ReLink: Recovering Links between Bugs and Changes

Rongxin Wu[†], Hongyu Zhang[†], Sunghun Kim[§] and S.C. Cheung[§]

[†]School of Software, Tsinghua University
Beijing 100084, China
wrx09@mails.tsinghua.edu.cn, hongyu@tsinghua.edu.cn

[§]Department of Computer Science and Engineering
The Hong Kong University of Science and Technology, Hong Kong, China
{hunkim, scc}@cse.ust.hk

ABSTRACT

Software defect information, including links between bugs and committed changes, plays an important role in software maintenance such as measuring quality and predicting defects. Usually, the links are automatically mined from change logs and bug reports using heuristics such as searching for specific keywords and bug IDs in change logs. However, the accuracy of these heuristics depends on the quality of change logs. Bird et al. found that there are many missing links due to the absence of bug references in change logs. They also found that the missing links lead to biased defect information, and it affects defect prediction performance.

We manually inspected the explicit links, which have explicit bug IDs in change logs and observed that the links exhibit certain features. Based on our observation, we developed an automatic link recovery algorithm, ReLink, which automatically learns criteria of features from explicit links to recover missing links. We applied ReLink to three open source projects. ReLink reliably identified links with 89% precision and 78% recall on average, while the traditional heuristics alone achieve 91% precision and 64% recall. We also evaluated the impact of recovered links on software maintainability measurement and defect prediction, and found the results of ReLink yields significantly better accuracy than those of traditional heuristics.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement–*Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics*

General Terms

Bias, Measurement, Experimentation

Keywords

Mining software repository, missing links, data quality, bugs, changes.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ESEC/FSE'11, September 5–9, 2011, Szeged, Hungary.
Copyright 2011 ACM 978-1-4503-0443-6/11/09...$10.00.

1. INTRODUCTION

Software defect information, including links between bug reports in bug tracking system and committed changes in source code repository, is the key information for software maintenance such as measuring software quality and predicting defects. It is possible to understand software maintenance efforts based on quality metrics derived from the links between bugs and changes, such as the percentage of buggy files [22, 33]. Defect information is also used to train models for defect prediction [17, 18, 33, 35, 36].

To collect links between bugs and changes automatically, many researchers mine bug reports in bug tracking systems and change logs in version archives. Heuristics traditionally used include searching for keywords (such as "Fixed" or "Bug") and bug IDs (such as "#42233") [5, 21, 27, 28, 33, 34] in change logs.

Recent studies revealed that these heuristics likely yield biased defect data, since they primarily rely on the comments in change logs [6, 7, 25]. Developers often maintain high quality change logs, but it is possible that they omit bug references in change logs. For example, when a developer fixes a bug in a revision, she may not document the fixed bug ID in the change log. Then, traditional heuristics miss the links between the bug and the change. As a result, defect information collected by traditional heuristics includes bias, especially lots of false negatives – missing links. Bird and Bachmann et al. confirmed this problem and reported that 54% of fixed bugs in the bug database are not linked to change logs [6, 7].

Unfortunately, these biased defect data affect software quality measurement and defect prediction performance. Bird et al. found that the BugCache algorithm [17] is sensitive to the biased data [7]. Kim et al. found that the change classification algorithm [16] is also sensitive to the biased data [18] when the number of instances is small. It is desirable to collect more accurate defect information by recovering the missing links.

To explore possibilities of recovering the missing links automatically, we conducted a qualitative study to identify characteristics of explicit links based on the bug IDs in change logs. We found that the links between bugs and changes exhibit certain features. For example, the bug-fixing time is close to the change-commit time, the change logs and the bug reports share textual similarity, and the developers responsible for a bug are typically the committers of the bug-fixing change.

Based on these findings, we propose an automatic link recovery algorithm, ReLink. Relink automatically learns satisfaction criteria of features from explicit links, and by applying the learned

criteria it checks if the features of an unknown link satisfy the criteria. If the unknown link satisfies all the feature criteria, it is considered a valid link.

We have applied ReLink to three open source projects (ZXing, OpenIntents, and Apache) and two simulation studies (on Apache and Eclipse MAT). We have evaluated the recovered links by comparing them to the ground truth links, which are manually recovered and verified links. Our experimental results are promising: on average, for the three open source projects, ReLink recovered links with 78% recall and 89% precision, while the traditional heuristics can only identify links with 64% recall and 91% precision.

We have also evaluated the practical impact of defect information obtained by ReLink. We measured the effects of ReLink on several software maintainability metrics. We also built defect prediction models using defect information with/without ReLink. Our experimental results indicate ReLink has nontrivial positive impacts on the maintenance studies.

