Using Bug Descriptions to Reformulate Queries during Text-Retrieval-based Bug Localization

Oscar Chaparro ·
Juan Manuel Florez ·
Andrian Marcus

Received: date / Accepted: date

Abstract Text Retrieval (TR)-based approaches for bug localization rely on formulating an initial query based on the full text of a bug report. When the query fails to retrieve the buggy code artifacts, developers can reformulate the query and retrieve more candidate code documents. Existing research on query reformulation focuses mostly on leveraging relevance feedback from the user or on expanding the original query with additional information. We hypothesize that the title of the bug reports, the observed behavior, expected behavior, steps to reproduce, and code snippets provided by the users in bug descriptions, contain the most relevant information for retrieving the buggy code artifacts, and that other parts of the descriptions contain more irrelevant terms, which hinder retrieval. This paper proposes and evaluates a set of query reformulation strategies based on the selection of existing information in bug descriptions, and the removal of irrelevant parts from the original query. The results show that selecting the bug report title and the observed behavior is the strategy that performs best across various TR-based bug localization approaches and 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 consistent across different thresholds N. Selecting the steps to reproduce or the expected behavior (when provided in the bug reports) along with the bug title and the observed behavior leads to higher performance (*i.e.*, between 31.4% and 41.7% more queries) and comparable consistency, yet it is applicable in fewer cases. These reformulation strategies are easy to use and are independent of the underlying retrieval technique.

O. Chaparro, J. M. Florez, and A. Marcus
Department of Computer Science
The University of Texas at Dallas
Richardson, TX, USA
E-mail: {ojchaparroa, jflorez, amarcus}@utdallas.edu

Keywords Bug Descriptions · Query Reformulation · Bug Localization · Text Retrieval

1 Introduction

Text Retrieval (TR) has been widely used by researchers to support developers during bug localization in source code (Saha et al., 2013; Wang and Lo, 2014; Wang et al., 2014a; Le et al., 2015; Nguyen et al., 2011; Nichols, 2010; Rao and Kak, 2011; Sisman and Kak, 2012; Wang and Lo, 2016; Wong et al., 2014; Ye et al., 2016b,a; Youm et al., 2017; Zhou et al., 2012; Wen et al., 2016; Zhang et al., 2016; Le et al., 2014; Eddy et al., 2018)[1]. 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 bug. In the general TRBL process (Marcus and Haiduc, 2013; Dit et al., 2012; De Lucia et al., 2012), 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 (Marcus et al., 2004). 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 artifacts 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 (Wang and Lo, 2014; Saha et al., 2013; Wang and Lo, 2016; Youm et al., 2017; Ali et al., 2012; Takahashi et al., 2018), execution traces from the bug report (Moreno et al., 2014; Wong et al., 2014; Wang and Lo, 2016; Youm et al., 2017; Sisman et al., 2016), past bug reports and code changes (Zhou et al., 2012; Wang and Lo, 2014, 2016; Youm et al., 2017; Saha et al., 2013; Davies et al., 2012; Wong et al., 2014; Rath et al., 2018; Sisman and Kak, 2012), etc. Other research has

[1] See Section 6 for details

focused on techniques to reformulate the initial query, mostly by leveraging relevance feedback from the user (Gay et al., 2009), pseudo-relevance feedback based on previous search results (Haiduc et al., 2013), or additional information to replace or expand the query (*e.g.*, adding synonyms) (Shepherd et al., 2007; Rahman and Roy, 2016; Marcus et al., 2004). 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 (Chaparro and Marcus, 2016; Mills et al., 2018) 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 (Rahman and Roy, 2016, 2017a, 2018; Haiduc et al., 2013; Chaparro and Marcus, 2016; Chaparro et al., 2017a).

This paper proposes and investigates 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 S2R), and the expected software behavior (*i.e.*, EB) (Zimmermann et al., 2010; Davies and Roper, 2014; Chaparro et al., 2017a,b). Both, the title and description, *i.e.*, the textual account of the problem written by the reporter, is what we call *bug description.* We hypothesize that the TITLE of the bug reports, the OB, EB, and S2R, 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.

This paper introduces and empirically evaluates 31 different reformulation strategies, which reduce the initial query to parts that correspond to the TITLE, OB, EB, S2R, 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 (Hatcher and Gospodnetic, 2004), Lobster (Moreno et al., 2014), BugLocator (Zhou et al., 2012), BRTracer (Wong et al., 2014), and Locus (Wen et al., 2016), which retrieve code artifacts at different code granularities (*i.e.*, file, class, and method). Using existing TRBL datasets (Zhou et al., 2012; Wong et al., 2014; Moreno et al., 2014; Mills et al., 2017; Chaparro et al., 2017a; Lee et al., 2018), 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. The results indicate that the combination of TITLE+OB is the strategy that performs best across the five TRBL approaches, as it returns 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. Combining the TITLE and the OB with the S2R (when present in the bug reports) leads to higher 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 higher performance (*i.e.*, 41.7% more queries with respect to no reformulation) and comparable consistency, yet its low applicability makes it less practical. The results support our hypothesis, which means that developers can reformulate an initial query by simply selecting the part that describes the TITLE, OB, and S2R/EB (when present), and expect better retrieval results.

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 (Dit et al., 2012). 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 S2R/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.

This paper is a substantial extension of our previous research on using OB to reformulate queries for TRBL (Chaparro et al., 2017a). We extend our prior work in four major ways:

– We investigate how specific types of information from bug descriptions (*i.e.*, EB, S2R, TITLE, and CODE) can be used to reformulate the initial query in TRBL application. One of the main findings of this research is that using OB only (as we reported in the previous work) is not the best query reformulation strategy in this application.
– Our empirical study investigates the effect of query reformulation on TRBL by using nearly three times more queries than in our prior work, which were collected from a total of 30 open source software projects (*i.e.*, nine

more projects than in our prior work). The number of queries and projects strengthens the external validity of our conclusions.

– In addition to the four TRBL techniques used in our prior research, the evaluation presented in this paper includes one additional state-of-the-art TRBL technique, called Locus (Wen et al., 2016), which further strengthens the external validity of our research.
– The evaluation of the strategies focuses on HITS@N, a measure of the proportion of queries that return at least one buggy code document within the top-N results. We contend that this metric, compared to traditional metrics such as MRR or MAP, approximates better an actual reformulation usage scenario given a TRBL tool. The user is likely to inspect only the top-N results to find the relevant code documents, before switching strategies. The ranks of the relevant documents, outside top-N, is less relevant to the user. In other words, we consider a reformulation strategy to be effective when it improves HITS@N, rather than MRR or MAP.

