Empirical Evaluation of Bug Linking

Tegawendé F. Bissyandé1, Ferdian Thung2, Shaowei Wang2, David Lo2, Lingxiao Jiang2 and Laurent Réveillère1

1LaBRI, University of Bordeaux, France
2Singapore Management University, Singapore
{bissyand,reveille}@labri.fr, {ferdianthung,shaoweiwang.2010,davidlo, lxjiang}@smu.edu.sg

Abstract—To collect software bugs found by users, development teams often setup bug trackers using systems such as Bugzilla. Developers would then fix some of the bugs and commit corresponding code changes into version control systems such as svn or git. Unfortunately, the links between bug reports and code changes are missing for many software projects as the bug tracking and version control systems are often maintained separately. Yet, linking bug reports to fix commits is important as it could shed light into the nature of bug fixing processes and expose patterns in software management.

Bug linking solutions, such as ReLink, have been proposed. The demonstration of their effectiveness however faces a number of issues, including a reliability issue with their ground truth datasets as well as the extent of their measurements.

We propose in this study a benchmark for evaluating bug linking solutions. This benchmark includes a dataset of about 12,000 bug links from 10 programs. These true links between bug reports and their fixes have been provided during bug fixing processes. We designed a number of research questions, to assess both quantitatively and qualitatively the effectiveness of a bug linking tool. Finally, we apply this benchmark on ReLink to report the strengths and limitations of this bug linking tool.

I. INTRODUCTION

Software bugs greatly affect system reliability and as such entail significant effort to learn how to avoid them, predict them, and fix them when they appear. Work on software maintenance [1]–[3] and evolution [4]–[6] often require information on both the bugs that are reported and the fixes that developers applied. Such valuable information is available in bug tracking systems such as Bugzilla and version control systems such as Subversion. When analyzed together, information from the two kinds of systems can be used to better understand software development and maintenance processes, measure software cost, triage and reduce duplicate bug reports, predict bug locations, recommend bug fixes, and many other software engineering tasks [2], [3], [7], [8]. Unfortunately, information from these two kinds of systems are generally maintained separately. Links between bug reports and bug fixes are therefore not readily available to researchers or practitioners to analyze.

To address the problem of bug linking, a number of solutions have been proposed. Most of the solutions that aim to establish bug links rely on the fact that meticulous developers, when pushing a fix into the code version control system, always insert specific information that identifies the corresponding bug [9]. Thus, these solutions can establish bug links based on heuristics to match a set of indicative keywords (e.g., Fixed, Bug) and the corresponding bug identifiers (e.g., #1234) in code change logs with those in bug reports [9]–[11]. Sureka et al. have used a probabilistic approach to trace such links [12]

Other research work has shown that available datasets in both bug tracking and version control systems are actually plagued by quality issues and require bug linking solutions to be augmented with heuristics for verifying the correctness of their results [13]. The Linkster tool was designed in this respect to enable an expert developer to quickly find, examine, and annotate relevant changes that were identified through heuristics [14]. However it does not solve the problem of incompleteness and bias in datasets as many “missing” links cannot be uncovered with these heuristics.

ReLink extends previous bug linking approaches by implementing an information retrieval based solution [15]. Using similarity metrics, ReLink is able to find up to twice more links found by previous approaches. To evaluate ReLink, however, the authors of ReLink used as the ground truth a dataset with links that were manually labeled by themselves and a posteriori by an Apache Web Server developer for their Apache Web Server dataset. Several issues are then raised by this process:

1) The collected “ground truth” is quantitatively constrained by the tediousness of a-posteriori manual labeling.

2) The data may be plagued by bias as the labelers are not the actual bug fixers, those who without doubt could link a bug report with all, and only, the commits that address it.

Furthermore, the effectiveness of ReLink has been evaluated only against traditional approaches without introducing variations in the input data, such as the quality and quantity of training data used in their bug linking process. We undertake to build a benchmark for evaluating the effectiveness of bug linking tools with a dataset of 10 programs1 and we provide a more extensive evaluation of ReLink.

