2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER) | 978-1-7281-9630-5/20/$31.00 ©2021 IEEE | DOI: 10.1109/SANER50967.2021.00024

Combining Query Reduction and Expansion for Text-Retrieval-Based Bug Localization

Juan Manuel Florez[1], Oscar Chaparro[2], Christoph Treude[3], Andrian Marcus[1]
[1]*The University of Texas at Dallas,* [2]*College of William & Mary,* [3]*The University of Adelaide*
jflorez@utdallas.edu, oscarch@wm.edu, christoph.treude@adelaide.edu.au, amarcus@utdallas.edu

Abstract—**Automated text-retrieval-based bug localization (TRBL) techniques normally use the full text of a bug report to formulate a query and retrieve parts of the code that are buggy. Previous research has shown that reducing the size of the query increases the effectiveness of TRBL. On the other hand, researchers also found improvements when expanding the query (*i.e.*, adding more terms). In this paper, we bring these two views together to reformulate queries for TRBL. Specifically, we improve discourse-based query reduction strategies, by adopting a combinatorial approach and using task phrases from bug reports, and combine them with a state-of-the-art query expansion technique, resulting in 970 query reformulation strategies. We investigate the benefits of these strategies for localizing buggy code elements and define a new approach, called QREX, based on the most effective strategy. We evaluated the reformulation strategies, including QREX, on 1,217 queries from different software systems to retrieve buggy code artifacts at three code granularities, using five state-of-the-art automated TRBL approaches. The results indicate that QREX increases TRBL effectiveness by 4% - 12.6%, compared to applying query reduction and expansion in isolation, and by 32.1%, compared to the no-reformulation baseline.**

I. Introduction

Many techniques for bug localization leverage the fact that bug reports and source code share a substantial amount of vocabulary and use text retrieval techniques to find the buggy code artifacts [1–3]. A common problem faced by *text-retrieval-based bug localization* (TRBL) techniques is that bug reports are not written to be used as queries in a retrieval task. Instead, bug reporters focus on describing the observed (unexpected) software behavior (OB), the expected behavior (EB), and the steps to reproduce (S2R) the bug, often adding code snippets (CODE) and other information (OTHER), aiming to help developers reproduce and understand the bugs. In consequence, the performance of TRBL techniques is hindered by the presence of information that acts as noise.

One way to address this problem is reformulating the queries generated from bug reports (*a.k.a.* query reformulation [4]), which is an effective approach for improving TRBL [1, 5]. Two common reformulation methods are *query reduction* and *query expansion*. The former removes non-relevant terms from the initial query while the latter adds extra relevant terms. A recent query reformulation approach, BLIZZARD by Rahman *et al.* [6], selects the statistically most important terms from a bug report and adds terms extracted from the code documents retrieved by the entire report (*i.e.*, query expansion via pseudo-relevance feedback). More recently, Chaparro *et al.* [7] proposed a query reduction technique that relies on selecting the structural parts from a bug report (OB, EB, S2R, CODE, or the report TITLE) and discarding OTHER parts. At the same time, Haiduc *et al.* [8, 9] showed that some queries benefit from reduction, while others benefit from expansion.

A reasonable inference from this body of work is that combining query reduction and expansion techniques may lead to better query reformulations for TRBL. In this paper, we investigate this conjecture by evaluating 970 query reformulation strategies. These strategies come from the refinement of a query reduction approach that selects terms from bug report parts [7] combined with a query expansion technique [6].

The 970 query reformulation strategies were empirically evaluated by using five state-of-the-art TRBL approaches and 1,217 bug reports from multiple software systems to retrieve buggy code artifacts at three code granularities, namely file, class, and method. This data was manually curated and used in previous research [7], which allowed for comparing against previously proposed reformulation strategies.

We named the best performing strategy (of the 970) QREX. QREX first composes a reduced query by selecting (1) the most information-rich structural parts of the bug report [7], and (2) task phrases [10] from the remaining components. The reduction step is an improvement over the approach proposed by Chaparro *et al.* [7]. Finally, QREX selects the statistically important terms from the reduced query and adds related terms to produce a final query, using BLIZZARD [6]. QREX improves the effectiveness of TRBL by 32.1% over the baseline where the initial queries are not reformulated, and by 4% - 12.6% compared to the two techniques [6, 7] used individually.

In summary, the main contributions of the paper are:

1) A comprehensive investigation of 970 query reformulation strategies that implement query reduction and/or query expansion techniques to reformulate initial queries, and their impact on the effectiveness for TRBL; and
2) QREX, a new query reformulation technique for TRBL, which blends query reduction and expansion techniques, and is more effective than prior reformulation approaches [6, 7]. QREX combines new and existing reformulation techniques in an innovative way: (i) a new *combinatorial* approach for selecting the structural parts of the bug report for query reduction; (ii) a new reduction strategy that uses terms from task phrases [10] found in the OTHER parts of the bug report; and (iii) query expansion using BLIZZARD.

978-1-7281-9630-5/21/$31.00 ©2021 IEEE
DOI 10.1109/SANER50967.2021.00024

II. Query Reformulation Strategies

We introduce a set of query reformulation strategies that combines query reduction and expansion. The query reduction part uses a set of new query reduction strategies that extracts task phrases from different parts of a bug report using a *combinatorial* approach, while the query expansion uses BLIZZARD [6], a state-of-the-art query expansion approach.

We describe the strategies in this section, while Sections III and IV report their comprehensive evaluation, which allowed us to define QREX as the most effective of the strategies.

A. Query Reformulation Usage Scenario

Query reformulation strategies must be easy to use by developers and their usage should be well defined. With this in mind, we adapt the usage scenario proposed by Chaparro *et al.* [7], which is based on four steps:

1) First, *the developer issues an initial query* (manually or automatically) using the full text of the bug report to return a set of N (*e.g.*, 10) code candidates with the TRBL technique of their choice (*e.g.*, BRTracer [11]). The developer may use additional information required by the technique (stack traces from the bug reports, past bug report information, *etc.*).
2) Then, *the developer inspects the N returned candidates*, and if any of them is deemed buggy or if she opts to use another bug localization strategy (*e.g.*, dependency search) that leads to the buggy code, then the process ends and reformulating the query is not required.
3) Otherwise, *the developer reformulates the initial query* by, first, using a query reduction strategy, and second, using BLIZZARD to expand the reduced query. While this step is developer-initiated, it can be automated by integrating techniques such as DEMIBUD [12, 13], TaskNav [10], and BLIZZARD [6]. The developer runs the reformulated query with the same TRBL engine to obtain additional N code artifacts. The N results returned by the initial query are not included in the new N results retrieved by the reformulated query because they were deemed non-buggy.
4) Finally, *the developer investigates the new N results*. If a buggy code artifact is found within the result list (or it leads to the buggy code), then the bug localization process ends and the reformulation is *successful*. If still no buggy code artifacts are found, the reformulation is *unsuccessful*. At this point, the developer may employ a different reformulation strategy or switch to other methods for finding the buggy code.

B. The Content of Bug Reports

The proposed reformulation strategies leverage the structure of bug reports for query reduction, similarly to the work of Chaparro *et al.* [7]. A bug report is composed of different parts, notably the report title (TITLE), the observed (unexpected) system behavior (OB), the expected software behavior (EB), the steps to reproduce the bug (S2R), and code snippets (CODE). Chaparro *et al.* selected and combined the terms found in these parts as the reduced query. Their results suggest that these parts contain more relevant information for TRBL than the remaining content of the bug report (OTHER).

We contend that not all terms in the OTHER parts of the bug descriptions are irrelevant. We propose to extract *task phrases* [10, 14] from the OTHER parts of the report and use them in the query. We define OTHER as any natural language sentence that is not included in the TITLE, OB, EB, S2R, or CODE.

Task phrases (*a.k.a.* tasks) are natural language expressions in software documents (including bug reports) that describe how to accomplish some action [10, 14], *e.g.*, in a system or the source code. Task phrases are in the form of verbs associated with a direct object and/or a prepositional phrase. Specifically, task phrases are composed of three elements: `[action] [object] [predicate]`. The `[action]` is a verb phrase that indicates the operation performed by an actor (*i.e.*, the end-user, developer, or software system); the `[object]` is a noun phrase that corresponds to the entity affected by the action; and the `[predicate]` is a prepositional phrase that gives further details about the action or the object. For example, in the sentence "*set thumbnail size in templates*", "*set*" is the `[action]`, "*thumbnail size*" is the `[object]`, and "*in templates*" is the `[predicate]`. The `[predicate]` can be absent (*e.g.*, "*set thumbnail size*"). Task phrases can be automatically extracted from the bug report parts using existing tool support (*e.g.*, TaskNav [10]).