We provide an online replication package (Chaparro et al., 2019) 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 the paper.

The rest of this paper is structured as follows. Section 2 explains in detail the proposed reformulation strategies. Section 3 describes the design of the empirical study for the evaluation of the reformulation strategies, and Section 4 presents and discusses the empirical results. Section 5 discusses the threats to validity we identified, followed by the related work in Section 6. Finally, we present our conclusions in Section 7.

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 (Wang and Lo, 2014; Saha et al., 2013; Wang and Lo, 2016; Youm et al., 2017; Ali et al., 2012) 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 (Chaparro et al., 2017a) 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 to the OB, we hypothesize that it contains relevant terms with respect to code retrieval.
- Steps to reproduce (*a.k.a.* **S2R**): it describes the steps that the user followed to trigger the OB. The S2R 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.

Figure 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, S2R, 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, S2R, 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 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*".

Fig. 1: Bug report #89621 from Eclipse. The highlighted text corresponds to the title (TITLE), code snippets (CODE), observed behavior (OB), expected behavior (EB), and steps to reproduce (S2R).

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. [S2R]

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 S2R

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

3 Empirical Evaluation

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 4 discusses the evaluation results. We used five TRBL techniques (Sections 3.1 and 3.2) to locate the buggy code artifacts for a large set of queries/bug reports (Section 3.3). Then, for a subset of the queries for which the tools failed to retrieve the buggy code documents within the top-N results (Section 3.4), we manually identified the structure of the corresponding bug descriptions (Section 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 3.6 and 3.7).

3.1 TRBL Techniques

We used five TRBL techniques to perform our empirical evaluation on both initial and reformulated queries, namely Lucene (Hatcher and Gospodnetic, 2004), Lobster (Moreno et al., 2014), BugLocator (Zhou et al., 2012), BR-Tracer (Wong et al., 2014), and Locus (Wen et al., 2016). We were interested in finding out whether the reformulation strategies are equally effective on different TBRL techniques.

Lucene (Hatcher and Gospodnetic, 2004) is a retrieval technique implemented in the open source library of the same name (Lucene, 2017). Lucene combines the standard information retrieval Boolean model and the Vector Space Model (VSM), based on the TF-IDF representation (Salton et al., 1975), 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 (Moreno et al., 2014) 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 (Zhou et al., 2012) 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 (Wong et al., 2014) 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 (Wen et al., 2016) 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, *i.e.*, small code segments, product of 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.

3.2 Implementation of the TRBL Techniques

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 (Moreno et al., 2014).

Regarding Locus, we used the implementation provided in Bench4BL (Lee et al., 2018). 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 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. Our replication package (Chaparro et al.,

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

2019) contains the fix to Locus' issue, as well as the code we used to run our experiments.

We used our own implementation of BugLocator (Zhou et al., 2012) and BRTracer (Wong et al., 2014), based on the description provided in their corresponding publications. We obtained the implementations of BugLocator and BRTracer made available by their authors, along with the experimental data they used (Zhou et al., 2012; Wong et al., 2014), 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 (Wong et al., 2014). 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 was not provided by the authors and the corresponding system version is no longer available online. However, 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 (Zhou et al., 2012). 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 detailed in the 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 1: Performance differences between our implementation (Our impl.) of BugLocator and the original implementation (Original impl.) provided by its authors (Zhou et al., 2012).

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%

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, an examination of the implementation revealed that 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 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 (Wong et al., 2014). 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.

Table 2: Performance differences between our implementation (Our impl.) of BRTracer and the original implementation (Original impl.) provided by its authors (Wong et al., 2014).

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%

2. The calculation of the SimiScore (the score boost of source 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:

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

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 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 (Wong et al., 2014; Wen et al., 2016).

3.3 TRBL Data

We compiled three TRBL data sets from existing TRBL data (see Table 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 (Moreno et al., 2014) from 16 versions of 13 open source

Table 3: Statistics of the TRBL data sets used in the evaluation.

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.

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 (Wong et al., 2014) and on data from the Bench4BL TRBL benchmark (Lee et al., 2018). 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.
– The method-level data set (*i.e.*, **MDS**) is based on Mills *et al.*'s data on query quality assessment (Mills et al., 2017), and on Chaparro *et al.*'s adaptation (Chaparro et al., 2017a) of Just *et al.*'s Defects4J data (Just et al., 2014) 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 (Chaparro et al., 2017a). 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 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, *i.e.*, 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 both Eclipse and SWT. These queries are originally duplicated in Wong *et al.*'s data (Wong et al., 2014). The duplication stems from the fact that SWT is a subproject of Eclipse. In any case, the code corpus for both

[3] This data set is called BRT in our prior work (Chaparro et al., 2017a).

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). Our replication package includes the full list of projects, versions, and queries, including the duplicated ones (Chaparro et al., 2019).

We were not able to use all the queries from the Bench4BL data (Lee et al., 2018) 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 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). See our replication package for the list of discarded queries as well as the reformatted query set (Chaparro et al., 2019).

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). The full list of projects for each data set and their TRBL data is included in our online replication package (Chaparro et al., 2019).

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 Java files (*i.e.*, 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 (Porter, 1980). 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 (Wen et al., 2016) for more details. The full set of stop words, bug reports, queries, gold sets, and preprocessing code are available in our replication package (Chaparro et al., 2019).

3.4 *Low-quality* Queries

As mentioned in Section 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] (Rahman et al., 2018; Sim et al., 2011), 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.*, 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 (Mills et al., 2017; Lee et al., 2018; Marcus and Haiduc, 2013; Dit et al., 2012; Wang and Lo, 2014; Zhou et al., 2012; Moreno et al., 2014; Wong et al., 2014; Chaparro et al., 2017a). 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, in this paper, 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

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

Table 4: Number (and proportion) of queries for which the TRBL techniques fail to retrieve the buggy code documents within the top-5 results.

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 3.

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 (Lee et al., 2018) and in Locus' original evaluation (Wen et al., 2016), 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 (Lee et al., 2018).

Table 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 5).

