Empirical Evaluation of Bug Linking

Tegawendé F. Bissyandé[1], Ferdian Thung[2], Shaowei Wang[2], David Lo[2], Lingxiao Jiang[2] and Laurent Réveillère[1]
[1]*LaBRI, University of Bordeaux, France*
[2]*Singapore 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 probalistic 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, thos 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 programs[1], 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

[1]ReLink was originally assessed on 3 programs

manual check to verify that links in the dataset are sound and complete. These links are inserted by the actual developers during their actual bug-fixing activities over a long period of time. The benefit of using such a dataset is clear: we can get a large number of highly accurate bug links, and with this ground truth, we can propose various dimensions to test the effectiveness of bug linking tools.

In addition, there are many other widely-used information retrieval techniques besides the one used in ReLink, such as vector space modelling (VSM), latent semantic indexing (LSI), and latent Dirichlet allocation (LDA) (c.f. [16]); they have been shown to be effective for many software engineering tasks, such as software traceability [17]–[20]. Thus in this study, we adapt these existing IR techniques for bug linking and compare them against ReLink. These experiments are performed to gain insights on the effectiveness of the IR technique that was used in the bug linking tool.

We find that the ReLink tool offers good precision in recovering links that are actually correct, but is less effective in recovering missing links. We also found that training data has limited impact on the results of ReLink due to other filtering steps used by ReLink to consolidate its outputs. Cross-project validation has shown that, overall, training data cannot be borrowed across different projects. Finally, our comparison between ReLink and other standard Information Retrieval solutions shows that there is still room for improving the effectiveness of ReLink.

The contributions of this work are as follows:

1) We provide a benchmark dataset[2] of known true links that could be used to evaluate bug linking work, and discuss a number of research questions for assessing the effectiveness of a bug linking tool.
2) We evaluate the effectiveness of ReLink, a recently proposed bug linking tool, on the benchmark.
3) We compare the effectiveness of ReLink versus standard information retrieval approaches that have been used in prior studies on software traceability.
4) We qualitatively characterize the kinds of links that ReLink misses and others that ReLink wrongly assigns.

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.

a) Excerpt of commit change log

Revision: r363 **Author**: srowen **Date**: Apr 15, 2008
Log message: Fix **Issue** 50 , not building on Windows, by adding some small workarounds for Windows paths in build files for Android

b) Excerpt of issue report

Issue 50: Building Android project problems
Status: Fixed **Owner**:srowen **Closed**: Apr 2008
Reported by his...@gmail.com, Mar 30, 2008
When building a BarcodeReader.apk on Windows, I got the below error...

I attached a diff to fix the problem for android/build.xml.
Please, check and merge it if you think it is useful.

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 project[3]. 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-

[2]The benchmark is available at http://momentum.labri.fr/bugLinking/

[3]https://code.google.com/p/zxing/

a) Excerpt of commit change log

Revision: r154 **Author**: srowen **Date**: Jan 22, 2008
Log message: Explictly add Yes/No commands to "`Open xxx`" dialog to ensure that both options show on all platforms

b) Excerpt of issue report

Issue 20: "`Open xxx`" dialog has only "`Cancel`" option
Status: Fixed **Owner**:srowen **Closed**: Feb 2008
Reported by project member srowen, Jan 22, 2008
Looks like the way the app works now, the "`OK`" button in dialogs like "`Open URL?`" does not show up on some phones. That is bad.

Comment 1 by horvath....@gmail.com, Jan 23, 2008
Hi! I have Nokia N61i and I am in the same situation! ...

Comment 2 by project member srowen, Jan 23, 2008
This is fixed in subversion...

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 Retrieval 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) metric [22] to compute the weight of each unique term.