C. Query Reduction Strategies

We propose a set of query reduction strategies different from the ones proposed by Chaparro *et al.* [7]. In order to explain the differences, we formally define a query reduction strategy.

We define the set of bug report components $\mathbb{C} = \{T, O, E, S, C, R\}$, where each element denotes one of the six parts of a bug report we focus on: T=TITLE, O=OB, E=EB, S=S2R, C=CODE, and R=OTHER. Task phrases are extracted from these parts except C. We denote the set of task phrases from each part as $\mathbb{T} = \{T_t, O_t, E_t, S_t, R_t\}$. For example, tasks extracted from the OB are denoted as O_t.

We represent a bug report as the set $\mathbb{B} \subseteq (\mathbb{C} \cup \mathbb{T})$. A set $\mathbb{B}_1 = \{T, E, E_t, R, R_t\}$ would represent a bug report that contains TITLE, EB, and OTHER. This report also contains task phrases in its EB and OTHER. Note that T_t is absent; this means that there are no tasks in the TITLE of this bug report. Also, this bug report is missing OB, S2R, and CODE. If a component X is not present in the bug report, naturally there will be no tasks from that component (X_t) either.

A *query reduction strategy* indicates whether a bug report component should be included fully in the reformulated query, partially included (*i.e.*, only tasks from it should be included), or completely omitted. More formally, we define a strategy as:

$$\mathbb{S} \in (\mathcal{P}(\mathbb{C} \cup \mathbb{T}) - \varnothing), \nexists X \in \mathbb{C} \mid X \in \mathbb{S} \wedge X_t \in \mathbb{S} \quad (1)$$

This means that a strategy is a combination of parts in $\mathbb{C}$ and $\mathbb{T}$ with the property that if a full component is included in the strategy (*e.g.*, O), then tasks from that component cannot be part of the strategy (*i.e.*, O_t) and vice-versa. A missing component in the strategy indicates that it is omitted from the reduced query. For conciseness, we denote a reduction strategy as the concatenation of its components, *e.g.*, $\mathbb{S}_e = \{T, E, R_t\}$ becomes TER_t. This strategy indicates that the reformulated query should include only the TITLE, EB, and the tasks extracted from OTHER.

D. Combinatorial vs. Conjunctive Reformulations

Using the strategies defined above, we define two types of query reduction approaches. This is both a formalization and an improvement of the query reduction approach proposed by Chaparro *et al.* [7]. Section III discusses the implication of our improvement for the evaluation of the reformulation strategies.

A query reduction using strategy $\mathbb{S}$ applied to a bug report $\mathbb{B}$ is defined in Equation (2).

$$r(\mathbb{S}, \mathbb{B}) = \mathbb{R} \subset (\mathbb{C} \cup \mathbb{T}) \tag{2}$$

The reformulated query ($\mathbb{R}$) is obtained by selecting the parts specified in $\mathbb{S}$ from the bug report $\mathbb{B}$. If $\mathbb{R}$ is the empty set, then $\mathbb{B}$ cannot be reformulated with the strategy $\mathbb{S}$.

Our reformulations differ from those defined by Chaparro *et al.* [7] in *two major ways*: (1) the inclusion of *tasks* and OTHER, and (2) the applicability of the reformulations. With respect to (1), our strategies include OTHER and tasks extracted from the bug reports parts (*i.e.*, the elements of $\mathbb{T}$), which are not considered by Chaparro *et al.* [7].

With respect to (2), the reformulations defined by Chaparro *et al.* [7] can only be applied *if and only if* all parts from $\mathbb{S}$ are present in $\mathbb{B}$. We call such reformulation approaches *conjunctive*, as they require all parts of the strategy to be present in the bug report. In consequence, conjunctive reformulations can only be applied to a subset of the bug reports, which limits their applicability. For example, the best reformulation strategy identified by Chaparro *et al.* [7] is TOE, but it can only be applied to 22.5% of the bug reports (*i.e.*, only so many reports contain the T, O, and E components at the same time).

Conversely, we call our proposed reduction approaches *combinatorial*, as they use the maximal subset of the parts specified by $\mathbb{S}$ that are available in $\mathbb{B}$. This means that in the cases where a conjunctive reformulation cannot be applied because some of the parts from $\mathbb{S}$ are not present in $\mathbb{B}$, the combinatorial approach will use the remaining elements from $\mathbb{S}$ contained in $\mathbb{B}$ as the reformulated query. In consequence, our reformulations can be applied to more bug reports, compared to the conjunctive ones.

We formally define *applying a combinatorial strategy* as:

$$r_{comb}(\mathbb{S}, \mathbb{B}) = \mathbb{S} \cap \mathbb{B} \tag{3}$$

The combinatorial reduced query consists of the largest subset of strategy ($\mathbb{S}$) components found in the bug report $\mathbb{B}$.

Similarly, we formally define *applying a conjunctive strategy* to a bug report in Equation (4), which dictates that the reduced query is generated *only if all* components from the strategy $\mathbb{S}$ are present in the bug report $\mathbb{B}$.

$$r_{conj}(\mathbb{S}, \mathbb{B}) = \begin{cases} \mathbb{S} & \text{if } \mathbb{S} \cap \mathbb{B} = \mathbb{S} \\ \varnothing & otherwise \end{cases} \tag{4}$$

As an example, consider the strategy $\mathbb{S}_1 = \{O, S_t\}$, which indicates that the user should select the entire OB and the tasks from the S2R. $r_{comb}(\mathbb{S}_1, B)$ is the operation of applying $\mathbb{S}_1$ as a *combinatorial* reformulation on a bug report. This operation would result in one of the outcomes shown in Equation (5), depending on the bug report.

$$r_{comb}(\mathbb{S}_1, \mathbb{B}) = \begin{cases} \{O\} & \text{if } O \in \mathbb{B} \wedge S_t \notin \mathbb{B} \\ \{S_t\} & \text{if } O \notin \mathbb{B} \wedge S_t \in \mathbb{B} \\ \{O, S_t\} & \text{if } O \in \mathbb{B} \wedge S_t \in \mathbb{B} \\ \varnothing & \text{if } O \notin \mathbb{B} \wedge S_t \notin \mathbb{B} \end{cases} \tag{5}$$

Note that the options are all possible combinations of selecting and not selecting each one of the components in $\mathbb{S}_1$, hence the name combinatorial. In contrast, applying $r_{conj}(\mathbb{S}_1, B)$, *i.e.*, the *conjunctive* reformulation $\mathbb{S}_1$ on an arbitrary bug report B is shown in Equation (6).

$$r_{conj}(\mathbb{S}_1, \mathbb{B}) = \begin{cases} \{O, S_t\} & \text{if } O \in \mathbb{B} \wedge S_t \in \mathbb{B} \\ \varnothing & otherwise \end{cases} \tag{6}$$

E. Query Reformulation Strategies

We define 485 query reduction strategies based on the possible reformulation approaches specified above: each element of $\mathbb{C}$, except CODE (*i.e.*, five components), can be selected entirely, partially (only task phrases), or omitted; the CODE can be either selected or not selected. This translates into $3^5 \times 2 = 486$ possibilities. We do not consider $S_0 = \varnothing$ (*i.e.*, nothing is selected), resulting in 485 reduction strategies, including the 31 strategies proposed by Chaparro *et al.* [7]. All 485 query reduction strategies can be applied using either the combinatorial or the conjunctive approach.