Table 5: Number (and proportion) of queries for which the TRBL techniques fail to retrieve the buggy code documents within the top-N results.

Data set	**Top-5**	**Top-10**	**Top-15**
CDS	307 (37.7%)	231 (28.3%)	202 (24.8%)
FDS	2,925 (66.0%)	2,427 (54.8%)	2,115 (47.8%)
MDS	199 (55.3%)	175 (48.6%)	157 (43.6%)
Total	**3,431 (61.2%)**	**2,833 (50.6%)**	**2,474 (44.1%)**

Data set	**Top-20**	**Top-25**	**Top-30**
CDS	176 (21.6%)	161 (19.8%)	139 (17.1%)
FDS	1,926 (43.5%)	1,773 (40.0%)	1,653 (37.3%)
MDS	147 (40.8%)	131 (36.4%)	124 (34.4%)
Total	**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.

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 (**S2R**) 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.

3.5.1 Bug Report Sampling

As shown in Table 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).

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). 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. 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

Table 6: Number (and proportion) of sampled queries for which the TRBL techniques fail to retrieve the buggy code documents within the top-N results.

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 7: Number (and proportion) of sampled queries for which the TRBL techniques fail to retrieve the buggy code documents within the top-5 results.

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).

done for any threshold N. Table 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 5 is lower than the total number of *low-quality* queries shown in Table 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 3.3, a small subset 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.

3.5.2 Identification of OB, EB, and S2R

The first two authors of this paper and one master student conducted the identification of OB, EB, and S2R in the 1,221 sampled bug reports. The reports 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 (Seaman, 1999) on the bug reports using the coding framework and criteria defined in our prior work (Chaparro et al., 2017b,a). The coders' job was to tag the sentences in the title and description of the reports that corresponded to OB, EB, and S2R. This task was performed using a web-based text annotation tool called BRAT[5], in combination with one of our own tools that splits the text into sentences and paragraphs, based on the Stanford CoreNLP toolkit (Manning et al., 2014) and heuristics (*e.g.*, using punctuation).

We summarize the most important criteria used by the coders to tag the OB, EB, and S2R in the bug descriptions. The full list can be found in our replication package (Chaparro et al., 2019).

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 S2R 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 (Chaparro et al., 2017a).

[5] `http://brat.nlplab.org/`

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 S2R may contain OB or EB sentences. In such cases, the OB/EB/S2R 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 S2R and was allowed for coding. An example of this case is: "*When I execute the script attached, I get the following error: ...*".
- Some S2R may be labeled with phrases such as "to reproduce:" or "steps to reproduce". The label per se cannot be considered S2R, only the natural language content that such phrases are labeling must be coded as S2R. An example of these cases is shown in Figure 2.
- Imperative and conditional sentences are often used to describe S2R (Chaparro et al., 2017b). 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 S2R, thus should not be tagged. In contrast, the sentence *"When I share a URL in my Facebook wall page, Facebook will not retrieve..."* gives a specific and clear description of the S2R, 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 (Chaparro et al., 2017b,a). 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.

3.5.3 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[6] provided by Ponzanelli *et al.* (Ponzanelli et al., 2015), 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.

[6] `https://stormed.inf.usi.ch/`

Fig. 2: Bug report #101434 from Eclipse. The highlighted text corresponds to the title (TITLE), observed behavior (OB), expected behavior (EB), and steps to reproduce (S2R).

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 [S2R]

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 S2R

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

3.5.4 Structure Identification Results

Overall, 1,185 (*i.e.*, 97.1%) of the tagged bug reports describe an OB (see Table 8), and only 284 (23.2%), 625 (51.2%), and 481 (39.4%) of the bug reports contain sentences corresponding to EB, S2R, and CODE, respectively. These proportions are in line with the ones measured in other bug reports data sets (Davies and Roper, 2014; Chaparro et al., 2017b,a). 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/S2R/TITLE/CODE sentences from a single report, which takes seconds. For example, from the bug report shown in Figure 2, the user would only select the highlighted text and use it for reformulation, depending on the reformulation strategy. Note that the OB, EB, S2R, 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 8: Number (and proportion) of sampled queries that contain TITLE, OB, EB, S2R, and CODE.

Data set	TITLE	OB	EB	S2R	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%)**

3.6 Evaluation Procedure and Measures

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 2). We reformulate the *low-quality initial queries* by retaining the sentences tagged as OB, EB, S2R, 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 S2R), 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, S2R, 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 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 (Wang and Lo, 2014; Zhou et al., 2012; Moreno et al., 2014; Wong et al., 2014) and it is ideal for assessing the performance of TRBL 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

Table 9: Number (and proportion) of reduced queries (for N=5) generated by each reformulation strategy and executed by each TRBL technique.

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**

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 could be 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) (Dit et al., 2012).

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:

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

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:

$$\text{MAP}(Q) = \frac{1}{|Q|}\sum_{q\in Q}\frac{1}{|R_q|}\sum_{r\in R_q}\text{precision}(\text{rank}(r)) \tag{2}$$

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):

$$\text{Improv}(M) = \frac{M_a - M_b}{M_b} \tag{3}$$

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 (Hollander et al., 2013), 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 $< 5\%$.

3.6.1 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 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 in this paper 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 (Chaparro et al., 2019). 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[7]. 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).

[7] See Table 11 and our replication package for more details.

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 *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 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 subcategory.

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

Table 10: Overall number (and proportion) of reduced queries generated by each reformulation strategy, and the applicability of each 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.

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*).

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*

Table 11: Average number (and proportion) of queries for which **Lucene** retrieves at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

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.
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$< 5\%$).

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.

Table 12: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* when using **Lucene**.

Applicability	Effectiveness: VE	SE	SI	VI
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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

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 4.1), Lobster (Section 4.2), BugLocator (Section 4.3), BRTracer (Section 4.4), and Locus (Section 4.5). In addition, we analyze the results for each code granularity (Section 4.6) and for all TRBL techniques on aggregate (Section 4.7). We provide examples and discuss the best and worst reformulation strategies (Section 4.8), including the trade-offs between successful and unsuccessful queries (Section 4.9).