To build the benchmark dataset, we investigate a set of clean data where the links between bug reports and code revisions that fix the bugs are well maintained. We perform a

1ReLink was originally assessed on 3 programs.
The structure of this paper is as follows. In Section II, we describe bug linking and ReLink in more details. Section III describes the dataset that we use as the benchmark. We elaborate how we obtain and use this dataset. Section IV details the research questions and the metrics that we use for assessing a bug linking tool. We describe our evaluation results in Section V. We provide a list of related studies in Section VI. We conclude with future work in Section VII.

II. Bug Linking

Bug linking is the process of integrating information from bug tracking systems with information from version control systems to map developer code changes with the corresponding reported issues/bugs. Once extracted, such information can be used to understand development activities and measure software maintainability which in return can be used to predict defects or recommend bug fixes and to help improve software quality.

Figure 1. An example of explicit bug link in Zxing

Figure 1 shows sample code commit log and issue/bug report that are from the Zxing project3. In this case, the developer who committed the code voluntarily referred to the bug report (Issue 50) that is handled by his proposed fix. Thus, the link between the two logs are explicit. Using heuristics for scanning change logs to match a set of common keywords, one can easily uncover a number of such explicit links. Previous approaches to bug linking leverage such heuristics for mining bug links. Unfortunately, such approaches have weaknesses:

- There are no specific formats for referring to bugs in code change logs, which makes it impossible to exhaustively and automatically uncover all explicit links. For example, developers may insert a bug identifier as part of a sentence (e.g., “solve problem 101”, “see #123”, “fixed 423”) with the possibility for typos (e.g., “fic 239”) [21], or may refer to the bug as an issue (“issue #184”), a problem report (“PR: 11312”), etc.

- Adding bug references to a change log is not mandatory. This leads to a situation where many commits that fix bugs have no references to the relevant bug. In this context, previous approaches to bug linking are insufficient. Figure 2 shows an example of a change log from the same Zxing project where no reference to the bug fixed by the commit is provided.

Incompleteness and bias are therefore two main problems with previous approaches and may impact other studies based on the links produced. These weaknesses have been recently addressed in a novel approach, namely ReLink, which is a recent work on bug linking.

Bug linking with ReLink is based on an algorithm for identifying and assessing a set of features of links in a two-
Term Frequency-Inverse Document Frequency (TFIDF) met-
words and normalization using a stemming algorithm along
represented as a
in which each document containing
in the corresponding bug report messages.

is likely to be described and commented with the same terms
state the problem that the commit resolves, and the problem

trieval technology to compute their similarity. Indeed, the
them good candidates for processing with Information Re-
change committers in software repositories and bug owners

between them could be identified, e.g. by mining bug report
although the person committing the bug fix is not always
in bug tracking systems. Indeed, they have observed that
Bug owner and change committer: ReLink authors
have performed an empirical study of explicit links to
establish that, often, there exist some relationships between
change committers in software repositories and bug owners
in bug tracking systems. Indeed, they have observed that
although the person committing the bug fix is not always
the one responsible for handling the bug report, the mapping
between them could be identified, e.g. by mining bug report
comments where developers discuss the fix.

Time Interval: The ReLink algorithm considers the
interval between the bug-fixing time and the change-commit
time to filter out false positives and confirm the possibility of
a link using a threshold inferred from the explicit links
that were identified by previous heuristics.

Bug owner and change committer: ReLink authors
have performed an empirical study of explicit links to
establish that, often, there exist some relationships between
change committers in software repositories and bug owners
in bug tracking systems. Indeed, they have observed that
although the person committing the bug fix is not always
the one responsible for handling the bug report, the mapping
between them could be identified, e.g. by mining bug report
comments where developers discuss the fix.

Text similarity: Finally, in ReLink, bug reports and
change logs are considered as text documents which make
them good candidates for processing with Information Re-
trival technology to compute their similarity. Indeed, the
text of a bug-fixing commit log is often meant to explicitly
state the problem that the commit resolves, and the problem
is likely to be described and commented with the same terms
in the corresponding bug report messages.