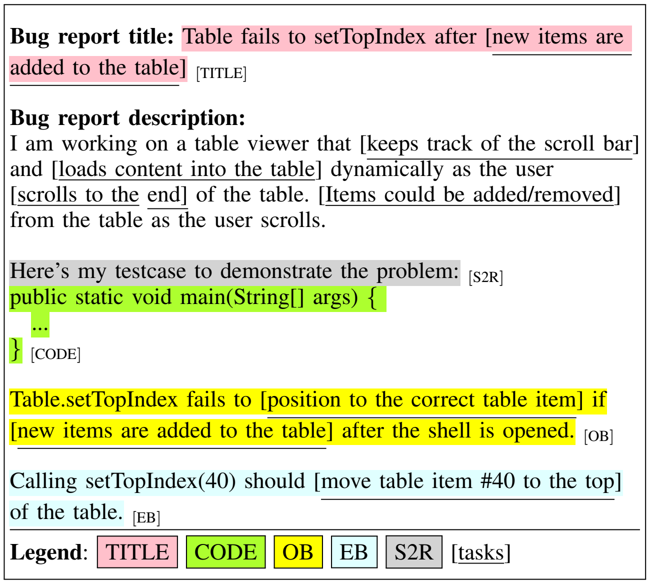
We propose to combine query reduction with statistical term selection and query expansion, using BLIZZARD [6] on the reduced queries. BLIZZARD is a query reformulation approach that leverages stack traces and the natural language found in bug reports, as well as code identifiers from the reports and source code. This information is represented in a graph that encodes relationships among the terms. The PageRank algorithm is then applied on the graph to determine the most important terms for TRBL. Depending on the bug report, BLIZZARD can select terms that compose the reformulated query and/or add extra terms to it.

Combining BLIZZARD with the query reduction strategies results in 485 additional strategies. We denote the use of BLIZZARD in combination with a reduction strategy by using the *b* postfix in the strategy. For example, using BLIZZARD with TER_t is denoted as TER_tb. Thus, in total, we define and investigate 970 query reformulation strategies.

F. Query Reformulation Example

Figure 1 shows an example of a bug report from the SWT project [15]. The sentences corresponding to the bug report structural components have been highlighted and color-coded. The fragments of sentences corresponding to tasks appear underlined between square brackets. If a developer were to apply the *combinatorial* reformulation with strategy TOS_tR_t, she would select (either manually or automatically): (1) the entire TITLE, and (2) the entire OB, and (3) the tasks from OTHER

Fig. 1: Excerpt from bug report #81264 (SWT 3.1.) [15]



Bug report title: Table fails to setTopIndex after [new items are added to the table] [TITLE]

Bug report description:
I am working on a table viewer that [keeps track of the scroll bar] and [loads content into the table] dynamically as the user [scrolls to the end] of the table. [Items could be added/removed] from the table as the user scrolls.

Here's my testcase to demonstrate the problem: [S2R]

```
public static void main(String[] args) {
  ...
}
```
[CODE]

Table.setTopIndex fails to [position to the correct table item] if [new items are added to the table] after the shell is opened. [OB]

Calling setTopIndex(40) should [move table item #40 to the top] of the table. [EB]

Legend: TITLE CODE OB EB S2R [tasks]

("*keeps track of the scroll bar*", "*loads content into the table*", "*scrolls to the end*", and "*Items could be added/removed*"). Notice that the S2R sentences do not contain any task phrases (*i.e.*, $S_t \notin \mathbb{B}$). Still, the bug report is reformulated using the combinatorial reformulation. In contrast, the conjunctive reformulation with the same strategy cannot be applied in this instance, as it requires all components to be present in the bug report. Finally, once the bug report is reduced, BLIZZARD is applied on the reduced query.

III. Empirical Evaluation Design

We conducted a comprehensive empirical study with four goals in mind: (1) evaluating the new combinatorial reformulations; (2) evaluating the use of task phrases for query reformulation; (3) assessing the effect of using query expansion/term selection; and (4) identifying the overall best reformulation strategy for defining QREX. Given these goals, we define five research questions (RQs):

- **RQ$_1$:** What is the effect on TRBL of the combinatorial reformulation approach, compared to the conjunctive one?
- **RQ$_2$:** What is the effect on TRBL of adding the tasks from OTHER into the existing reformulation strategies?
- **RQ$_3$:** What is the effect on TRBL of using tasks from the TITLE, OB, EB, and S2R for query reformulation?
- **RQ$_4$:** What is the effect of applying query expansion/term selection on the reduced queries?
- **RQ$_5$:** Which are the query reformulation strategies that lead to the best TRBL performance and how do they compare to state-of-the-art reformulation approaches?

For answering the RQs we used 5 TRBL techniques (Section III-A) to retrieve the buggy code artifacts for 1,217 *low-quality* queries/bug reports (Section III-B). Then, we used the proposed strategies to reformulate these queries, and compared their performance against the baseline that does not reformulate the queries and existing reformulations (Section III-C).

A. TRBL Techniques

The proposed reformulation strategies are independent of the TRBL technique, meaning that they can be used with any existing TRBL approach. In order to strengthen the generalization of our evaluation, we use five TRBL approaches, also used in prior empirical evaluations [7, 16]:

1) Lucene [17] combines the classical vector space model and the boolean text retrieval model and it can be used to retrieve code artifacts at *any code granularity* (*i.e.*, methods, classes, or files).
2) Lobster [18] leverages stack traces from bug reports, by boosting the relevance of code classes that appear in the traces or near the trace elements in the program dependency graph. Lobster works at *class-level granularity*.
3) BugLocator [19] leverages code information related to bugs that were reported and fixed in the past and also takes into account the length of code files. This technique boosts the relevancy of code files that are textually similar to files changed in previous bug fixes and to longer code files. BugLocator works at *file-level granularity*.
4) BRTracer [11] augments BugLocator by utilizing stack traces from bug reports and segmentation of code files. Similar to Lobster, it increases the relevancy of code files that appear in the traces. In addition, it compares segments of code files to the bug report (as opposed to the whole code file), and uses the highest similarity to represent the entire code file. BRTracer works at *file-level granularity*.
5) Locus [20] uses fine-grained code segmentation of files based on changes made in the project history. The relevancy of code files is determined by the relevancy of the small code segments and by the recency and frequency of these changes. Locus works at *file-level granularity.*

This sample of TRBL approaches was chosen with the intention of including multiple retrieval techniques that use diverse sources of information. If a reformulation strategy consistently improves the average performance for multiple TRBL approaches, this will increase our confidence on the robustness of the reformulation strategies, including QREX.

B. TRBL Data

We use the TRBL data used by Chaparro *et al.* [7], which includes 1,405 queries generated from entire bug reports (*i.e.*, their title/summary and description). The data spans 198 versions of 30 open-source projects written in Java, which vary in size and domain. Table I shows an overview of the dataset.

We used Chaparro *et al.*'s dataset for at least four reasons: (1) its bug reports were manually analyzed by multiple people to label the sentences corresponding to the TITLE, OB, EB, S2R, and CODE; (2) it was adapted from multiple sources in the existing research, including a recent TRBL reproducibility study (Bench4BL [16]), and from query reformulation [21] and fault localization [22] research; (3) it comes in three code granularities: file, class, and method; and (4) *low-quality* queries are already identified in the dataset; out of the 1,405 queries, 1,217 are *low-quality*, *i.e.*, they fail to retrieve the

TABLE I: Overview of the TRBL data used in the evaluation

Code granularity	# of systems*	# of versions*	# of queries*	# of lq^a queries
Class	13	16	360	270
File	11	99	832	158
Method	13	88	213	789
Total	**30**	**198**	**1,405**	**1,217**

* Total distinct items ^{a} *low-quality*

buggy code artifacts in the top-5 candidates returned by any of the five TRBL approaches.

The dataset includes: (1) the corresponding fixed (a.k.a., buggy or relevant) code artifacts for each query, which represent the ground truth; (2) the source code corpora (preprocessed and otherwise) for each project version, which represent the document search space for TRBL; and (3) the original bug reports used to generate the *low-quality* queries, which have their sentences labeled as corresponding to the TITLE, OB, EB, S2R, or CODE. From the labeled sentences we infer which sentences correspond to OTHER information.

We used TaskNav [10] to automatically extract the task phrases from the different parts of the bug reports. TaskNav extracts tasks corresponding to the definition of task phrases presented in Section II-B. Based on initial experimentation, we found that configuring TaskNav with the following options achieves a result closest to our tasks definition: (1) remove TaskNav's pre-defined list of programming verbs, which was created for software documentation; and (2) extract only task phrases starting with a direct object relationship.

Of the 1,217 bug reports, all have a TITLE and 67% include tasks in TITLE; 38% of the reports contain CODE snippets, 97% have OB, 23% have EB, 51% have S2R, and 82% include OTHER; 87% of the bug reports include tasks in OB, 18% in EB, 43% in S2R, and 64% include tasks in OTHER.