4.1 Performance for Lucene

Tables 11 and 12 show the results obtained for Lucene across the 26 thresholds N and three code granularities (or data sets). The replication package (Chaparro et al., 2019) contains the breakdown for each data set and each threshold N (N=5, 6, 7, ..., 30). The way to interpret the results in Table 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 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 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 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 12, and the way to interpret the table is as follows. For example, let us keep looking at the OB[8] reformulation strategy. As OB can be used to reformulate 97.1% of the queries (see Table 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 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 11 and 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, *i.e.*, 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 (Chaparro et al., 2019). Table 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 11). The other two *highly-applicable* strategies, namely 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 is the most-effective (3rd most-effective overall), as it improves TRBL for 29.2% more queries (on average) compared to no reformulation (*i.e.*, 45.9%/51.2% average MRR/MAP improvement). As for the other *moderately-applicable* strate-

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

gies, 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 N, 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 (Chaparro et al., 2019).

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, *i.e.*, more than 40% HITS@N/MAP/MRR deterioration (see Tables 11 and 12, and our online replication package). Also, these strategies consistently lead to deterioration for 25 thresholds N (see Table 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.

4.2 Performance for Lobster

The results obtained for Lobster reveal that 25 (out of 31) reformulation strategies are *effective* (see Tables 13 and 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 the last two rows of Table 13[9]). 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, *i.e.*, they both achieve 222.2% avg. HITS@N improvement, and 1,167.6%/839.8% avg. MRR/MAP improvement[10]. However, the applicability of these strategies is rather *low*, as they can reformulate $\sim$3 initial queries only. All *highly-applicable* are *very-effective* (see Table 14), and consistently improve HITS@N across all 26 thresholds N, with respect to no reformulation (see Table 13). Among these, TITLE is the strategy with the highest effectiveness (*i.e.*, 97.1% avg. HITS@N

[9] 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 3).

[10] See our replication package for the detailed MRR/MAP results (Chaparro et al., 2019).

Table 13: Average number (and proportion) of queries for which **Lobster** retrieves at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

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.
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%).

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

Table 14: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* 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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

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 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.

4.3 Performance for BugLocator

Tables 15 and 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, none of these improvements are statistically significant (Mann-Whitney, 5% significance level) and these strategies present low consistency

Table 15: Average number (and proportion) of queries for which **BugLocator** retrieves at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

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.
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).

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

Table 16: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* when using **BugLocator**.

		Effectiveness			
		VE	**SE**	**SI**	**VI**
Applicability	**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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

using no reformulation for each one of the 26 thresholds (see Table 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.

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.

4.4 Performance for BRTracer

The results for BRTracer (see Tables 17 and 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 18), as only ∼55 queries (out of ∼474[11]) 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%), 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

[11] The # of queries for TITLE in Table 17 represents the avg. total # of queries for BRTracer.

Table 17: Average number (and proportion) of queries for which **BRTracer** retrieves at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

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.
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$< 5\%$).

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. Remem-

Table 18: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* 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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

ber that BRTracer is an extension of BugLocator, consequently, BRTracer's robustness comes from BugLocator.

4.5 Performance for Locus

Tables 19 and 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 less number of queries than when using no reformulation (*i.e.*, they are *ineffective*). Eight of them are *very-ineffective* because

Table 19: Average number (and proportion) of queries for which **Locus** retrieves at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**).

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.
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$< 5\%$).

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 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

Table 20: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* when using **Locus**.

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

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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

the buggy code documents within the top-N results for more cases across all 26 thresholds N (compared to no reformulation).

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 21 and 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-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 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

Table 21: Average number of reduced queries (|Q|) and HITS@N improvement (Improv) for **Lucene** on each data set when using each reformulation strategy.

Reformulation	**CDS**		**FDS**		**MDS**		**All data sets**	
strategy	**|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%

Average values across thresholds N={5, 6, 7, ..., 30}.
Strategies sorted by avg. HITS@N improvement for all data sets.

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 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 de-

Table 22: 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.

Reformulation	**CDS**				**FDS**				**MDS**			
Strategy	**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%

H@N is the average HITS@N improvement across 26 thresholds N.
Strategies sorted by average HITS@N improvement across all data sets.

terioration (*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 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 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 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 (Just et al., 2014), *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 (Just et al., 2014): (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 S2R 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.

4.7 Overall Reformulation 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 23, 24, and 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 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 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% 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

Table 23: Average number (and proportion) of queries for which **all five TRBL techniques** retrieve at least one buggy code document within the top-N results using each one of the reformulation strategies (Reform) vs. no reformulation (No reform); and aggregated number of thresholds for which each strategy is Effective (**E**), Neutral (**N**), and Ineffective (**I**), across all TRBL techniques.

Reformulation	**# of**	**HITS@N**			**Thresh.**		
strategy	**queries**	**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. Strategies sorted by average HITS@N improvement (Improv).
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$< 5\%$).

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.

Table 24: Mean Reciprocal Rank (MRR) and Mean Average Precision (MAP) achieved by **all five TRBL techniques** using each one of the reformulation strategies (Ref) vs. no reformulation (No ref).

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.
Strategies sorted by avg. HITS@N improvement (same order as in Table 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$< 5\%$).

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 S2R 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, S2R, and CODE), the TITLE, OB, and S2R 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

Table 25: Categorization of each reformulation strategy according to their *Effectiveness* and *Applicability* when using **all five TRBL techniques**.

Applicability	VE	SE	SI	VI
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**), S2R (**S**), TITLE (**T**), and CODE (**C**).

describing OB and S2R (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) than without reformulation.

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 S2R 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 3). The title describes a problem related to “enhanced for statements” in “scrapbook pages”. The S2R and the CODE snippet provide the information for triggering the problem. Note that they do not state the problem, but

Fig. 3: Bug report #102778 from Eclipse and its corresponding buggy file. 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 [S2R].