ReLink builds upon the Vector Space Model (VSM) [22]
in which each document containing $n$ distinct terms is
represented as a $n$-dimension vector. After preprocessing
of bug reports and change logs to remove common stop
words and normalization using a stemming algorithm along
with a synonym replacement phase, ReLink relies on the
Term Frequency-Inverse Document Frequency (TFIDF) met-

Figure 2. An example of missing link in Zxing

fold run. In the first run, ReLink relies on the explicit links
that can be uncovered with previous approaches to build a
learning base to learn about the features that characterize
bug links. In the second run, ReLink uses those features to
further recover "missing" links—links involving fix logs that
do not contain any explicit reference to their corresponding
bug reports. To select features of links, the ReLink authors
have first performed a manual analysis of some explicit links.
We briefly describe in the following the features considered
in ReLink.

Time Interval: The ReLink algorithm considers the
interval between the bug-fixing time and the change-commit
time to filter out false positives and confirm the possibility of
a link using a threshold inferred from the explicit links
that were identified by previous heuristics.

Bug owner and change committer: ReLink authors
have performed an empirical study of explicit links to
establish that, often, there exist some relationships between
change committers in software repositories and bug owners
in bug tracking systems. Indeed, they have observed that
although the person committing the bug fix is not always
the one responsible for handling the bug report, the mapping
between them could be identified, e.g. by mining bug report
comments where developers discuss the fix.

Text similarity: Finally, in ReLink, bug reports and
change logs are considered as text documents which make
them good candidates for processing with Information Re-
trival technology to compute their similarity. Indeed, the
text of a bug-fixing commit log is often meant to explicitly
state the problem that the commit resolves, and the problem
is likely to be described and commented with the same terms
in the corresponding bug report messages.

ReLink builds upon the Vector Space Model (VSM) [22]
in which each document containing $n$ distinct terms is
represented as a $n$-dimension vector. After preprocessing
of bug reports and change logs to remove common stop
words and normalization using a stemming algorithm along
with a synonym replacement phase, ReLink relies on the
Term Frequency-Inverse Document Frequency (TFIDF) met-

Algorithm 1: The ReLink algorithm

Input: $L$: set of all possibilities of links
$L_e ← ()$: Links found by ReLink;
$L_e ← mineLinksUsingPreviousHeuristics()$;
$(T_e, S_e) ← determineTimeAndSimilarityThresholds(L_e)$;
$M_{bug, log} ← determineCommenterAndCommitterMappings(L_e)$;

foreach link $l$ in $L_e$ do

if (bugCommenter(l), changeCommitter(l)) $∈ M_{bug, log}$ then

if $\exists: t ← bugCommentTime(l)$ $|$ satisfiesThreshold($t, T_e$) then

if $b ← bug report$; $c ← change log$;
$Sim_b ← computeTextSimilarity(b, c)$;
if satisfiesThreshold($Sim_b, S_e$) then

$L_e ← L_e + \{l_e, l\}$;

return $L_e + L_e$.

Algorithm 1 presents the overall high-level description
of ReLink’s processing steps. According to this algorithm,
ReLink produces links which include explicit links mined
through previous heuristics and missing links identified
based on selected features of links.

The ReLink paper for automatic recovery of missing links
reported very good performance: the average precision rate
was 89% and the average recall was 78% for a limited
dataset containing 3 projects. However, the ReLink authors
relied on a manually labeled “ground truth” for their ex-
periments. Given the importance of bug linking, we believe
that it is necessary to more thoroughly evaluate ReLink and
assess its effectiveness on a variety of projects, a variety
of training datasets, and a variety of usage scenarios to
effectively establish its strengths and limits with regards to
bug linking.