We used the implementation of BLIZZARD provided in the replication package of its publication [6]. We replicated BLIZZARD's evaluation results, which can be found in our online appendix [23]. BLIZZARD is applied on the reduced queries when a reformulation strategy indicates so. BLIZZARD's input is the reduced query, the bug report, and the (indexed) code corpus of the system. The output is a reformulated query having a subset of the terms from the reduced query (term selection) as well as extra terms (query expansion).

C. Query Execution and Measures

The empirical evaluation mimics the usage scenario described in Section II-A, which is based on four steps. In step one, the developer issues the initial query using the entire bug report, which is used for retrieval. In step two, she inspects the top-N results returned by a TRBL technique. If she does not find any buggy code artifact, then, in step three, she makes the choice of either reformulating the initial query by using a reformulation strategy or using the same initial query to retrieve N additional candidates (*i.e.*, no reformulation). Finally, in step four, she inspects the new N results looking for the buggy artifacts.

Note that, in practice, query reformulation can be done in an iterative fashion. We decided to control our experimental setting to these four steps, which involve one query reformulation step only (*i.e.*, step three). This allowed us to closely analyze the effects of the reformulations for TRBL.

The reformulated queries and the initial queries are executed using the five TRBL engines. However, not all techniques are designed to run on all code granularities, so we only execute them where applicable. This leads to 7 combinations of granularity and technique: Lucene-File, Lucene-Class, Lucene-Method, BugLocator-File, BRTracer-File, Locus-File, and Lobster-Class. Since the choice of N is important, we use six thresholds for the evaluation: $N \in \{5, 10, 15, 20, 25, 30\}$. We perform an evaluation of each reformulation for the *low-quality* queries at each N for each combination of granularity and technique. This means that the set of *low-quality* queries change for each N, since they are those initial queries that fail to retrieve the buggy artifacts in the top-N results (using any of the five TRBL engines). In the end, we have 42 (7×6) groups of retrieval results for each reformulation: one for each combination of granularity and technique (7), and each threshold N (6).

The retrieval effectiveness of both the initial queries (*i.e.*, the *no-reformulation baseline*) and the reformulated ones is compared using the metrics described below. When measuring the effectiveness, we only consider the N results retrieved in step three, which do *not* include the N results returned by the initial query in step two. In other words, we measure the ability of the reformulated queries in retrieving the buggy code artifacts in the next N results.

This experimental setting is similar to that used by Chaparro *et al.* [7]. However, there is one important difference in our evaluation. Different reformulations can be applied to different subsets of the queries. In consequence, in that work, the effectiveness between *two reformulations* was not directly compared. Instead, each reformulation was compared against the no-reformulation baseline. In contrast, in order to allow the direct comparison of reformulations, we change the evaluation as follows. When a reformulation $r(\mathbb{S}, \mathbb{B})$ cannot be applied to $\mathbb{B}$ (*i.e.*, when $r(\mathbb{S}, \mathbb{B}) = \varnothing$), we use the next N results of the initial query as the results of applying the strategy on that bug. We do this for all reformulations: the new combinatorial ones (which we propose) and the conjunctive ones (defined by prior work). In this way, every strategy is applied to all *low-quality* queries and they can be compared directly. This decision approximates the reformulation usage scenario better, considering that when a reformulation is not applicable, the user would simply not reformulate and just look at N more code artifacts retrieved by the initial query. Consequently, the applicability of the reformulations is directly measured by the retrieval effectiveness, as it is computed over the same set of queries, rather than by a separate applicability measurement as used by prior work [7].

We use well-known and widely used metrics to assess the performance of the strategies [6, 7]. %HITS@N (*a.k.a.* %H@N) is the percentage of queries for which at least one relevant code document is retrieved in the top-N candidates (in step three). %H@N values closer to one (1) indicate higher

TRBL performance. Mean reciprocal rank (MRR) averages $1/r_q$ across all queries, where r_q is the rank of the first relevant document retrieved by query q. As r_q gets smaller (the document ranks near the top of the result list), $1/r_q$ and MRR get closer to one (1). Mean average precision (MAP) aggregates the average precision p_q for each query q: $avg(p_q)$, where p_q is the average of the proportion of relevant code documents for q found in the top-k results whenever a relevant document is ranked in position k. As with MRR, a MAP value closer to one (1) indicates higher TRBL performance.

We focus the evaluation on %H@N, as we consider that it better reflects the reformulation usage scenario, given that it considers only the top-N retrieved candidates (as the end-user would do), as opposed to the entire list (as MRR and MAP do). An improvement in MRR and MAP can be significant, however, it is only relevant in this application if the improvement is the consequence of better rankings in top-N rather than outside the top-N candidates. Studying the variation of %H@N is also easy to interpret: it indicates how many queries are transformed from *low-quality* into *high-quality* or remain *high-quality* through the reformulations. We present the results of MRR and MAP, also, for the sake of completeness.

If the %H@N for the reformulated queries is higher than the one for the initial queries, we can conclude that the reformulation is more effective for retrieval than no reformulation. If the measures are the other way around, we can conclude that reformulation does not provide any benefit, as there is no gain over investigating N more results returned by the initial query. We apply the same reasoning when directly comparing two reformulations against each other.

IV. Evaluation Results and Discussion

We analyze the results of the empirical evaluation and answer the research questions. The complete results, statistical tests, and used data are available in our online appendix [23].

We summarize the results achieved by the no-reformulation baseline, averaging across thresholds, granularities, and TRBL techniques. The *baseline* achieves 0.23 %H@N, 0.06 MRR, and 0.07 MAP for ≈297 queries, on average. The performance is low because the queries are *low-quality*, so they represent the most challenging cases for automated TRBL.

A. RQ_1: Combinatorial vs. Conjunctive Strategies

To answer RQ_1, we need to establish in which cases a combinatorial reformulation $r_{comb}(\mathbb{S}, \mathbb{B})$ performs better on average than its conjunctive counterpart $r_{conj}(\mathbb{S}, \mathbb{B})$. For this, we focus on all reduction strategies except the strategies that include a single component only (*e.g.*, O), as for such strategies the combinatorial and conjunctive reformulations are the same. Since both types of reformulations target the query reduction strategies, we do not consider the ones that use Blizzard.

In the end, we compared 474 pairs of strategies. Each strategy in the pairs is used on all the queries, whose TRBL results are grouped and averaged over 42 subsets, corresponding to the combinations of threshold N, code granularity, and TRBL technique. We average the metrics to obtain an overall TRBL performance measurement. The Mann-Whitney test [24] was used to compare the 42 values for each pair of strategies. We consider the difference statistically significant if p-value<0.05.

Fig. 2: Avg. %H@N for $r_{conj}(\mathbb{S}, \mathbb{B})$ vs. $r_{comb}(\mathbb{S}, \mathbb{B})$. X-axis: strategies, ordered by the *combinatorial* Avg. %H@N.

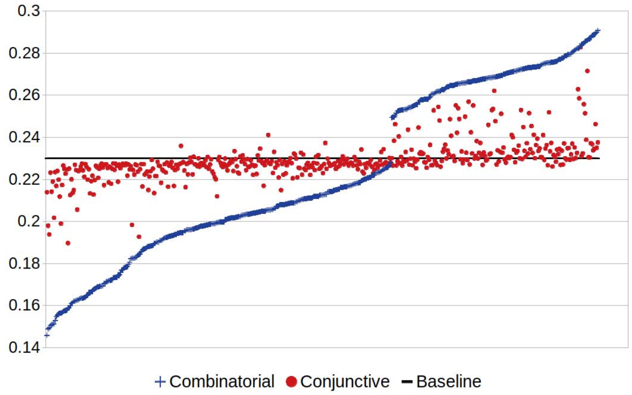


Fig. 2 shows the comparison of avg. %HITS@N for the combinatorial vs. conjunctive approaches, with the pairs of reformulations organized from left to right in order of ascending *combinatorial* avg. %H@N. The detailed results of the 474 comparisons can be found in our online appendix [23].