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. [S2R]
Add a new scrapbook page that $\boxed{\text{contains}}$ this $\boxed{\text{source}}$: [S2R]

```
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`

```
[...]
/**
* A parser for code snippets.
*/
public class CodeSnippetParser extends Parser
                        implements EvaluationConstants {
    [...]
    int last Statement  = -1; // end of last top level  statement
    [...]
    /**
    *  Creates  a new code snippet parser.
    */
    public CodeSnippetParser(ProblemReporter problemReporter,
                            EvaluationContext evaluationContext,
                            [...] ) {
        [...]
        this.reportOnlyOne SyntaxError  = true;
        this.javadocParser.checkDocComment = false;
    }
    [...]
    protected CompilationUnitDeclaration endParse(int act) {
        [...]
        // otherwise it  contained  the type, [...]
        int start = unitResult.problems[i].get Source Start();
        [...]
    }
    [...]
    protected void report SyntaxErrors (boolean isDietParse,
                                                int oldFirstToken) {
        [...]
        super.report SyntaxErrors (isDietParse, oldFirstToken);
    }
    [...]
}
```

Bug report found at https://bugs.eclipse.org/bugs/show_bug.cgi?id=102778 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, *i.e.*, creating a java project that contains some source code is a common task in Eclipse. In this case, S2R 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 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*[12]. Assume the developer inspects the top-5 documents, then reformulates and executes the query, and inspects the next top-5 documents[13]. Using TITLE, OB, and S2R **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 S2R 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 S2R, when combined with OB or TITLE, deteriorates the rank of the buggy file.

Consider the example in Figure 4, regarding bug report #4330 from ArgoUML and its respective buggy class `TabToDo`. In this case, the OB states that there is an "exception" 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 S2R. The S2R 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,

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

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

Fig. 4: Bug report #4330 from ArgoUML and its corresponding buggy class. 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 [S2R].

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 $\boxed{\text{ToDoPane}}$ [S2R]
- press the ”send $\boxed{\text{email}}$ to $\boxed{\text{expert}}$” button [S2R]

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.argouml.ui.cmd.ActionEmailExpert.action
                                        Performed(ActionEmailExpert.java:57)
at javax.swing.AbstractButton.fireActionPerformed(Unknown Source)
at javax.swing.AbstractButton$Handler.actionPerformed(Unknown Source)
at javax.swing.DefaultButtonModel.fireActionPerformed(Unknown Source)
at javax.swing.DefaultButtonModel.setPressed(Unknown Source)
at javax.swing.plaf.basic.BasicButtonListener.mouseReleased(Unknown Source)
at java.awt.AWTEventMulticaster.mouseReleased(Unknown Source)
at java.awt.AWTEventMulticaster.mouseReleased(Unknown Source)

[...]
```

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` appear frequently in the buggy class, hence, they are highly relevant. When the S2R is 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 S2R contain specific terms about the problem and are useful for TRBL.

Although we consider combining OB and TITLE (as well as S2R, 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 OB, TITLE or S2R, 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 OB, TITLE, and/or S2R. 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 OB and EB are.

It is important to note that S2R 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 3), the S2R includes 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 S2R 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 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, S2R, 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 TRBL performance of the reformulation strategies.

Table 26: 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.

(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 values across the 3 data sets and 26 thresholds N.

4.9 Trade-offs between Successful and 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, *i.e.*, OB+TITLE. We also analyze the case when OB+TITLE is combined with the S2R, *i.e.*, 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 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 are the approaches with the highest proportions of *unsuccessful* → *successful* queries, which are substantially higher than the proportions for BugLocator and BRTracer. These results further show the robustness of BugLocator and BRTracer with respect to noisy queries.

Figure 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[14] leads to retrieving the buggy method in the 13th position, *i.e.*, 6 positions down the result list. Figure 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 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[15], 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 S2R (*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

[14] 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)."
[15] Found at `https://issues.apache.org/jira/browse/TIKA-1152`

Fig. 5: Bug report #727 from Math and its corresponding buggy method. 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].

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` future work, we will investigate ways to incorporate (parts of the) stack traces into the reformulation.

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 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 measuring 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 OB, EB, and S2R, 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 (Chaparro et al., 2017b,a). In order to reduce subjectivity, we used the set of common coding criteria that we defined in our prior work (Chaparro et al., 2017b,a). 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 (Wang and Lo, 2014; Zhou et al., 2012; Moreno et al., 2014; Wong et al., 2014). 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 easy 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 (Chaparro et al., 2017a; Haiduc et al., 2013).

The *internal validity* of our evaluation is affected by our TRBL data sets and approaches. Based on data previously used in TRBL studies (Zhou et al., 2012; Wong et al., 2014; Moreno et al., 2014; Mills et al., 2017; Chaparro et al., 2017a; Lee et al., 2018), 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 (Chaparro et al., 2017a) to strengthen the generalizability of our conclusions. Finally, we used five TRBL techniques, namely Lucene (Hatcher and Gospodnetic, 2004), Lobster (Moreno et al., 2014), BugLocator (Zhou et al., 2012), BRTracer (Wong et al., 2014), and Locus (Wen et al., 2016). Investigating the effectiveness of our reformulation strategies with other TRBL techniques is part of our future research agenda.

6 Related Work

In this section, we describe the main TRBL approaches and discuss existing work on query reformulation in the context of source code retrieval.

6.1 TR-based Bug Localization

TRBL is closely related to TR-based concept/feature location in source code (Marcus and Haiduc, 2013; Dit et al., 2012) and TR-based traceability link recovery (De Lucia et al., 2012). These are all formulated as document retrieval problems. A requirement (*e.g.*, feature description, bug report, *etc.*) 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 implement the requirement. The targeted (*i.e.*, relevant) code documents are the ones that contain the feature described in the requirement. Both TRBL and concept/feature location can be considered instances of traceability link recovery, but they differ in the types of artifacts they use as queries. What differentiates TR-based bug localization from the more general code retrieval approaches is the use of bug reports as queries.

TRBL techniques often use additional information related to the current bug report to adjust the ranking of the relevant code documents. Additional information leveraged by existing TRBL techniques includes: code structure (Wang and Lo, 2014; Saha et al., 2013; Wang and Lo, 2016; Youm et al., 2017; Ali et al., 2012; Takahashi et al., 2018), part-of-speech tags (Zhou et al., 2017), similar bug reports (Zhou et al., 2012; Wang and Lo, 2014, 2016; Youm et al., 2017; Saha et al., 2013; Davies et al., 2012; Wong et al., 2014; Rath et al., 2018), code version history (Sisman and Kak, 2012; Wang and Lo, 2016; Youm et al., 2017; Wang and Lo, 2014), stack traces (Moreno et al., 2014; Wong et al., 2014; Wang and Lo, 2016; Youm et al., 2017; Sisman et al., 2016; Wen et al., 2016), or combinations of the above (Wang and Lo, 2016; Youm et al., 2017; Wang and Lo, 2014; Saha et al., 2013; Wong et al., 2014; Shi et al., 2018; Dao et al., 2017).