Algorithm 1: The ReLink algorithm

```
Input: L // set of all possibilities of links
L_r ← ∅ // Links found by ReLink;
L_e ← mineLinksUsingPreviousHeuristics();
(T_t, S_t) ← determineTimeAndSimilarityThresholds(L_e);
M_bc_cc ← determineCommenterAndCommitterMappings(L_e);
foreach link l in L − L_e do
    if (bugCommenter(l), changeCommitter(l)) ∈ M_bc_cc then
        if ∃t · t ← bugCommentTime(l) | satisfiesThreshold(t, T_t)=True
        then
            l_b ← bug report;
            l_c ← change log;
            Sim_l ← computeTextSimilarity(l_b, l_c);
            if satisfiesThreshold(Sim_l, S_t) then
                L_r ← {L_r, l};
return L_r + 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 experiments. 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 Foundation[4]. 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 JIRA[5], 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

[4]http://www.apache.org/
[5]https://www.atlassian.com/software/jira/

Table I
SOFTWARE SYSTEMS IN THE BENCHMARK DATASET

Program	Description	# bug reports	# labeled links
activemq	Message broker	1068	1560
felix	OSGi implementation	1660	2287
hadoop	Support for distributed computing	2451	4421
lucene	Search library	952	2139
mahout	Machine learning library	244	327
opennlp	Machine learning toolkit	100	127
stdcxx	C++ standard library	398	571
struts	Web application framework	83	92
xalan	C++ XSLT processor	139	157
xerces	C++ API for XML parsing	178	200

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 just a part of a release version number.

Repository:	**Revision**	**Date**	**User**	**Message**
ASF	#151795	Mon Feb 07	dnaber	increase version number from 1.5 to 1.9, so that the jar is called called **LUCENE**−1.9−rc1−dev.jar

Figure 3. Wrongly labeled link

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/

Table II
10-FOLD CROSS VALIDATION RESULTS

	activemq	felix	hadoop	lucene	mahout	opennlp	stdcxx	struts	xalan	xerces
Average # of test links	203	321	67	66	38	15	119	10	16	23
Precision	0.922	0.821	0.712	0.7	0.927	0.4	0.007	1.0	0.6	0.975
Recall	0.181	0.17	0.101	0.088	0.164	0.027	0.029	0.207	0.038	0.188
F-measure	0.302	0.28	0.176	0.155	0.276	0.051	0.011	0.331	0.072	0.307

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 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? Thouroughly 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 insighs 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.

$$Precision = \frac{|\{labeled\ links\} \cap \{link\ inferred\ by\ tool\}|}{|links\ inferred\ by\ tools|} \quad (1)$$

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

