and S. Khurshid, "Automated Generation of Oracles for Testing User-Interaction Features of Mobile Apps," in *Proceedings of the 7th International Conference on Software Testing, Verification and Validation (ICST'14)*, 2014, pp. 183–192.

[235] M. S. Zanetti, I. Scholtes, C. J. Tessone, and F. Schweitzer, "Categorizing Bugs with Social Networks: A Case Study on Four Open Source Software Communities," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 1032–1041.

[236] H. Zaragoza, N. Craswell, M. J. Taylor, S. Saria, and S. E. Robertson, "Microsoft Cambridge at TREC 13: Web and Hard Tracks." in *TREC*, vol. 4, 2004, pp. 1–1.

[237] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*, 2nd ed. Morgan Kaufmann Publishers Inc., 2009.

[238] T. Zhang, J. Chen, H. Jiang, X. Luo, and X. Xia, "Bug Report Enrichment with Application of Automated Fixer Recommendation," in *Proceedings of the 25th International Conference on Program Comprehension (ICPC'17).* IEEE Press, 2017, pp. 230–240.

[239] T. Zhang, H. Jiang, X. Luo, and A. T. S. Chan, "A Literature Review of Research in Bug Resolution: Tasks, Challenges and Future Directions," *The Computer Journal*, vol. 59, no. 5, pp. 741–773, 2016.

[240] W. Zhang, Z. Li, Q. Wang, and J. Li, "FineLocator: A novel approach to method-level fine-grained bug localization by query expansion," *Information and Software Technology*, vol. 110, pp. 121 – 135, 2019.

[241] Y. Zhang, D. Lo, X. Xia, T. D. B. Le, G. Scanniello, and J. Sun, "Inferring Links between Concerns and Methods with Multi-abstraction Vector Space Model," pp. 110–121, 2016.

[242] Y. Zhao, T. Yu, T. Su, Y. Liu, W. Zheng, J. Zhang, and W. G. Halfond, "ReCDroid: Automatically Reproducing Android Application Crashes from Bug Reports," in *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering (ICSE'19)*, 2019, pp. 128–139.

[243] B. Zhou, X. Xia, D. Lo, C. Tian, and X. Wang, "Towards More Accurate Content Categorization of API Discussions," in *Proceedings of the International Conference on Program Comprehension (ICPC'14)*, 2014, pp. 95–105.

[244] J. Zhou and H. Zhang, "Learning to rank duplicate bug reports," in *Proceedings of the 21st International Conference on Information and Knowledge Management (CIKM'12)*, 2012, pp. 852–861.

[245] J. Zhou, H. Zhang, and D. Lo, "Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports," in *Proceedings of the International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[246] Y. Zhou, Y. Tong, T. Chen, and J. Han, "Augmenting bug localization with part-of-speech and invocation," *International Journal of Software Engineering and Knowledge Engineering*, vol. 27, no. 06, pp. 925–949, 2017.

[247] Y. Zhou, Y. Tong, R. Gu, and H. Gall, "Combining text mining and data mining for bug report classification," *Journal of Software: Evolution and Process*, vol. 28, no. 3, pp. 150–176, 2016.

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

[249] T. Zimmermann, R. Premraj, N. Bettenburg, S. Just, A. Schröter, and C. Weiss, "What Makes a Good Bug Report?" *IEEE Transactions on Software Engineering*, vol. 36, no. 5, pp. 618–643, 2010.

[250] T. Zimmermann, R. Premraj, J. Sillito, and S. Breu, "Improving bug tracking systems," in *Proceedings of the International Conference on Software Engineering (ICSE'09)*, 2009, pp. 247–250.

BIOGRAPHICAL SKETCH

Oscar Chaparro is a PhD candidate in Software Engineering at The University of Texas at Dallas, advised by Dr. Andrian Marcus. He received his BEng and MEng degrees in Systems Engineering and Computing from Universidad Nacional de Colombia, in Bogotá, Colombia.

His research interests are in Software Engineering, with emphasis on Software Maintenance and Evolution. His research focuses on developing techniques, tools, methodologies, and practices that help software developers efficiently understand and change large software systems, so that they produce high-quality software. His dissertation work aims at improving the quality of bug reports written by end users and assisting software developers during bug triage and resolution. Oscar's dissertation research focuses on identifying, verifying, and analyzing the textual information of bug reports, by adapting and integrating techniques from software analysis, natural language processing, and machine learning.

Oscar has authored several publications in top software engineering venues, such as the ACM/SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE), the IEEE/ACM International Conference on Software Engineering (ICSE), the Empirical Software Engineering (EMSE) journal, and the IEEE International Conference on Software Maintenance and Evolution (ICSME). He obtained the IEEE TCSE Distinguished Paper Award at ICSME'17 and the ACM SIGSOFT Distinguished Paper Award at ESEC/FSE'19. He served on the organizing and program committee of the 3rd International Workshop on Dynamic Software Documentation (DySDoc3) in 2018. Oscar has four years of industry experience in software research and development.