Overall, we found that 178 (38%) combinatorial strategies achieve higher avg. %H@N than their conjunctive counterparts. Out of these, 144 strategies (81% of 178) achieve statistically significant improvement. The overall avg. %H@N improvement of the combinatorial approach is 14.3%. The remaining 296 (62%) combinatorial strategies perform worse in terms of %H@N than their conjunctive counterparts. However, while most combinatorial reformulations do not improve over their conjunctive counterparts, we observed a clear trend. All 178 combinatorial reformulations with improvement over the conjunctive versions also achieve higher avg. %H@N than the baseline. Conversely, from the remaining 296 strategies where the combinatorial approach is not better, 218 (74%) of the conjunctive strategies do not achieve a higher avg. %H@N than the *baseline*. Furthermore, from the 237 (50% of 474) conjunctive strategies that improve over the baseline, 159 (67% of 237 or 89% of 178) of the combinatorial counterparts achieve higher avg. %H@N. This means that combinatorial reformulations significantly improve upon the best conjunctive reformulations, while being ineffective in cases where the conjunctive approach performs lower than the baseline.

In most cases, the combinatorial reformulations achieve improvement in terms of MRR and MAP (303 and 301 out of 474 strategies for MRR and for MAP, respectively). This improvement is statistically significant for 177 strategies for both MRR and MAP. The average improvement across strategies is 30% and 31% for MRR and MAP, respectively. The relative improvement in terms of MRR and MAP is higher than that in terms of avg. %H@N. However, we argue that this improvement does not imply that the strategy is more effective from the point of view of the developer, since most of it happens outside of the top-N results. This is exactly

Fig. 3: Avg. %H@N for $r(\mathbb{S}, \mathbb{B})$ vs. $r(\mathbb{S}R_t, \mathbb{B})$. X-axis: strategies, ordered by the Avg. %H@N with tasks from OTHER.

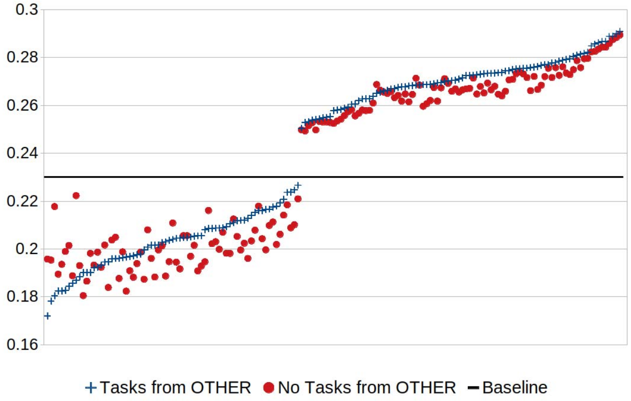
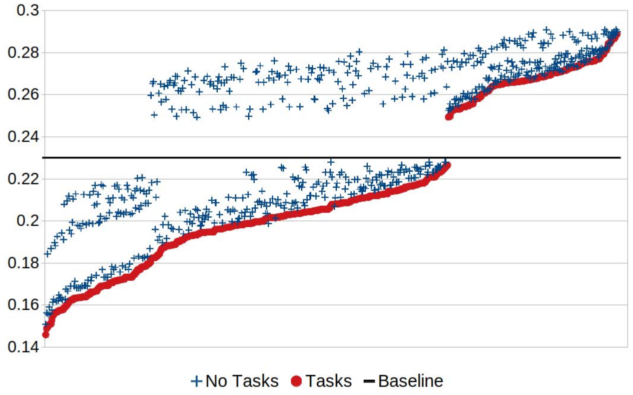


Fig. 4: Avg. %H@N for $r(\mathbb{Y}X, \mathbb{B})$ vs. $r(\mathbb{Y}X_t, \mathbb{B})$. X-axis: strategies, ordered by the Avg. %H@N with tasks.



the kind of intuition captured by the %H@N metric, which considers the top-N results only, hence our focus on it.

RQ$_1$ answer: Combinatorial reformulations improve TRBL performance in 178 of 474 (38%) cases by 14.3% avg. HITS@N, compared to the conjunctive counterparts. These reformulations achieve higher effectiveness than the baseline. In 144 (81%) of the cases, the improvement is statistically significant. From the remaining cases, 218 (74%) of the conjunctive reformulations are ineffective to begin with compared to the baseline, and so are the combinatorial ones.

Based on these results, we use only combinatorial reformulations in evaluating the strategies from this point forward.

B. RQ$_2$: Using Tasks from OTHER

As mentioned before, prior work [7] suggests that OTHER parts of the bug report contain noisy terms, and excluding them from the query improves TRBL. However, we argue that OTHER still contains relevant information for TRBL, which is present in task phrases. To answer RQ$_2$, we examine the effect of including tasks from OTHER as part of a strategy.

In this analysis, we compare strategy pairs $(\mathbb{S}, \mathbb{S}R_t)$ where $\mathbb{S}$ is any combination of TITLE, OB, EB, S2R, the tasks from these sources, and CODE; and $\mathbb{S}R_t$ is the same strategy with the addition of OTHER$_t$. We do not consider strategies that use BLIZZARD because we want to measure the effect of OTHER$_t$ during query reduction, hence we compare 161 pairs of strategies. The complete results can be found in our online appendix [23], including the results for MRR and MAP.

As seen in Fig. 3, 132 of the 161 (82%) strategies result in avg. %H@N improvement when tasks from OTHER (*i.e.*, OTHER$_t$) are used compared to when they are not. The improvement for these reformulations is 2.6% avg. H@N, which is statistically significant in 51 cases (39% of 132). Of the 132 strategies, 86 (65%) improve over the no-reformulation baseline. Of the 29 strategies that do not improve when adding OTHER$_t$, only 4 (14%) achieve improvement over the baseline. This means that most of the strategies that do not improve with OTHER$_t$ were ineffective to begin with.

RQ$_2$ answer: Adding tasks from OTHER to any strategy improves the TRBL performance in 132 of 161 cases (82%) with an average improvement of 2.6% HITS@N.

We briefly discuss a couple of cases in which adding tasks from OTHER had the largest positive effect on performance. In bug report #6009 from Derby [25], the focus on task phrases effectively filtered out examples that were detrimental to the retrieval. In bug report #1613 from OpenJPA [26], the majority of the extracted task phrases stem from a sentence explaining the underlying cause of the bug ("*All persistent classes in an inheritance hierarchy must use a single implicit field...*") rather than its symptom ("*MetaDataException...*"), thus improving the TRBL performance.

On the other hand, a case where task phrases had a negative result is bug report #5424 from Derby [27]. In this report, task phrases were only extracted from speculation around what was likely not the cause of the bug ('*'The test is newly converted with DERBY-5084 so not likely a regression...*") instead of the actual issue. In this case, the vocabulary of the tasks did not match well the lexicon of the expected buggy code artifact.

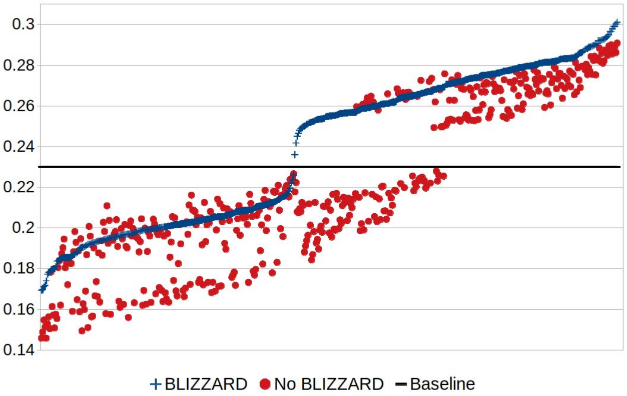
C. RQ$_3$: Using Tasks from non-OTHER Parts

To answer RQ$_3$, we measure the effect of using tasks from bug report parts different from OTHER. We compare pairs of strategies $(\mathbb{Y}X, \mathbb{Y}X_t)$ where X is one of TITLE, OB, EB, or S2R; and $\mathbb{Y}$ is a combination of components not containing X (using tasks or full content). As before, we do not consider the strategies that use BLIZZARD since we focus on query reduction. The pairs of this kind amount to 648 pairs total.

Fig. 4 shows the results. Only 8 out of 648 (1.2%) $\mathbb{Y}X_t$ strategies achieve higher avg. %H@N than the $\mathbb{Y}X$ strategies and none of them achieve higher avg. %H@N with statistical significance. 640 of 648 (98.8%) $\mathbb{Y}X_t$ strategies achieve lower avg. %H@N than the $\mathbb{Y}X$ strategies, with 9.5% avg. %H@N deterioration across strategies. 30 and 5 strategies achieve (0.6% & 0.4%) improvement for MRR and MAP, respectively.