We focus the discussion in this section on approaches designed specifically for bug retrieval (*i.e.*, they use information from or related to bug reports), rather than more generic concept/feature location and traceability link recovery approaches, which could also be used for bug localization. All TRBL approaches follow a common process, consisting of:

1. Building a corpus using the source code of the software.
2. Indexing the corpus using a TR model.
3. Formulating an initial query based on the bug report.
4. Ranking the documents with respect to the query, based on the TR model used and additional information related to the bug report.
5. Inspecting the retrieved documents. If the buggy code document is found, the process ends.
6. Reformulating the query if the buggy code is not identified, and resuming the process at step 4.

Our contributions focus on steps 5 and 6. For step 5, we assume that the user will examine the first N results before deciding that a relevant code artifact was not retrieved by the query formulated in step 3 (*i.e.*, the initial query). Our work provides a set of strategies for the reformulation of the initial query (step 6), which is commonly formulated using the full textual description of a bug report (Dit et al., 2012). We argue that the user should select certain parts of the bug report if they are present (TITLE, OB, and S2R/EB) and then re-run the newly created query with their TRBL approach of choice (step 4). In order to determine whether this approach is effective for TRBL, we compare the first N results produced by the reformulated query and the ones produced by the original query, excluding the previous N results already inspected and deemed irrelevant by the developer. In other words, we compare the results after reformulation with the case in which the user checks the following N results of the original query without applying reformulation.

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 (Zhang et al., 2016; Ye et al., 2016a,b; Le et al., 2014; Eddy et al., 2018; Hoang et al., 2018; Xiao et al., 2018), most of the research has focused on leveraging additional information related to the bug reports, as mentioned above.

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, *e.g.*, the number of times a code artifact has been fixed (Sisman and Kak, 2012; Wang and Lo, 2016; Wen et al., 2016; Youm et al., 2017) or how long ago this happened (Sisman and Kak, 2012; Wang and Lo, 2014; Wen et al., 2016).

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, *e.g.*, the textual similarity of the fixed bug with the query (Zhou et al., 2012; Wang and Lo, 2014, 2016; Youm et al., 2017; Saha et al., 2013; Davies et al., 2012; Wong et al., 2014). Recently, feature requests have been leveraged in addition to bugs, in the same way described before (Rath et al., 2018).

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 (Moreno et al., 2014; Wong et al., 2014; Wang and Lo, 2016; Youm et al., 2017; Sisman et al., 2016). 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 (Moreno et al., 2014) or the files' import statements (Wong et al., 2014; Youm et al., 2017).

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 (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 of the document components (8 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 (Wang and Lo, 2014; Saha et al., 2013; Wang and Lo, 2016; Youm et al., 2017; Ali et al., 2012). These approaches treat the title as a separate part of the bug report (similar to our TITLE strategy), however, they still use the full text of the bug report. We argue that some parts of the bug report text can be removed to improve TRBL performance.

It has also been proposed to use code smells as a separate source of information (Takahashi et al., 2018). 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.*. (Zhou et al., 2017) propose boosting the retrieval weight of bug report terms tagged as nouns by an automatic part-of-speech tagger.

Despite this rich body of existing work in improving TRBL, our focus is on helping developers reformulating a query when it fails to retrieve at least one relevant code artifact (*i.e.*, step 6 in the above process).

6.2 Query Reformulation in TRBL and Code Retrieval

Existing research highlighted the challenges that developers face when reformulating queries for code retrieval (Starke et al., 2009; Bajracharya and Lopes, 2012; Damevski et al., 2016). On one hand, TRBL approaches mitigate the problems associated with formulating an initial query by utilizing the bug report (Chaparro and Marcus, 2016). On the other hand, existing research provides little or no guidance on what parts of the bug reports to use when reformulating a query when no relevant results are retrieved within the first few entries of the ranked list (Kevic and Fritz, 2014).

Three general query reformulation strategies are found in the literature, namely, query expansion (Carpineto and Romano, 2012), query replacement (Gibiec et al., 2010; Guo et al., 2016), and query reduction (Lu and Keefer, 1995; Rahman and Roy, 2017a; Kevic and Fritz, 2014). 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 (Gay et al., 2009); pseudo-relevance feedback (Haiduc et al., 2013; Sisman and Kak, 2013), which leverages the lexicon of the previous top code documents retrieved; the use of English or software ontologies (*e.g.*, WordNet or custom-built models) (Shepherd et al., 2007; Rahman and Roy, 2017b), which contain related terms to the ones in a query (*e.g.*, synonyms); or co-occurring term information from various software sources, such as source code, Stack Overflow (SO) questions, or regulatory documents (Rahman and Roy, 2016; Marcus et al., 2004; Dietrich et al., 2013). Similar techniques have been applied in the context of code search, where the initial queries are reformulated based on thesauri (*e.g.*, lexical databases from SO) (Li et al., 2016; Lazzarini Lemos et al., 2015; Ge et al., 2017), relevance feedback from users (Wang et al., 2014b), pseudo-relevance feedback from SO results (Nie et al., 2016), co-occurrence and frequency of query terms with previous results and source code (Hill et al., 2014; Roldan-Vega et al., 2013), and textual similarity between the query and Application Programming Interfaces (Lv et al., 2015).

Query replacement has been utilized mostly for traceability link recovery (Gibiec et al., 2010), 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 (Guo et al., 2016).

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 (Chaparro and Marcus, 2016). Similarly, Mills *et al.* (Mills et al., 2018) 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.* (Rahman and Roy, 2016) 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.* (Haiduc et al., 2013) followed a similar strategy. Kevic *et al.* (Kevic and Fritz, 2014) 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 (Kevic and Fritz, 2014). In another work, Rahman *et al.* (Rahman and Roy, 2017a) 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 and whether they reference code entities and/or appear in particular parts of the bug reports, *e.g.*, in stack traces (Rahman and Roy, 2018). Related to term selection, other research focused on weighing terms (from the query) that occur in method names and calls (Bassett and Kraft, 2013) or terms corresponding to source code file names (Dilshener et al., 2016).

Our reformulation strategy is in line with query reduction, as we are selecting part of the initial query and discarding the rest of the query terms. We do not argue that all the terms we select are most relevant, but rather we claim that the terms we do not select are less relevant. To the best of our knowledge, our original work (Chaparro et al., 2017a) was the first to investigate how the type of textual content from bug descriptions used as queries can improve TR-based bug localization via query reformulation (specifically, query reduction), and it remains the only work in the subject until the time of writing this paper.