CURRICULUM VITAE

Oscar Chaparro

June 11, 2019

Contact Information

Department of Computer Science
The University of Texas at Dallas
800 W. Campbell Rd., ECSS 4.620
Richardson, TX 75080, USA

Email: `ojchaparroa@utdallas.edu`
Website: `https://ojcchar.github.io`

Education

- **PhD in Software Engineering**, 2019
  University of Texas at Dallas, Richardson, TX, USA
  Dissertation: Automated Analysis of Bug Descriptions to Support Bug Reporting and Resolution
  Advisor: Dr. Andrian Marcus
- **MEng in Systems Engineering and Computing**, 2012
  Universidad Nacional de Colombia, Bogotá, Colombia
  Thesis: Semiautomatic Reverse Engineering Tool for Oracle Forms Information Systems
- **BEng in Systems Engineering**, 2010
  Universidad Nacional de Colombia, Bogotá, Colombia
  Thesis: Software Visualization in Maintenance Processes: A General Overview

Honors and Awards

- ACM SIGSOFT Distinguished Paper Award, at the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19), 2019
- Dissertation Research Award ($5,000), The University of Texas at Dallas, 2018
- IEEE TCSE Distinguished Paper Award, at the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17), 2017
- Thomas C. Rumble graduate fellowship ($20,000), Wayne State University, 2013
- NSF Travel Grants for ICSE'16/17, ICSME'16, and SWAN'17
- Master's Thesis Laureate Distinction, Universidad Nacional de Colombia, 2012
- Honor Enrollment (MEng), Universidad Nacional de Colombia, 2011
- Honor Enrollment (BEng), Universidad Nacional de Colombia, 2007, 2010

Peer-Reviewed Publications

1. **O. Chaparro**, C. Bernal-Cárdenas, J. Lu, K. Moran, A. Marcus, M. Di Penta, D. Poshyvanyk, and V. Ng, "Assessing the Quality of the Steps to Reproduce in Bug Reports," in *Proceedings of the 27th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'19)*, 2019, (to appear). **ACM SIGSOFT Distinguished Paper Award**
2. **O. Chaparro**, J. M. Florez, U. Singh, and A. Marcus, "Reformulating Queries for Duplicate Bug Report Detection," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'19)*, 2019, pp. 218-229
3. **O. Chaparro**, J. M. Florez, and A. Marcus, "Using Bug Descriptions to Reformulate Queries during Text-Retrieval-based Bug Localization," in *Empirical Software Engineering*, Springer, 2019, pp. 1-61 (online first).
4. **O. Chaparro**, J. M. Florez, and A. Marcus, "Using Observed Behavior to Reformulate Queries during Text Retrieval-based Bug Localization," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 376-387. **IEEE TCSE Distinguished Paper Award**
5. **O. Chaparro**, J. Lu, F. Zampetti, L. Moreno, M. Di Penta, A. Marcus, G. Bavota, and V. Ng, "Detecting Missing Information in Bug Descriptions," in *Proceedings of the 11th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'17)*, 2017, pp. 396-407.
6. **O. Chaparro**, "Improving Bug Reporting, Duplicate Detection, and Localization," in *Proceedings of the 39th ACM/IEEE International Conference on Software Engineering (ICSE'17) - Doctoral Symposium*, 2017, pp. 421-424.
7. M. P. Robillard, A. Marcus, C. Treude, G. Bavota, **O. Chaparro**, N. Ernst, M. A. Gerosa, M. Godfrey, M. Lanza, M. Linares-Vásquez, G. C. Murphy, L. Moreno, D. Shepherd, and E. Wong, "On-demand Developer Documentation," in *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME'17) - NIER Track*, 2017, pp. 479-483.
8. **O. Chaparro** and A. Marcus, "On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance," in *Proceedings of the 38th ACM/IEEE International Conference on Software Engineering (ICSE'16) - NIER Track*, 2016, pp. 716-718.
9. **O. Chaparro**, J. M. Florez, and A. Marcus, "On the Vocabulary Agreement in Software Issue Descriptions," in *Proceedings of the 32nd IEEE International Conference on Software Maintenance and Evolution (ICSME'16) - ERA Track*, 2016, pp. 448-452.
10. **O. Chaparro**, G. Bavota, A. Marcus, and M. D. Penta, "On the Impact of Refactoring Operations on Code Quality Metrics," in *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution (ICSME'14) - ERA Track*, 2014, pp. 456-460.

11. **O. Chaparro**, J. Aponte, F. Ortega, and A. Marcus, “Towards the Automatic Extraction of Structural Business Rules from Legacy Databases,” in *Proceedings of the 19th IEEE Working Conference on Reverse Engineering (WCRE’12)*, 2012, pp. 479-488.

Research and Professional Experience

- Research/Teaching Assistant, Aug. 2014 - Present
  The University of Texas at Dallas, Richardson, TX, USA
- Research Assistant, May - July 2014
  Wayne State University, Detroit, MI, USA
- Software Developer and R&D Leader, Oct. 2011 - July 2013
  ITC Soluciones Tecnológicas SAS, Bogotá, Colombia
- Software Developer, Aug. 2009 - Oct. 2011
  IT Osmosys LTDA, Bogotá, Colombia

Professional Memberships

- IEEE/IEEE Computer Society (Student Member since 2012)
- ACM/ACM Special Interest Group on Software (Student Member since 2016)