RQ$_3$ answer: Using tasks from TITLE, OB, EB, or S2R instead of the full component reduces TRBL performance by 9.5% avg. HITS@N in 640 out of 648 cases (98.8%).

Fig. 5: Avg. %H@N for $r(\mathbb{S}, \mathbb{B})$ vs. $r(\mathbb{S}b, \mathbb{B})$. X-axis: strategies, ordered by the Avg. %H@N with BLIZZARD.



The interpretation of this result is that parts of the bug report that are not OTHER are dense in terms that are relevant to TRBL. Removing terms from these parts of the bug report is more likely than not to result in TRBL deterioration.

D. RQ_4: Assessing Query Expansion/Term Selection

We answer RQ_4 by comparing the TRBL performance of using and not using query expansion/term selection through BLIZZARD. We compare the strategy pairs ($\mathbb{S}$, $\mathbb{S}b$), where $\mathbb{S}$ is any of 485 reduction strategies and $\mathbb{S}b$ is the application of BLIZZARD on the reduced query generated by $\mathbb{S}$. We have in total 485 pairs and we present the results for avg. %H@N improvement in Fig. 5. Once again, the complete results can be found in the online appendix [23].

From the 485 strategies, 391 (80%) obtain higher TRBL performance when BLIZZARD is used, in terms of avg. %H@N. The improvement is 11% avg. H@N. In 295 cases (75% of 391), the %H@N improvement is statistically significant. The average improvement, in terms of MRR and MAP, is 22.8% and 21.2%, respectively. If BLIZZARD is used, 245 (62% of 391) strategies achieve higher avg. %H@N than the no-reformulation baseline, while when BLIZZARD is not used, only 154 (39% of 391) improve over the baseline. From the remaining 94 strategies with no improvement, 63 strategies (67%) are not better than the baseline independently of the application of BLIZZARD. For these cases, query reduction is not better than the baseline and applying query expansion/term selection on the reduced queries is not better either.

RQ_4 answer: Applying query expansion/term selection through BLIZZARD on the reduced queries improves the TRBL performance in 391 of 485 cases (80%) with an improvement of 11% (avg. %HITS@N). In 295 of these cases (75%), the improvement is statistically significant.

E. RQ_5: Best Reformulation Strategies

We analyze the combinatorial reformulations, based on all the 970 reformulation strategies described in Section II-E. That is, we consider strategies with and without tasks and with and without BLIZZARD. We compare the strategies with state-of-the-art reformulation techniques, namely BLIZZARD [6], which is applied on the full bug report; and the best reformulation identified by Chaparro *et al.* [7] (TOE) using the combinatorial approach (as opposed to the conjunctive one from [7], to enable direct performance comparison). We also compare each strategy with the no-reformulation baseline. We compute the avg. %H@N for each strategy, using the same six N thresholds as before, across TRBL techniques and granularities and we analyze the overall results as well as the results for each TRBL technique and granularity, separately. Table II shows the top-5 strategies that achieve the highest avg. %H@N. The detailed results for all the other strategies are available in the online appendix [23].

TABLE II: Strategies with the highest avg. %H@N

Strategy	Avg. %H@N	Avg. improv. vs baseline		
		%H@N	MRR	MAP
$TOER_tb$	0.30	32.09%	75.74%	75.10%
TOR_tb	0.30	31.70%	68.01%	69.24%
$TOES_tR_tb$	0.30	31.25%	76.08%	74.56%
TOE_tR_tb	0.30	31.20%	69.57%	70.90%
TOS_tR_tb	0.30	30.76%	69.45%	69.64%

TABLE III: Strategies that achieve avg. %H@N improvement for all techniques and all granularities.

Rank	Strategy	Avg. %H@N	Avg. improv. vs baseline		
			%H@N	MRR	MAP
1	$TOER_tb$	0.30	32.09%	75.74%	75.10%
4	TOE_tR_tb	0.30	31.20%	69.57%	70.90%
18	$TOER_t$	0.29	27.61%	64.20%	62.48%
21	$TOESR_t$	0.29	27.23%	66.24%	66.31%
29	$TOES_tR_t$	0.29	26.73%	65.38%	63.94%

The top-17 best strategies use BLIZZARD, including the ones shown in Table II, and achieve avg. %H@N between 29% and 30%. From these, 12 strategies include tasks from OTHER (R_t). The strategy that achieves the highest improvement (by 32.09% avg. HITS@N) over the no-reformulation baseline is $TOER_tb$, which achieves 30% avg. HITS@N (vs 22.8% of the baseline – see Table V). The best non-BLIZZARD strategy that does not use tasks (TOE) is ranked 24th with 28.9% avg. HITS@N (27% improvement). This is the best strategy proposed by Chaparro *et al.*, which is more effective than BLIZZARD alone, which achieves 26.7% avg. HITS@N and is ranked 259th. The results show the advantage of combining tasks from OTHER and BLIZZARD over the state of the art.

1) Detailed analysis: We analyze the results in more detail, by identifying the strategies that achieve improvements for all granularities and all techniques. Table III shows the top-5 of such strategies and their improvements. All five strategies include tasks from OTHER (*i.e.*, R_t) and the first two strategies

TABLE IV: Best strategies for each granularity-technique

GT[a]	Strategy	Avg. %H@N	Avg. improv. vs baseline		
			%H@N	MRR	MAP
BRT-F	$TOES_tR_tb$	0.24	4.73%	19.74%	21.61%
BL-F	$TOESR_t$	0.17	2.36%	2.25%	8.40%
LB-C	$TSCR_t$	0.38	107.36%	241.91%	241.37%
LC-F	$TOER_t$	0.39	26.25%	46.11%	48.41%
LU-C	TO_tESR_tb	0.46	46.89%	116.03%	120.24%
LU-F	TOE_tR_tb	0.30	40.88%	83.91%	81.28%
LU-M	TO_tSCRb	0.25	35.03%	59.05%	51.08%

[a] Granularity-Technique: F=File, C=Class, M=Method, LU=Lucene, LC=Locus, BL=BugLocator, BRT=BRTracer, LB=Lobster

TABLE V: Results for the best strategies (with R_t or b), the best strategies from prior work, and the no-reformulation baseline.

Strategy	Avg. H@N		Avg. % of queries			Avg. % of baseline vs reformulated queries					
	#	%	Improv.	Deter.	Equal	$L \rightarrow H$	$H \rightarrow L$	$H \rightarrow H$	$L \rightarrow L$	$L_{+2N} \rightarrow H$	$H \rightarrow L_{2N}$
$TOER_tb$	77.0	0.30	**58.5%**	38.3%	**3.2%**	14.0%	6.7%	16.1%	63.2%	8.7%	2.8%
$TOEb$	73.8	0.30	54.1%	42.4%	3.4%	**14.2%**	7.5%	15.3%	**63.0%**	**9.2%**	2.7%
$TOER_t$	74.5	0.29	53.0%	**31.7%**	15.3%	11.5%	**5.2%**	**17.6%**	65.7%	6.9%	**1.8%**
TOE [7]	73.1	0.29	50.7%	35.6%	13.7%	12.2%	6.1%	16.7%	65.0%	7.7%	2.0%
BLIZZARD [6]	70.1	0.27	52.4%	43.6%	4.0%	10.6%	6.7%	16.1%	66.6%	6.1%	3.3%
Baseline	63.2	0.23	–	–	–	–	–	–	–	–	–

also include BLIZZARD , which are the 1st and 4th with the highest overall TRBL performance. We observe that all these strategies share the TITLE, the OB, the EB (in a few cases, tasks from EB), and tasks from OTHER.

We also investigate the best strategy for each pair of technique and granularity independently and show the results in Table IV. For six of seven technique-granularity combinations, the best strategy includes tasks from OTHER (*i.e.*, R_t). Four of the seven strategies use BLIZZARD. The avg. %H@N achieved by the best strategies across technique-granularity combinations ranges from 17.5% to 45.8% (compared to the 17.1% - 31.6% avg. H@N of the baseline). $TOER_tb$ ranks between position 2 and 12 for all technique-granularity combinations except LB-C and LC-M, for which it ranks in positions 176 and 163, respectively. The TRBL performance achieved by $TOER_tb$ ranges from 17.3% to 45.6% avg. HITS@N.