provides the formulation for its computation.

$$Recall = \frac{|\{labeled\ links\} \cap \{links\ inferred\ by\ tool\}|}{|labeled\ links|} \quad (2)$$

- Finally, we compute the **F-measure**, the harmonic mean between Recall and Precision. We consider that both Precision and Recall are equally important and thus, they are equally weighted in the computation of F-measure in Equation (3).

$$F - measure = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

V. ReLink Evaluation Results

In this section we report the evaluation of ReLink with our benchmark dataset and following the research questions previously outlined. The discussions are based on the metrics described above. For our evaluation process, we use the ReLink tool as a black box[8], only providing desired inputs and analyzing the outputs. We have downloaded the version of the tool that was available at the project web page[9] 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 *struts* 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.

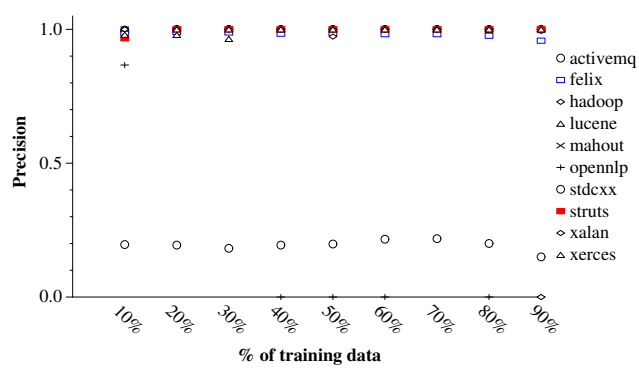

Figure 4. Precision – sensitivity to training data

The impact on recall rates is observable on datasets of the highest and lowest sizes. Figure 5 shows that recall is increasing for the *felix* program and overall is dropping for the *struts* program. These results suggest that smaller datasets, in which outliers are more noticeable, would affect the recall of ReLink.

Finally, in Figure 6, we notice that the F-measure follows the results of the Recall metrics. This was expected based on

[8]We have checked that ReLink contains rules for the patterns used in Apache systems for denoting bug IDs

[9]http://www.cse.ust.hk/~scc/ReLink.htm

Table III
EVALUATION OF LINK ACCURACY FOR PARTIALLY LINKED LINKS

% partial links used for training		activemq	felix	hadoop	lucene	mahout	opennlp	stdcxx	struts	xalan	xerces
25%	Precision	1	0.977	1	1	1	1	0.212	1	1	1
	Recall	0.114	0.092	0.133	0.036	0.016	0.03	0.058	0.103	0.03	0.261
	F-measure	0.204	0.169	0.235	0.07	0.032	0.059	0.091	0.188	0.059	0.414
50%	Precision	0	1	1	0	0	0	0.111	0	0	1
	Recall	0	0.008	0.015	0	0	0	0.023	0	0	0.16
	F-measure	0	0.015	0.03	0	0	0	0.038	0	0	0.276
75%	Precision	0	1	0	0	0	0	0.083	0	0	1
	Recall	0	0.008	0	0	0	0	0.016	0	0	0.095
	F-measure	0	0.015	0	0	0	0	0.027	0	0	0.174

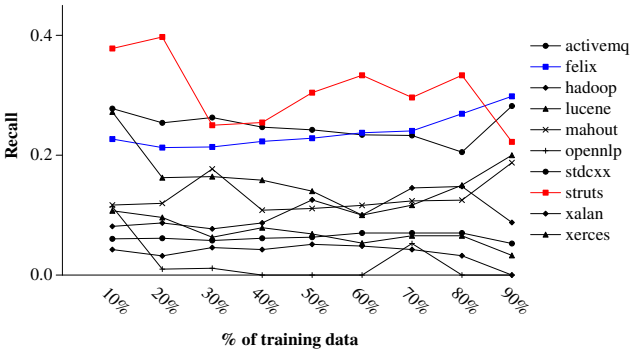

Figure 5. Recall – sensitivity to training data

the results of Precision which did not appear to be affected by the variation in proportion of training data.

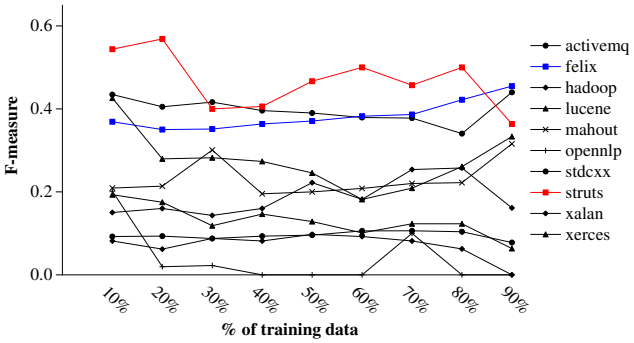

Figure 6. F-measure – sensitivity to training data

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.