7 Conclusions and Future Work

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 (S2R), 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 S2R, 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 S2R (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 S2R 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.

References

Nasir Ali, Aminata Sabane, Yann-Gael Gueheneuc, and Giuliano Antoniol. Improving bug location using binary class relationships. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'12)*, pages 174–183, 2012.

Sushil Krishna Bajracharya and Cristina Videira Lopes. Analyzing and mining a code search engine usage log. *Empirical Software Engineering*, 17(4-5): 424–466, 2012.

B. Richard Bassett and Nicholas A. Kraft. Structural information based term weighting in text retrieval for feature location. In *Proceedings of the International Conference on Program Comprehension (ICPC'13)*, pages 133–141, 2013.

Claudio Carpineto and Giovanni Romano. A survey of automatic query expansion in information retrieval. *Computing Surveys*, 44(1):1, 2012.

Oscar Chaparro and Andrian Marcus. On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*, pages 716–718, 2016.

Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME'17)*, pages 376–387, 2017a.

Oscar Chaparro, Jing Lu, Fiorella Zampetti, Laura Moreno, Massimiliano Di Penta, Andrian Marcus, Gabriele Bavota, and Vincent Ng. Detecting missing information in bug descriptions. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'17)*, 2017b. 396-407.

Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Replication package, 2019. URL `https://tinyurl.com/y7bzqnwc`.

Kostadin Damevski, David Shepherd, and Lori Pollock. A field study of how developers locate features in source code. *Empirical Software Engineering*, 21(2):724–747, 2016.

Tung Dao, Lingming Zhang, and Na Meng. How does execution information help with information-retrieval based bug localization? In *Proceedings of the International Conference on Program Comprehension (ICPC'17)*, pages 241–250, 2017.

Steven Davies and Marc Roper. What's in a Bug Report? In *Proceedings of the International Symposium on Empirical Software Engineering and Measurement (ESEM'14)*, pages 26:1–26:10, 2014.

Steven Davies, Marc Roper, and Murray Wood. Using bug report similarity to enhance bug localisation. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'12)*, pages 125–134, 2012.

Andrea De Lucia, Andrian Marcus, Rocco Oliveto, and Denys Poshyvanyk. Information retrieval methods for automated traceability recovery. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 71–98. Springer, 2012.

Timothy Dietrich, Jane Cleland-Huang, and Yonghee Shin. Learning effective query transformations for enhanced requirements trace retrieval. In *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*, pages 586–591, 2013.

Tezcan Dilshener, Michel Wermelinger, and Yijun Yu. Locating bugs without looking back. In *Proceedings of the International Conference on Mining Software Repositories (MSR'16)*, pages 286–290, 2016.

Bogdan Dit, Meghan Revelle, Malcom Gethers, and Denys Poshyvanyk. Feature location in source code: A taxonomy and survey. *Journal of Software: Evolution and Process*, 25(1):53–95, 2012.

Brian P Eddy, Nicholas A Kraft, and Jeff Gray. Impact of structural weighting on a latent dirichlet allocation–based feature location technique. *Journal of Software: Evolution and Process*, 30(1):e1892, 2018.

Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 351–360, 2009.

Xi Ge, David C Shepherd, Kostadin Damevski, and Emerson Murphy-Hill. Design and evaluation of a multi-recommendation system for local code search. *Journal of Visual Languages & Computing*, 39:1–9, 2017.

Marek Gibiec, Adam Czauderna, and Jane Cleland-Huang. Towards mining replacement queries for hard-to-retrieve traces. In *Proceedings of the International Conference on Automated Software Engineering (ASE'10)*, pages 245–254, 2010.

Jin Guo, Marek Gibiec, and Jane Cleland-Huang. Tackling the term-mismatch problem in automated trace retrieval. *Empirical Software Engineering*, pages 1–40, 2016.

Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*, pages 842–851, 2013.

Erik Hatcher and Otis Gospodnetic. *Lucene in Action*. Manning Publications, 2004.

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)*, pages 34–43, 2014.

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).

Myles Hollander, Douglas A Wolfe, and Eric Chicken. *Nonparametric statistical methods*, volume 751. John Wiley & Sons, 2013.

René Just, Darioush Jalali, and Michael 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)*, pages 437–440. ACM, 2014.

Katja Kevic and Thomas Fritz. Automatic search term identification for change tasks. In *Proceedings of the International Conference on Software Engineering (ICSE'14)*, pages 468–471, 2014.

Otavio Augusto Lazzarini Lemos, Adriano Carvalho de Paula, Hitesh Sajnani, and Cristina 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)*, pages 41–50, 2015.

Tien-Duy B Le, Ferdian Thung, and David Lo. Predicting effectiveness of ir-based bug localization techniques. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE'14)*, pages 335–345, 2014.

Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE'15)*, pages 579–590, 2015.

Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves 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)*, ISSTA 2018, pages 61–72, 2018.

Zhixing Li, Tao Wang, Yang Zhang, Yun Zhan, and Gang Yin. Query reformulation by leveraging crowd wisdom for scenario-based software search. In *Proceedings of the Asia-Pacific Symposium on Internetware (Internet-*

ware'16), pages 36–44, 2016.

X. Allan Lu and Robert B Keefer. Query expansion/reduction and its impact on retrieval effectiveness. *NIST Special Publication*, pages 231–231, 1995.

Apache Lucene. `https://lucene.apache.org/`, 2017.

Fei Lv, Hongyu Zhang, Jian-guang Lou, Shaowei Wang, Dongmei Zhang, and Jianjun Zhao. Codehow: Effective code search based on api understanding and extended boolean model. In *Proceedings of the International Conference on Automated Software Engineering (ASE'15)*, pages 260–270, 2015.

Christopher D Manning, Mihai Surdeanu, John Bauer, Jenny Rose Finkel, Steven Bethard, and David McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of the Annual Meeting of the Association for Computational Linguistics (ACL'14)*, pages 55–60, 2014.

Andrian Marcus and Sonia Haiduc. Text retrieval approaches for concept location in source code. In *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, volume 7171 of *Lecture Notes in Computer Science*, pages 126–158. Springer, 2013.