III. BENCHMARK DATASET

Our bug linking evaluation is performed based on a
benchmark dataset collected from ten open source projects
hosted by the Apache Software Foundation4. These
programs are described in Table I with the number of bug
reports considered for each program and the number of
labeled links collected. Overall, the dataset includes about
7,000 bug reports fixed by one or more commits, thus
leading to around 12,000 bug links in the dataset. Our choice
of these software systems is influenced by (1) the capabilities
of JIRA5, a commercial bug/issue tracking system used by
Apache projects, (2) the maturity of the projects, (3) the
diverse application domains of the projects, and (4) the
different programming languages used in the programs.

JIRA has various features that make it a desirable toolkit
for dealing with bug reports. For example, as a voting-
based system, JIRA is often relied upon for effectively
prioritizing important issues (e.g., the ones that interest users
the most) [23]. Another feature of JIRA is that it provides

4http://www.apache.org/
5https://www.atlassian.com/software/jira/
add-ons for connecting issues to revision control systems. In the case of the programs used in our benchmark dataset, the Apache JIRA-based issue tracker\(^6\) was linked to the Apache subversion repository\(^7\), allowing links to be automatically inferred when commits are checked into the repository. One benefit of the JIRA add-ons is that it is very convenient for programmers to refer to the bug report they are addressing in their commit logs. Thus, there are immediately more opportunities for recovering links with improved quality. In JIRA, issue/bug identifiers are composed of two parts separated by a dash: a keyword that identifies the project (e.g., “LUCENE” for the Lucene project) and a unique number for each issue/bug in the project.

Table I details the number of labeled links that can be extracted from the issue tracker for each program. These numbers only include links to bugs fixes in the development trunk, excluding branches. For an efficient link mining by the issue tracker, the strategy used stills requires some manual effort from the developers, thus introducing opportunities for wrong links if developers make typos or mistakenly use in their change log a format that may be confused with a bug reference format. Figure 3 details an example of a link automatically inferred by the issue tracker for the issue LUCENE−1. As one can immediately notice, this link was wrongly mined since the identifier in this case is actually part of a number of commits addressing the same bug. In this case, we assume that the unlinked commit was caused by missing references to its corresponding issue/bug report.

In order to use labeled links as the ground truth we have set to manually assess those links to ensure that they were properly inferred. 

Soundness: To assess the soundness of the labeled links, we have randomly sampled 100 links distributed across the ten projects and manually checked whether these were true links. We have found that 100% of those links were true. The link example in Figure 3 actually refers to a revision number that is part of a branch, thus outside the development trunk considered in our benchmark dataset.

This empirical evaluation reveals that Apache developers are meticulous in their efforts to insert bug references in the change logs of their fixing commits. However, we have also noted that some bug reports were automatically imported from a previous Bugzilla-based setup of the project’s bug tracking system into the JIRA install, which may reduce the reliability of links that were not labeled by bug fixers. We have therefore parsed all labeled links from our datasets and found that 37 out of 6507 involved a bug report with an initial Bugzilla ID. Manually checking these links exposed 22 wrong links with LUCENE−3. In all these links the expression “LUCENE-3”, which was actually part of the release version number “Lucene3.1.0”, was mistakenly inferred as a bug number during the automated labeling by JIRA. We have then removed these links from the benchmark dataset.

Completeness: Completeness is an important property that should be ensured to accurately evaluate false negative rates of bug linking tools. We find that every issue/bug in our benchmark dataset has been linked to some commits. In addition, for links provided by JIRA for the issues/bugs, we want to make sure whether all code commits related to an issue/bug have been linked to the issue/bug in JIRA. For this purpose, we randomly sampled 100 links to check whether a bug report may be related to a commit not linked by JIRA. Two situations may explain why a given commit is not linked to a bug report. First, the commit may not be a bug-fixing commit, or it may fix a bug that was not reported in the bug tracking system. Second, the commit may be part of a number of commits addressing the same bug. In this case, we assume that the unlinked commit was caused by missing references to its corresponding issue/bug report in its commit log when it actually belongs to the fix split in several commits. We have investigated the dataset and found that when a bug is fixed by many commits, those are usually close in time, and for the ten programs, the number of commits do not reach 20 for a given bug. Thus, for each linked commit in the sample set, we manually examine the 10 commits that precede it and the 10 commits that follow it to see whether there are unlinked commits for the same issue/bug. Our manual investigation has established that 100% of the sampled links were complete.