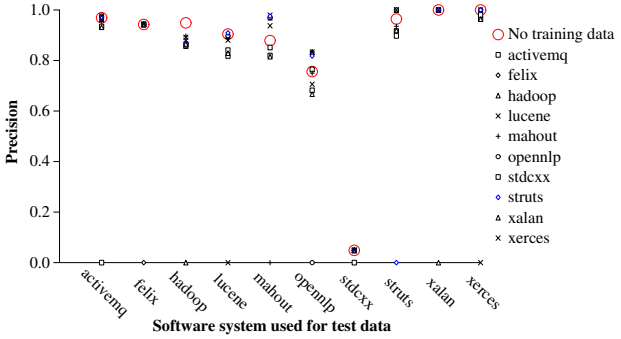

Figure 7. Precision – Cross project evaluation

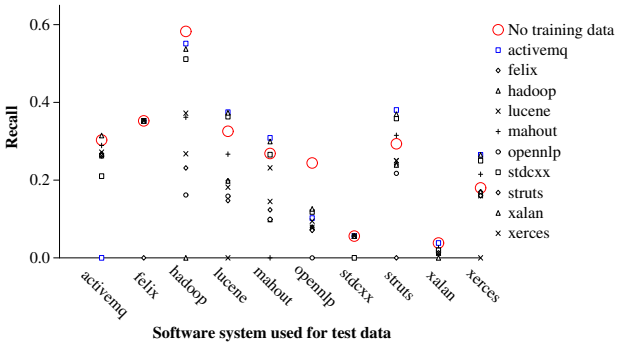

Figure 8. Recall – Cross project evaluation

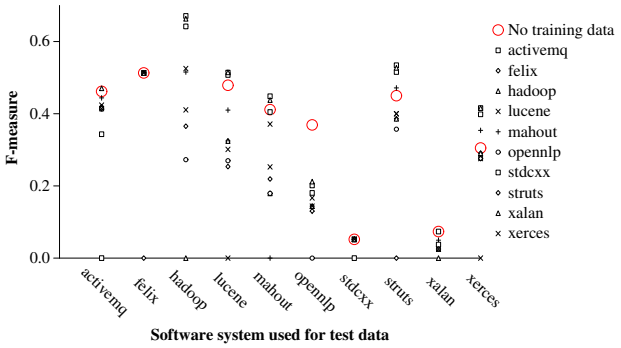

Figure 9. F-measure – Cross project evaluation

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

Table IV
COMPARISON OF RELINK WITH EXISTING IR TECHNIQUES: VECTOR SPACE MODELING (VSM), LATENT SEMANTIC ANALYSIS (LSA), AND LATENT DIRICHLET ALLOCATION (LDA)

		activemq	**felix**	**hadoop**	**lucene**	**mahout**	**opennlp**	**stdcxx**	**struts**	**xalan**	**xerces**
Precision	ReLink	0.922	0.821	0.712	0.700	0.927	0.400	0.007	1.0	0.600	0.975
	VSM	0.068	0.148	0.303	0.137	0.172	0.169	0.028	0.087	0.017	0.061
	LSA	0.035	0.095	0.025	0.077	0.069	0.160	0.010	0.101	0.017	0.046
	LDA	0.011	0.021	0.013	0.036	0.031	0.075	0.011	0.028	0.017	0.657
Recall	ReLink	0.181	0.17	0.101	0.088	0.164	0.027	0.029	0.207	0.038	0.188
	VSM	0.128	0.226	0.455	0.283	0.306	0.278	0.077	0.218	0.045	0.121
	LSA	0.116	0.175	0.125	0.151	0.135	0.163	0.066	0.185	0.062	0.084
	LDA	0.352	0.254	0.08	0.263	0.344	0.556	0.404	0.194	0.056	0.015
F-measure	ReLink	**0.302**	**0.28**	0.176	**0.155**	**0.276**	0.051	0.011	**0.331**	**0.072**	**0.307**
	VSM	0.068	0.155	**0.331**	0.152	0.199	**0.18**	**0.035**	0.106	0.02	0.068
	LSA	0.037	0.099	0.029	0.083	0.078	0.155	0.014	0.11	0.018	0.051
	LDA	0.011	0.024	0.018	0.04	0.037	0.071	0.015	0.028	0.018	0.015

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, opennlp, and stdcxx. LDA and LSA outperform Relink for: opennlp and stdcxx. 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

a) Excerpt of commit change log

revision: 682831 **author**: apetrelli **date**: 2008−08−05 17:50:09
msg: Applied patch provided by Yannick Haudry

b) Excerpt of issue report (JIRA)

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...

Figure 10. ReLink missed link – Unleveraged feature of links

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.

a) Excerpt of commit change log

revision: 1177597 **author**: joern **date**: 2011−09−30 11:10:28
msg: Replaced encoding lookup with UTF−8 encoding, and removed restriction on specific language codes

b) Excerpt of issue report (JIRA)

key_name: OPENNLP−305
created: Fri, 30 Sep 2011 11:09:17 **assignee_name**:Joern Kottman
resolved: Mon, 31 Oct 2011 23:49:31 **resolution**: Fixed
summary: Update leipzig format parsing code to work with their latest release
description: The Leipzig project added more content and changed the encoding and language codes. [...] UTF−8 [...]

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