Andrian Marcus, Andrey Sergeyev, Václav Rajlich, and Jonathan I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the Working Conference on Reverse Engineering (WCRE'04)*, pages 214–223, 2004.

Chris Mills, Gabriele Bavota, Sonia Haiduc, Rocco Oliveto, Andrian Marcus, and Andrea De Lucia. Predicting query quality for applications of text retrieval to software engineering tasks. *Transactions on Software Engineering and Methodology*, 26(1):3:1–3:45, 2017.

Chris Mills, Jevgenija Pantiuchina, Esteban Parra, Gabriele Bavota, and Sonia 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)*, pages 410–421, 2018.

Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei 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)*, pages 151–160, 2014.

Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *Proceedings of the International Conference On Automated Software Engineering (ASE'11)*, pages 263–272, 2011.

Brent D. Nichols. Augmented bug localization using past bug information. In *Proceedings of the Annual Southeast Regional Conference (ACMSE'10)*, pages 1–6, 2010.

Liming Nie, He Jiang, Zhilei Ren, Zeyi Sun, and Xiaochen Li. Query expansion based on crowd knowledge for code search. *IEEE Transactions on Services Computing*, 9(5):771–783, 2016.

Luca Ponzanelli, Andrea Mocci, and Michele Lanza. Stormed: Stack overflow ready made data. In *Proceedings of 12th Working Conference on Mining Software Repositories (MSR'15)*, pages 474–477, 2015.

Martin F Porter. An algorithm for suffix stripping. *Program*, 14(3):130–137, 1980.

Md Masudur Rahman, Jed Barson, Sydney Paul, Joshua Kayani, Federico Andrés Lois, Sebastián Fernandez Quezada, Christopher Parnin, Kathryn T. Stolee, and Baishakhi 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*, pages 465–475, 2018.

Mohammad Masudur Rahman and Chanchal K. Roy. Quickar: Automatic query reformulation for concept location using crowdsourced knowledge. In *Proceedings of the International Conference on Automated Software Engineering (ASE'16)*, pages 220–225, 2016.

Mohammad Masudur Rahman and Chanchal K. Roy. Strict: Information retrieval based search term identification for concept location. In *Proceeding of the Conference on Software Analysis, Evolution, and Reengineering (SANER'17)*, pages 79–90, 2017a.

Mohammad Masudur Rahman and Chanchal K Roy. Improved query reformulation for concept location using coderank and document structures. In *Proceedings of the International Conference on Automated Software Engineering (ASE'17)*, pages 428–439. IEEE Press, 2017b.

Mohammad Masudur Rahman and Chanchal K Roy. 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. (to appear).

Shivani Rao and Avinash Kak. Retrieval from software libraries for bug localization: a comparative study of generic and composite text models. In *Proceedings of the Working Conference on Mining software repositories (MSR'11)*, pages 43–52, 2011.

Michael Rath, David Lo, and Patrick Mäder. Analyzing Requirements and Traceability Information to Improve Bug Localization. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'18)*. ACM, 2018.

Manuel Roldan-Vega, Greg Mallet, Emily Hill, and Jerry Alan 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)*, pages 512–515, 2013.

Ripon K. Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E. Perry. Improving bug localization using structured information retrieval. In *Proceedings of the International Conference on Automated Software Engineering (ASE'13)*, pages 345–355, 2013.

G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, 18(11):613–620, 1975.

Carolyn B. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

David Shepherd, Zachary P. Fry, Emily Hill, Lori 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)*, pages 212–224, 2007.

Zhendong Shi, Jacky Keung, Kwabena Ebo Bennin, and Xingjun Zhang. Comparing learning to rank techniques in hybrid bug localization. *Applied Soft Computing*, 62:636–648, 2018.

Susan Elliott Sim, Medha Umarji, Sukanya Ratanotayanon, and Cristina V Lopes. How well do search engines support code retrieval on the web? *ACM Transactions on Software Engineering and Methodology*, 21(1):4, 2011.

Bunyamin Sisman and Avinash C. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'12)*, pages 50–59, 2012.

Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, pages 309–318, 2013.

Bunyamin Sisman, Shayan A. Akbar, and Avinash C. Kak. Exploiting spatial code proximity and order for improved source code retrieval for bug localization. *Journal of Software: Evolution and Process*, 29(1):e1805, 2016.

Jamie Starke, Chris Luce, and Jonathan Sillito. Searching and skimming: An exploratory study. In *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, pages 157–166, 2009.

Aoi Takahashi, Natthawute Sae-Lim, Shinpei Hayashi, and Motoshi Saeki. A Preliminary Study on Using Code Smells to Improve Bug Localization. In *Proceedings of the International Conference on Program Comprehension (ICPC'18)*, page 4. ACM, 2018.

Shaowei Wang and David 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)*, pages 53–63, 2014.

Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

Shaowei Wang, D. Lo, and J. Lawall. Compositional vector space models for improved bug localization. In *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, pages 171–180, 2014a.

Shaowei Wang, David Lo, and Lingxiao 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)*, pages 677–682, 2014b.

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)*, pages 262–273, 2016.

Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *Proceedings of the Conference on Software Mainte-*

nance and Evolution (ICSME'14), pages 181–190, 2014.

Yan Xiao, Jacky Keung, Kwabena E. Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 2018.

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*, 42(4):379–402, 2016a.

Xin Ye, Hui Shen, Xiao Ma, Razvan Bunescu, and Chang Liu. From word embeddings to document similarities for improved information retrieval in software engineering. In *Proceedings of the International Conference on Software Engineering (ICSE'16)*, pages 404–415, 2016b.

Klaus Changsun Youm, June Ahn, and Eunseok Lee. Improved bug localization based on code change histories and bug reports. *Information and Software Technology*, 82:177–192, 2017.

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. In *Proceedings of the International Conference on Software Maintenance and Evolution (ICSME'16)*, pages 110–121, 2016.

Jian Zhou, Hongyu Zhang, and David 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)*, pages 14–24, 2012.

Yu Zhou, Yanxiang Tong, Taolue Chen, and Jin Han. Augmenting bug localization with part-of-speech and invocation. *International Journal of Software Engineering and Knowledge Engineering*, 27(06):925–949, 2017.

Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schröter, and Cathrin Weiss. What Makes a Good Bug Report? *IEEE Transactions on Software Engineering*, 36(5):618–643, 2010.