The results of these investigations allow us to use the links extracted from the JIRA-based issue tracking system with little clean-up as our ground truth for evaluating bug linking tools.

IV. RESEARCH QUESTIONS & METRICS

We now discuss a number of important research questions that we have formulated to assess the effectiveness of a bug

---

\(^6\)https://issues.apache.org/jira/

\(^7\)https://svn.apache.org/repos/asf/
linking tool.

**RQ1. How effective is the tool in recovering links for non-linked bug reports?**

In this research question, we propose to evaluate the completeness and accuracy of the links generated by a bug linking tool when provided with completely non-linked bug reports. Indeed, a tool may fail to find a link for some bug reports (false negatives) and may assign incorrect links to other bug reports (false positives). It is therefore important to assess the effectiveness of the bug linking tool based on such cases.

**RQ2. How effective is the tool in recovering links for partially-linked bug reports?**

Partial links refer to links involving bugs that are fixed in several commits but not all of the commits are explicitly linked to the bugs. Using partial links in studies may introduce bias whose impact can be significant [14]. It is therefore often necessary to identify, for every bug report, all the commits that are related to it. Intuitively, recovering such links could be more readily possible than in the case of completely non-linked bug reports, as the similarity between the commits can also be leveraged. In this research question, we investigate whether a bug linking tool could recover missing links from partially-linked bug reports and whether it could be more accurate in doing so.

**RQ3. What is the sensitivity of the tool when training data is changed?**

Advanced techniques for recovering missing links, as with the ReLink tool, use machine learning algorithms that rely on training data for computing the similarity thresholds for detection of bug links. Variations in real-world datasets may therefore impact the performance of such bug linking tools. Consequently, for a bug linking tool that relies on machine learning approaches, it is important to investigate its sensitivity when training data is changed.

**RQ4. Could the tool be trained on one software system and used to link reports in other software systems?**

Related to the previous research question, a worst case scenario may arise when no training data can be found in the project. For example, for the first bug report in a software project, there is no training data available. Because explicit links are not readily available in all real-world projects, a bug linking tool would be more valuable if it can use training datasets from one project to infer links in another. We explore in this research question if the thresholds learned in one software system could be used in other systems.

**RQ5. How effective is the tool as compared with standard information retrieval solutions including VSM, LSI, and LDA?**

Bug linking algorithms, such as ReLink, can be built atop of Information Retrieval technology. However, since there are many standard information retrieval solutions, it is important to survey the benefits of the linking algorithm compared to the results that can be directly obtained with standard techniques. In this research question, we propose to compare the performance of the tool with standard information retrieval solutions that are used to measure the textual similarity of two documents.

**RQ6. What kinds of links are often missed by the bug linking tool?**

When a bug linking tool misses some links (false negatives), what are the characteristics of those links? Thoroughly studying this question can give insights for researching new ways to collect more links. We propose to answer this research question by performing a qualitative study of the false negatives of the tool’s outputs.

**RQ7. What are the characteristics of extraneous links generated by the bug linking tool?**

Besides false negatives, a bug linking tool can generate false positives, i.e., incorrect links. Exploring the characteristics of those links can provide insights on the limits of the solution implemented by the bug linking tool, and suggest potential research methodology for improving bug linking tools.

To quantitatively evaluate a bug linking tool, we propose to use standard metrics from the field of Information Retrieval, namely the Precision, Recall, and F-measure metrics. 

- **Precision**, as captured by Equation (1), quantifies the effectiveness of the tool to recover links that are actually correct.

  \[
  \text{Precision} = \frac{\left| \{\text{labeled links}\} \cap \{\text{link inferred by tool}\} \right|}{\left| \{\text{links inferred by tool}\} \right|} \tag{1}
  \]

- **Recall** on the other hand explores the capability of the tool to recover most of the missing links. Equation (2)

To create a text representation of this document, I would need to have the entire document or the specific section you are interested in. If you have an image of the document or a specific page, please upload it or provide the details so I can assist you better.
These experiments show that, in general, the quantitative analysis reveals that ReLink tool as a black box described above. For our evaluation process, we use the ReLink tool and analyzing the outputs. We have downloaded the version of the tool that was available at the project web page at the time of writing.