Table V compares the best reformulation strategies that use tasks and/or BLIZZARD with the best approaches from prior work. The results are sorted by avg. %H@N and show that our proposed strategies are more effective. The table also provides detailed results of the queries. It shows the proportion of queries that the strategies improve (column #4), deteriorate (column #5), and achieve equal avg. %H@N (column #6) with respect to the no-reformulation baseline. The table reveals that the best strategy ($TOER_tb$) improves more queries (58.5%) than the other strategies while deteriorating 38% of them. Applying BLIZZARD has an effect on the queries that achieve equal performance to the baseline, as the query proportion goes down from ≈13%/15% to ≈3%. In fact, BLIZZARD alone makes 4% of the queries to achieve the same performance as the baseline. However, compared to the other strategies, BLIZZARD's deterioration rate is the highest (43.6%).

We also analyze the (avg.) proportion of queries that are *low-quality* and become *high-quality* when they are reformulated (column #7 from Table V: $L \rightarrow H$). Unlike *low-quality* queries, *high-quality* queries retrieve at least one buggy code artifact within the top-N results (in the **3rd step** of the reformulation usage scenario described in Section II-A). We analyze all possible transitions between *low-* and *high-quality* queries (columns #7-10 from Table V). The results show that $TOER_tb$ and $TOEb$ achieve the highest $L \rightarrow H$ rate (14%+) while achieving an acceptable $H \rightarrow H$ query proportion (≈15%-16%), compared to the other strategies. $H \rightarrow H$ queries are those *high-quality* that remain *high-quality* after reformulation. Table V reveals that these two strategies are able to turn the *low-quality* queries that retrieve the buggy code artifacts below the top-2N results into *high-quality* queries ($L_{+2N} \rightarrow H$) in more cases than for the other strategies. This is the best-case scenario of query improvement. Likewise, these two strategies turn *high-quality* queries into L_{2N} queries ($H \rightarrow L_{2N}$) in more cases than the other strategies. This is the least-harmful scenario of deterioration, where *high-quality* queries are deteriorated into *low-quality* ones that retrieve the buggy code artifacts between positions N and $2N$ of the result list. These results mean that these two strategies behave better in the best-case scenarios of improvement and deterioration. In other words, other strategies improve queries to a lesser extent and deteriorate queries to a greater extent than $TOER_tb$ and $TOEb$ do. The results explain the overall avg. %H@N performance achieved by these strategies, and the combination of $L \rightarrow H$ and $H \rightarrow H$ cases make $TOER_tb$ stand out over $TOEb$ as the best strategy overall.

RQ$_5$ answer: $TOER_tb$ is the strategy with the highest TRBL performance across all TRBL techniques and code granularities (30.1% avg. HITS@N). This strategy outperforms state-of-the-art approaches by 4% (TOE [7]) and 12.6% (BLIZZARD [6]), and the no-reformulation baseline by 32.1% avg. HITS@N. However, there are slightly better strategies for some technique-granularity combinations. Each combination has a different best strategy.

F. Definition of QREX

We construct QREX using the $TOER_tb$ strategy, which employs the combinatorial reformulation approach. As shown by the evaluation, QREX improves over the state of the art (on average) and is robust over the selection of thresholds N, the TRBL engine applied, and code granularity.

The user scenario for QREX corresponds to the 4-step process detailed in Section II-A. Namely, the developer first runs the TRBL engine of her choice with the full text of the bug report. If she is unsuccessful in locating the buggy code after examining the top-N results, she would select as many of the following components as are available in the bug report: the TITLE, OB, EB, and task phrases from OTHER. Finally, the selected text will be input to BLIZZARD, and the resulting reformulated query will be run again with the same TRBL engine. We anticipate that selecting these components would take developers only slightly longer than it would take them to read the full report.

Our online appendix [23] reports the detailed TRBL performance achieved by QREX for each N, engine, and granularity.

V. THREATS TO VALIDITY

Threats to *internal validity* stem from the data and TRBL techniques used in our study. To mitigate these threats, we used datasets and TRBL tools used in existing research [7, 16, 28]. The datasets were carefully curated by their authors to remove spurious issues and buggy code artifacts. The identification of bug report parts is another threat. We used the labeled data provided by Chaparro *et al.* [7], who used a rigorous coding process and reported a high inter-coder agreement. Another threat is the use of TaskNav [10] for extracting tasks from the bug reports. While TaskNav's evaluations [10] reported high detection precision, some extracted tasks may be false positives and some others may have been missed by the tool.

Construct validity is affected by the metrics we chose to compare reformulations. We use the rank of the 1st relevant document as a proxy for the user finding a buggy code artifact within the top-N results. This experimental setup is standard in TRBL and query reformulation research [6, 7, 16, 29]. We consider that the %H@N metric is straightforward to interpret and it matches a realistic TRBL scenario better than MRR and MAP do. Nonetheless, we report the MRR and MAP results.

To increase the *external validity*, we used TRBL data consisting of 198 versions of 30 open-source software systems, from multiple domains and types (*e.g.*, libraries, web applications, *etc.*). These data were manually curated and used in previous research. We used five different TRBL approaches, which retrieve code at multiple granularity levels.

VI. Related Work

Our research is motivated by the work of Chaparro *et al.* [7, 28, 30], who showed that removing irrelevant query terms can lead to substantial TRBL improvement. Mills *et al.* [1] confirmed this finding and found that bug report vocabulary is all that is required to formulate effective queries. Chaparro *et al.* [7] found that selecting the title, the observed behavior (OB), and the expected behavior (EB) from the bug report is the strategy that performs best, yet its applicability is affected by the absence of the EB in many bug reports. Our work introduces a new way to identify relevant terms for query reduction, based on task phrases from OTHER parts of the report. Compared to Chaparro *et al.*'s approach, our combinatorial strategies are more effective and, by definition, more applicable because they depend less on uncommon elements in bug reports (*e.g.*, EB).

The work by Rahman *et al.* [6] is leveraged in our research. We combine our reduction strategies with BLIZZARD, and we empirically show that the combination leads to higher TRBL performance than BLIZZARD alone. Related to this work, Rahman and colleagues developed approaches that identify search terms based on (1) structured source code entities and their co-occurrences [31, 32]; (2) TextRank and POSRank [33]; and (3) mining and using Stack Overflow data for translating a query into a list of API classes [34–36].

Lemos *et al.* [37] apply automatic query expansion using WordNet and a thesaurus containing software-related word relations. In follow-up work, Lemos *et al.* [38] found that in some situations it is best to use keywords only, when these are sufficient to semantically define the desired function.

Hill *et al.* [39] compared two approaches for incorporating word proximity and order in retrieval: one based on ad-hoc considerations and another based on Markov Random Field (MRF) modeling. Sisman *et al.* [40] later confirmed that a Markov-model-based approach can outperform bag of words. In an earlier approach, Sisman and Kak [41] achieved significant improvements using a reformulation approach based on pseudo-relevance feedback.

Gay *et al.* [29] used relevance feedback in TR-based concept location in which developers judge search results and the TR system uses this information to perform a new search. Lu *et al.* [42] presented a similar approach to interactively reformulate a query based on the relations between words in source code.

Haiduc and colleagues [8, 9, 21] showed that different reformulation approaches are needed depending on the quality of the queries, and some queries benefit from reduction while others benefit from expansion. Mills *et al.* [5] showed that no single bug report component (EB, OB, *etc.*) can be assumed to contain optimal terms for TRBL. We support this finding and show that a combinatorial selection of certain components can lead to TRBL performance improvement on average.

VII. Conclusions and Future Work

An empirical study of 970 query reformulation strategies on 1,217 bug reports from multiple software systems led to the conclusion that the TITLE, OB, and EB are dense in relevant terms for TRBL. More importantly, our results suggest that the OTHER parts of a bug report also contain relevant information for retrieving buggy code artifacts. We found that (part of) such information is encoded in *task phrases*.

The evaluation of the 970 reformulation strategies allowed us to define a new query reformulation approach for TRBL applications (QREX), which uses query reduction on entire bug reports used as initial queries, and then, it applies query expansion/term selection on the reduced queries. Query reduction is done using a novel combinatorial selection of bug report components and task phrases. QREX is more effective and applicable than existing reformulation approaches.