```
key_name: STDCXX−11
created: Thu, 4 Aug 2005 11:09:02 +0000 assignee_name : Unassigned
resolved: Fri, 5 Aug 2005 10:45:55 +0000 resolution: Fixed
summary: IA64 32−bit atomic operations broken
description: The atomic operations on IA64 are broken in 32−bit mode:
        $ cat t.cpp && nice gmake SRCS=t.cpp
        #include <iostream>
        int main () { }
        aCC −c −D_RWSTDDEBUG −mt −D_RWSTD_USE_CONFIG [...]
        $ gdb −q t
        (gdb) r
        Starting program: /build/sebor/aCC−5.57−15s/examples/t [...]
        Program received signal SIGSEGV, Segmentation fault [...]
          si_code: 1 − SEGV_MAPERR − Address not mapped to object.
        0x4118da0:1 in __rw_atomic_add32+0x1 ()
        (gdb) where
        #0  0x4118da0:1 in __rw_atomic_add32+0x1 ()
        #1  0x4070880:0 in _<em>rw::</em><em>rw_atomic_[...]
```

Figure 12. Bug report involved in several ReLink extraneous links

of different heuristics through a number of features of links. Nonetheless, we have found that the *stdcxx* program involves datasets that make ReLink's precision drop significantly. Manual investigation of the false positives for this program has exposed two bug reports, namely `STDCXX-11` and `STDCXX-8`, for which ReLink mistakenly finds links to several commits. Figure 12 shows an excerpt of `STDCXX-11`.

In the bug reports extraneously linked we observe that the bug reporter has directly dumped his code, his compilation command lines and even gdb output. The generality of the terms appearing in the text may have lead ReLink to wrongly assign many links to the bug. Though developers expect users to provide crash information to easily reproduce the bug, such data can be provided as part of an attachment. Bug reporters however may not follow developer instructions. One research roadmap for improving ReLink could therefore consist of a more thorough analysis of bug reports to separately process source code data and to also exclude user command-line information which may be out of scope.

H. Threats to Validity

Our empirical evaluation bears some threats to both internal and external validity. The main threats to internal validity are related to the process of building the benchmark dataseti. We have tried to minimize this threat by our assessment of the soundness and completeness of the links extracted from the JIRA-based issue tracking system.

Threats to external validity refers to the generalizability of our findings. We minimize this threat by considering a wide range of projects of various domains written in different programming languages.

VI. Related Work

We highlight in the following subsections a number of related studies on evaluation framework and bug linking.

A. Bug Linking

There have been a number of studies that propose various techniques to either identify bug reports or link bug reports to the revisions that fix them. Antoniol et al. propose a technique to classify if a change request is a request for enhancement or a bug report [25]. Tian et al. propose a technique that detects if a revision is a bug fixing revision or not [26]. Bird et al. propose a technique named LINKSTER that could be used to aid developers in linking bug reports to the revisions that fix them [14]. Their approach is semi-automated.

Sureka *et al.* have used a probabilistic approach based on the Fellegi-Model for traceability link to recover bug links [12]. Wu *et al.* have recently proposed ReLink which is an information-retrieval based technique [15].

In this study, we evaluate the effectiveness of ReLink in several dimensions and compare it with existing work on information retrieval that has been applied to software traceability studies [17]–[19].

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. evaluates 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

[1] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *ESEC/FSE*, 2011.

[2] A. Mockus, R. T. Fielding, and J. D. Herbsleb, "Two case studies of open source software development: Apache and mozilla," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 3, pp. 309–346, 2002.

[3] T. Zimmermann, R. Premraj, and A. Zeller, "Predicting defects for eclipse," in *PROMISE*, 2007.