Since our ground truth always contains references to the bug reports that are involved in a link, which would hinder the evaluation of ReLink’s capability in finding “missing links”, we accordingly pre-process the inputs of change logs to remove the references from the portions of data that are used for testing, but leaving them in the training data for ReLink to infer link features using traditional heuristics.

A. RQ1: Link Effectiveness (Non-Linked)

Since ReLink relies on a learning algorithm, k-fold cross validation is a well suited statistical method to evaluate its effectiveness [24]. To answer the first research question we therefore perform a 10-fold cross validation for each of the programs in our benchmark dataset. For this purpose, we have randomly distributed the labeled links into 10 sets of equal size. For each program, we ran 10 experiments using every time 1 set as the testing set and the 9 others for training data. The results are shown in Table II for all programs.

Discussion: These experiments show that, in general, the ReLink tool has good precisions, reaching 100% for the stdcxx program, though this precision can drop in some cases, as for the stdcxx program. Recalls, however, are very low, which in turn cause low F-measures. The recall of ReLink is sacrificed by the algorithm in favor of precision.

B. RQ2: Link Accuracy (Partially-Linked)

For the second research question, we consider bug reports that are involved in multiple labeled links, i.e., bugs for which there are more than one corresponding revisions. For each bug, we successively consider 25%, 50% and 75% of the relevant links for training, and compute the effectiveness of ReLink in recovering the remaining. The results of these experiments are shown in Table III.

Discussion: The quantitative analysis reveals that ReLink does not succeed in inferring partially missing links more than in the case of non-linked bug reports. We suspect that this is due to the fact that while different change logs may address the same bug, they often do so with different terms which in return will reduce the success of ReLink. Indeed, the ReLink algorithm strictly considers the similarity between 1 commit change log and 1 bug report and, thus, does not leverage the similarity between commits that address the same bug.

C. RQ3: Sensitivity to Training Data

To assess the sensitivity of ReLink to training data, we perform a series of experiments with varying sizes of the training data. Practically, for each program, we have randomly distributed the labeled links into nine buckets of potential training data. We then run 9 successive experiments where in the first experiment labels from the first bucket are used for training, and in the second experiment we add labels from the second bucket to double the size of training data, and so on. This experimental scenario enables us to consider from 10% to 90% of the labeled links as training data and the remaining, i.e., from 90% to 10% as testing data. Figures 4, 5 and 6 show the results of our experiments for the different programs.

Discussion: In Figure 4, we note that precision is not significantly impacted by the size of training data. We believe that this is due to the fact that ReLink uses different heuristics aside from the similarities between change logs and bug reports, to consolidate its outputs.
Table III

<table>
<thead>
<tr>
<th>% partial links used for training</th>
<th>activemq</th>
<th>felix</th>
<th>hadoop</th>
<th>lucene</th>
<th>mahout</th>
<th>opennlp</th>
<th>stdcxx</th>
<th>struts</th>
<th>xalan</th>
<th>xerces</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>1</td>
<td>0.977</td>
<td>0.133</td>
<td>0.036</td>
<td>0.016</td>
<td>0.03</td>
<td>0.058</td>
<td>0.103</td>
<td>0.03</td>
<td>0.261</td>
</tr>
<tr>
<td>Recall</td>
<td>0.114</td>
<td>0.092</td>
<td>0.235</td>
<td>0.07</td>
<td>0.032</td>
<td>0.059</td>
<td>0.091</td>
<td>0.188</td>
<td>0.059</td>
<td>0.414</td>
</tr>
<tr>
<td>F-measure</td>
<td>0.204</td>
<td>0.169</td>
<td>0.235</td>
<td>0.07</td>
<td>0.032</td>
<td>0.059</td>
<td>0.091</td>
<td>0.188</td>
<td>0.059</td>
<td>0.414</td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.111</td>
<td>0</td>
<td>0</td>
<td>0.16</td>
</tr>
<tr>
<td>Recall</td>
<td>0</td>
<td>0.008</td>
<td>0.015</td>
<td>0</td>
<td>0</td>
<td>0.023</td>
<td>0.038</td>
<td>0</td>
<td>0</td>
<td>0.276</td>
</tr>
<tr>
<td>F-measure</td>
<td>0</td>
<td>0.015</td>
<td>0.03</td>
<td>0</td>
<td>0</td>
<td>0.038</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.276</td>
</tr>
<tr>
<td>75%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Precision</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.083</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Recall</td>
<td>0</td>
<td>0.008</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.016</td>
<td>0.027</td>
<td>0</td>
<td>0</td>
<td>0.095</td>
</tr>
<tr>
<td>F-measure</td>
<td>0</td>
<td>0.015</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.027</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0.174</td>
</tr>
</tbody>
</table>