In practice, QREX is meant to be used as a recommender system that assists the developers during bug localization when they use TRBL engines. When the developer wants to reformulate the query (*i.e.*, after inspecting N results), QREX recommends the developer to select the TITLE, OB, EB, and the tasks from OTHER parts of the bug report (if available) and then apply BLIZZARD to expand the reduced query.

Our future work will focus on automating the reformulations produced by QREX, as currently, developers need to use several unconnected tools or manual reformulations. In addition, we will evaluate the reformulation strategies using additional query expansion approaches.

Acknowledgments

This research was supported in part by grants from the National Science Foundation: CCF-1848608, CCF-1910976, CCF-1955837, and CCF-1955853.

References

[1] C. Mills, J. Pantiuchina, E. Parra, G. Bavota, and S. Haiduc, "Are bug reports enough for text retrieval-based bug localization?" in *Proceedings of the 34th IEEE International Conference on Software Maintenance and Evolution (ICSME'18)*, 2018, pp. 410–421.

[2] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: A taxonomy and survey," *Journal of Software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2012.

[3] A. Marcus and S. Haiduc, "Text retrieval approaches for concept location in source code," in *Software Engineering: International Summer Schools, ISSSE 2009-2011, Salerno, Italy. Revised Tutorial Lectures*, ser. Lecture Notes in Computer Science. Springer, 2013, vol. 7171, pp. 126–158.

[4] X. A. Lu and R. B. Keefer, "Query expansion/reduction and its impact on retrieval effectiveness," *NIST Special Publication*, pp. 231–231, 1995.

[5] C. Mills, E. Parra, J. Pantiuchina, G. Bavota, and S. Haiduc, "On the relationship between bug reports and queries for text retrieval-based bug localization," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3086–3127, Sep. 2020.

[6] M. M. Rahman and C. K. Roy, "Improving ir-based bug localization with context-aware query reformulation," in *Proceedings of the 26th ACM Joint Meeting on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, 2018, pp. 621–632.

[7] O. Chaparro, J. M. Florez, and A. Marcus, "Using bug descriptions to reformulate queries during text-retrieval-based bug localization," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2947–3007, 2019.

[8] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 842–851.

[9] S. Haiduc, G. De Rosa, G. Bavota, A. Marcus, R. Oliveto, and A. De Lucia, "Query quality prediction and reformulation for source code search: the refoqus tool," in *Proceedings of the International Conference on Software Engineering (ICSE'13)*, 2013, pp. 1307–1310.

[10] C. Treude, M. Sicard, M. Klocke, and M. Robillard, "Tasknav: Task-based navigation of software documentation," in *Proceedings of the 37th IEEE International Conference on Software Engineering (ICSE'15)*, 2015, pp. 649–652.

[11] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, "Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 181–190.

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

[13] Y. Song and O. Chaparro, "Bee: a tool for structuring and analyzing bug reports," in *Proceedings of the 28th Joint Meeting on Foundations of Software Engineering (ESEC/FSE'20)*, 2020, pp. 1551–1555.

[14] C. Treude, M. P. Robillard, and B. Dagenais, "Extracting development tasks to navigate software documentation," *IEEE Transactions on Software Engineering*, vol. 41, no. 6, pp. 565–581, 2015.

[15] E. issue tracker, "Swt bug report #81264," 2020. [Online]. Available: https://bugs.eclipse.org/bugs/show_bug.cgi?id=81264

[16] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4bl: Reproducibility study on the performance of ir-based bug localization," in *Proceedings of the 27th International Symposium on Software Testing and Analysis (ISSTA'18)*, ser. ISSTA 2018, 2018, pp. 61–72.

[17] E. Hatcher and O. Gospodnetic, *Lucene in Action*. Manning Publications, 2004.

[18] L. Moreno, J. J. Treadway, A. Marcus, and W. Shen, "On the use of stack traces to improve text retrieval-based bug localization," in *Proceedings of the Conference on Software Maintenance and Evolution (ICSME'14)*, 2014, pp. 151–160.

[19] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports," in *Proceedings of the International Conference on Software Engineering (ICSE'12)*, 2012, pp. 14–24.

[20] M. Wen, R. Wu, and S. Cheung, "Locus: Locating bugs from software changes," in *Proceedings of the 31st International Conference on Automated Software Engineering (ASE'16)*, 2016, pp. 262–273.

[21] C. Mills, G. Bavota, S. Haiduc, R. Oliveto, A. Marcus, and A. De Lucia, "Predicting query quality for applications of text retrieval to software engineering tasks," *Transactions on Software Engineering and Methodology*, vol. 26, no. 1, pp. 3:1–3:45, 2017.

[22] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA'14)*. ACM, 2014, pp. 437–440.

[23] J. M. Florez, O. Chaparro, C. Treude, and A. Marcus, "Combining query reduction and expansion for text-retrieval-based bug localization – online appendix," 2021. [Online]. Available: https://doi.org/10.5281/zenodo.4431018

[24] M. Hollander, D. A. Wolfe, and E. Chicken, *Nonparametric statistical methods*. John Wiley & Sons, 2013, vol. 751.

[25] A. issue tracker, "Derby bug report #6009," 2020. [Online]. Available: https://issues.apache.org/jira/browse/DERBY-6009

[26] ——, "Openjpa bug report #1613," 2020. [Online]. Available: https://issues.apache.org/jira/browse/OPENJPA-1613

[27] ——, "Derby bug report #5424," 2020. [Online]. Available: https://issues.apache.org/jira/browse/DERBY-5424

[28] O. Chaparro, J. M. Florez, and A. Marcus, "Using observed behavior to reformulate queries during text retrieval-based bug localization," in *Proceedings of the 33rd International Conference on Software Maintenance and Evolution (ICSME'17)*, 2017, pp. 376–387.

[29] G. Gay, S. Haiduc, A. Marcus, and T. Menzies, "On the use of relevance feedback in ir-based concept location," in *Proceedings of the International Conference on Software Maintenance (ICSM'09)*, 2009, pp. 351–360.

[30] O. Chaparro and A. Marcus, "On the Reduction of Verbose Queries in Text Retrieval Based Software Maintenance," in *Proceedings of the International Conference on Software Engineering (ICSE'16)*, 2016, pp. 716–718.

[31] M. M. Rahman and C. 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)*. IEEE Press, 2017, pp. 428–439.

[32] M. M. Rahman and C. Roy, "Poster: Improving bug localization with report quality dynamics and query reformulation," in *Proceedings of the International Conference on Software Engineering (ICSE'18)*, 2018, pp. 348–349.

[33] M. M. Rahman and C. 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)*, 2017, pp. 79–90.

[34] M. M. Rahman and C. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2018, pp. 473–484.

[35] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Code search in the ide using crowdsourced knowledge," in *Proceedings of the International Conference on Software Engineering (ICSE'17)*, 2017, pp. 51–54.

[36] M. M. Rahman, C. K. Roy, and D. Lo, "Automatic query reformulation for code search using crowdsourced knowledge," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1869–1924, 2019.

[37] O. A. L. Lemos, A. C. de Paula, F. C. Zanichelli, and C. V. Lopes, *Thesaurus-based Automatic Query Expansion for Interface-driven Code Search*, 2014, pp. 212–221.

[38] O. A. L. Lemos, A. C. de Paula, H. Sajnani, and C. V. Lopes, "Can the use of types and query expansion help improve large-scale code search?" in *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM'15)*, 2015, pp. 41–50.

[39] E. Hill, B. Sisman, and A. Kak, "On the use of positional proximity in ir-based feature location," in *Proceedings of the IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE'14)*, 2014, pp. 318–322.

[40] B. Sisman, S. A. Akbar, and A. C. Kak, "Exploiting spatial code proximity and order for improved source code retrieval for bug localization," *Journal of Software: Evolution and Process*, vol. 29, no. 1, p. e1805, 2017.

[41] B. Sisman and A. C. Kak, "Assisting code search with automatic query reformulation for bug localization," in *Proceedings of the Working Conference on Mining Software Repositories (MSR'13)*, 2013, pp. 309–318.

[42] J. Lu, Y. Wei, X. Sun, B. Li, W. Wen, and C. Zhou, "Interactive query reformulation for source-code search with word relations," *IEEE Access*, vol. 6, pp. 75 660–75 668, 2018.