[4] A. Bernstein, J. Ekanayake, and M. Pinzger, "Improving defect prediction using temporal features and non linear models," in *IWPSE*, 2007, pp. 11–18.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Trans. Softw. Eng.*, vol. 31, no. 4, pp. 340–355, 2005.

[6] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *MSR*, 2005, pp. 1–5.

[7] A. Hindle, D. M. German, and R. Holt, "What do large commits tell us?: a taxonomical study of large commits," in *MSR*, 2008.

[8] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *ASE*, 2006.

[9] M. Fischer, M. Pinzger, and H. Gall, "Populating a release history database from version control and bug tracking systems," in *ICSM*, 2003, pp. 23–32.

[10] H. Zhang, "An investigation of the relationships between lines of code and defects," in *ICSM*, 2009, pp. 274 –283.

[11] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller, "If your bug database could talk..." in *ISESE*, 2006, pp. 18–20.

[12] A. Sureka, S. Lal, and L. Agarwal, "Applying fellegi-sunter (fs) model for traceability link recovery between bug databases and version archives," in *APSEC*, 2011, pp. 146 –153.

[13] A. Bachmann and A. Bernstein, "Data retrieval, processing and linking for software process analysis," *Technical Report IFI-2009.0003, Dept. of Informatics, University of Zurich*, May 2009.

[14] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *FSE*, 2010.

[15] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "ReLink: Recovering links between bugs and changes," in *FSE*, 2011, pp. 15–25.

[16] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[17] G. Antoniol, G. Canfora, G. Casazza, and A. D. Lucia, "Information retrieval models for recovering traceability links between code and documentation," in *ICSM*, 2000.

[18] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *ICSE*, 2003.

[19] H. U. Asuncion, A. U. Asuncion, and R. N. Taylor, "Software traceability with topic modeling," in *ICSE (1)*, 2010, pp. 95–104.

[20] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on linux kernel," in *WCRE*, 2011, pp. 92–96.

[21] A. Murgia, G. Concas, M. Marchesi, and R. Tonelli, "A machine learning approach for text categorization of fixing-issue commits on cvs," in *ESEM , 2010*.

[22] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, 1st ed. Addison Wesley, May 1999.

[23] D. F. Bacon, Y. Chen, D. Parkes, and M. Rao, "A market-based approach to software evolution," in *OOPSLA*, 2009, pp. 973–980.

[24] R. R. Bouckaert, "Choosing between two learning algorithms based on calibrated tests," in *ICML03*, 2003, pp. 51–58.

[25] G. Antoniol, K. Ayari, M. D. Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement? a text-based approach to classify change requests," in *CASCON*, 2008.

[26] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," in *http://www.mysmu.edu/faculty/davidlo/drafts/severity.pdf*, 2012.

[27] D. Lo and S.-C. Khoo, "Quark: Empirical assessment of automaton-based specification miners," in *WCRE*, 2006.

[28] K. Bogdanov and N. Walkinshaw, "Computing the structural difference between state-based models," in *WCRE*, 2009.

[29] E. Engström, P. Runeson, and G. Wikstrand, "An empirical evaluation of regression testing based on fix-cache recommendations," in *ICST*, 2010.

[30] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.

[31] M. Pradel, P. Bichsel, and T. R. Gross, "A framework for the evaluation of specification miners based on finite state machines," in *ICSM*, 2010, pp. 1–10.

[32] E. Engström, M. Skoglund, and P. Runeson, "Empirical evaluations of regression test selection techniques: a systematic review," in *ESEM*, 2008, pp. 22–31.

[33] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *ICSE*, 1994, pp. 191–200.

[34] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault detection," in *ICSE*, Orlando, Florida, May. 2002, pp. 467–477.

[35] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.

[36] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.

[37] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-Based Multiple Fault Localization," in *ASE*, 2009.

[38] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.

[39] A. Lamkanfi, S. Demeyer, Q. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," in *CSMR*, 2011.