Figure 5. Recall – sensitivity to training data

Figure 6. F-measure – sensitivity to training data

Figure 7. Precision – Cross project evaluation

Figure 8. Recall – Cross project evaluation

Figure 9. F-measure – Cross project evaluation

D. RQ4: Cross Project Effectiveness

To answer the research question related to cross project effectiveness of ReLink, we ran experiments with all combinations of pair-program in our benchmark dataset. Thus, for each program, we consider training data from exclusively another program, and we repeat this scenario for all other programs. For a baseline result, we compute the effectiveness of ReLink when no training data is used. The results of our experiments are highlighted in Figures 7, 8 and 9.

Discussion: From the graphs detailing the precision and recall results, we observe that, overall, using training data from other projects datasets leads to lower precision and recall. In a few cases, such as with the mahout and opennlp programs, smaller sets of training data (e.g., from struts) have less impact on the precision of ReLink and may even improve it slightly.

E. RQ5: Comparison with Other IR Solutions

To answer this research question we use several standard information retrieval techniques namely vector space modeling (VSM), latent semantic analysis (LSA), and latent
Dirichlet allocation (LDA), which have also been used in studies on software traceability analysis. Following is the description of how we use these techniques for bug linking.

Practically for the purpose of fair comparison, we have implemented the considered standard information retrieval solutions to follow the same steps as described in the ReLink paper. These solutions perform a simple retrieval without considering some features of links (time interval and mapping between bug owner and change committer). The process for each model is the same. First, they pre-process the text data through stemming and stop words removal. Second, they take bug reports as query and search the relevant change logs for this query. For every bug report, the similarity scores between its text data and change logs are computed based on the model in use (VSM, LSA or LDA). Links are then inferred by selecting change logs for which the similarity score is above a threshold which was determined in a training phase as in ReLink.

To compare ReLink and the aforementioned IR techniques, we resort to 10-fold cross validation. The experiment scenario for each technique is similar to the one used for answering the first research question (RQ1) for ReLink. The results of these various IR techniques as compared to ReLink is shown in Table IV. From the table, we could notice that ReLink outperforms the existing IR techniques in terms of F-measure for: activemq, felix, lucene, mahout, struts, xalan, and xerces. VSM outperforms ReLink for: hadoop, openmlp, and stdexx. LDA and LSA outperform ReLink for: openmlp and stdexx. Thus in general, ReLink is better than existing IR approaches. The existing IR approaches are promising too as it could outperform ReLink on 2 or 3 out of the 10 programs. In the future, it would be interesting to propose an approach that could extend ReLink such that it could outperform all existing IR techniques on all datasets.

F. RQ6: Missing Links

In Figures 10 and 11 we detail two categories of missing links that Relink fails to recover. The examples are presented as used in the testing dataset where we had removed any explicit reference to the bug reports so as to assess the core algorithm of ReLink.

The first example highlights the fact that ReLink’s features of links could be augmented to take into account mappings between bug reporter name and patch acknowledgement texts. Indeed, although the change log and bug report description texts in Figure 10 are not similar, we can infer a link, based on the date, report metadata and the acknowledgement in change log.

Figure 10. ReLink missed link – Unleveraged feature of links

<table>
<thead>
<tr>
<th>a) Excerpt of commit change log</th>
</tr>
</thead>
<tbody>
<tr>
<td>revision: 682831 author: apetrelli date: 2008-08-05 17:50:09 msg: Applied patch provided by Yannick Haudry</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>b) Excerpt of issue report (JIRA)</th>
</tr>
</thead>
<tbody>
<tr>
<td>key_name: STR-3160 reporter_name: Yannick Haudry created: Tue, 22 Jul 2008 21:53:28 assignee_name: Antonio Petrelli resolved: Tue, 5 Aug 2008 18:09:13 resolution: Fixed summary: TilesRequestProcessor processTilesDefinition... description: Here is the code...</td>
</tr>
</tbody>
</table>

Figure 11. ReLink missed link – Excessive filtering

Figure 11 however details a different miss by ReLink. Although there exist text similarity between the change log and the bug report, and a mapping between the bug owner (i.e., assignee) and the change committer, ReLink dismisses a relevant link as the bug report was tagged “Resolved” until 1 month after it was actually fixed in the version control system. This kind of miss was also mentioned by the authors as a source of false negatives, explaining in part the poor recall of ReLink.

G. RQ7: Extraneous Links

The precision of ReLink results is usually very high as detailed in Sections V-A, V-B and V-C. This, as the authors have suggested in their paper [15], is largely due to the use...
### B. Empirical Evaluation & Evaluation Framework

A number of studies perform empirical evaluation to measure the effectiveness of existing approaches [27]–[30]. Lo and Khoo propose an evaluation framework called QUARK that evaluates existing automata-based specification mining tools [27]. Bogdanov and Walkinshaw extend QUARK by proposing a new metric to evaluate automata-based specification mining tools [28]. Pradel et al. extend the above two studies by yet another metric which is shown to outperform the existing metrics [31].

Engstrom et al. compare and contrast various regression test selection techniques [32]. Hutchins et al. evaluate the effectiveness of dataflow and control-flow based test adequacy criteria [33]. They produce a set of benchmark programs often referred to as the Siemens test suite. Siemens test suite itself has been widely used to evaluate many fault localization approaches, e.g., [34]–[38]. Jones et al. empirically evaluate a fault localization tool called Tarantula in [35] first proposed in [34]. Lucia et al. empirically evaluates the effectiveness of various association measures proposed in the data mining and statistics community for fault localization [38].

Wang et al. compare and contrast many information retrieval solution for concern localization problem (i.e., the detection of traceability links between a requirement document to program elements that implement it) [20]. Lamkanfi et al. investigate the effectiveness of various classification algorithms for the task of predicting severity labels of bug reports [39].

In this work, we also perform an empirical evaluation. Our study is orthogonal to the above as we are evaluating another important research problem namely the linking of bug reports to the revisions in source control repositories that fix them.

### VII. Conclusion and Future Work

Bug linking is an important problem which, if thoroughly addressed, will significantly improve software maintenance...
and evolution studies and enhance capabilities of various research tools for improving defect prediction and fix recommendations. Such studies are indeed largely discussed in the literature as useful for improving the quality of software.

In our work we provide a clean benchmark dataset for evaluating bug linking tools. We have applied several research questions to the state of the art tool, namely ReLink, to assess its effectiveness on recovering missing links. The results of our experiments show that, overall, ReLink achieves very good precisions, over 90%, for some programs, but delivers lesser recall rates, dropping below 10%. The F-measure results in our various scenarios show that there is room for improvement in the area of bug linking. Our qualitative assessments of ReLink’s missed and extraneous links, as well as the comparison with various standard IR techniques, point out some weaknesses in the algorithm and the filtering strategy of ReLink, thus opening up new directions for future work on bug linking.

**Availability.** The benchmark constructed in this work is available at: http://momentum.labri.fr/bugLinking

**REFERENCES**