In summary, our paper makes the following contributions:

- We propose ReLink, an automatic link recovery algorithm. We also report our experimental evaluation of ReLink.
- We report an empirical study on measuring the impact of recovered links on maintenance studies, especially on software defect prediction.

The remainder of this paper is organized as follows. In Section 2, we describe the traditional heuristics for mining links and their main challenges. In Section 3, we present ReLink, the proposed approach to mining links. We evaluate the performance of ReLink in Section 4 and its practical effects on software maintenance studies in Section 5. Sections 6 and 7 discuss the threats to validity and the related work respectively. We conclude the paper in Section 8.

2. MINING LINKS: TRADITIONAL HEURISTICS

This section reviews the use of traditional heuristics to mine the links between bugs and changes. In addition, we evaluate the quality of links mined with traditional heuristics, and discuss the involved challenges.

2.1 Traditional Heuristics

During software maintenance, it is common that changes and bugs are recorded in version control systems (such as CVS and SVN) and bug tracking systems (such as BugZilla), respectively. Developers often maintain change logs describing what they have changed and what bugs they have dealt with.

Traditional heuristics to identify bug-fixing changes rely on the premise that *developers leave hints or links about bug fixes in the change logs*. They look for specific keywords such as 'fixed', 'bug', and for information linking to bugs such as bug ID references in change logs. These traditional heuristics are widely used to mark bug fixes, identify bug-introducing changes, and build defect prediction models [5, 14, 16, 21, 27, 28, 33, 34].

However, the results of traditional heuristics largely rely on the change log quality. Recently, Bird et al. [7] noticed that, only a fraction of bug fixes are labeled in change logs explicitly, and this causes systematic biases in finding the links. The biases can significantly reduce the effectiveness of the follow-up studies. To overcome this challenge, Bird et al. [8] developed a manual link recovery tool, LINKSTER. The manually recovered links might be much more accurate. However, the practice does not scale because it requires significant manual effort.

2.2 Evaluation of Traditional Heuristics

To evaluate traditional heuristics and measure the number of missing links, we performed a replication study of Bird et al.'s [7] on two Android open source projects. In this study, we adopted the traditional heuristics proposed by Bachmann and Bernstein [5]:

1) Scan through the change logs for bug IDs in a given format (e.g. "issue 681", "bug 239" and so on).
2) Exclude all false-positive bug numbers (e.g. "r420", "2009-05-07 10:47:39 -0400" and so on).
3) Check if there are other potential bug number formats or false positive number formats, add the new formats and scan the logs iteratively.
4) Check if potential bug numbers exist in the bug-tracking database with their status marked as fixed.

Based on these heuristics we mined the links between change logs and bug reports. Then, we measured the number of fixed bugs successfully linked to change logs, as well as the number of links identified.

The two projects, ZXing[1] and OpenIntents[2], which we studied are highly active Android projects. ZXing is a barcode image-processing library for Android-based phones, and OpenIntents is an open intents library. Table 1 shows the results of the replication study. For ZXing, only 40.7% (55 out of 135) of the fixed bugs were found linked to change logs and 48.2% of the links were identified. Similarly, only 54 out of 101 bugs were linked to change logs for OpenIntents and 67.4% links were identified. Our results are consistent with Bird et al.'s results for Apache project [6]. They showed that 256 out of 559 (46%) bugs were linked and there were many missing links. In the next section, we briefly discuss the reasons of having missing links and the challenges of using traditional heuristics.

Table 1: Linking results of the traditional heuristics

Project	Revi-sions	# Fixed Bugs	% Linked Bugs	# Links	% Links Identified
ZXing	1-1694	135	40.7%	143	48.2%
OpenIntents	1-2890	101	53.5%	129	67.4%

2.3 Challenges

As shown in Table 1, there is a large number of bugs that cannot be linked to the committed changes by traditional heuristics. However, the results do not necessarily mean these bugs are not associated with any committed changes. We manually investigated the unlinked bugs and identified the following reasons that cause missing links:

Missing bug reference in change logs

Since leaving explicit bug reference in change logs is optional to developers, it is possible that developers do not write related bug IDs after they have fixed bugs. In this case, traditional heuristics fail to identify links between change logs and bugs. For example,

[1] http://code.google.com/p/zxing/
[2] http://www.openintents.org/

consider the change log[3] and the bug report[4] in Figure 1, which are taken from the ZXing project. There are no bug ID references in the change log therefore traditional heuristics cannot identify which bug the revision 148 has fixed.

However, our manual examination found that this change log can be actually linked to a bug report: we searched through all bug reports and found that the bug report #18 contained similar problem descriptions as the change log. Also, we noticed that there was a bug-fixing comment made for bug #18 on Jan 22, 2008, by the developer "srowen@gmail.com". The revision #148 was also committed on Jan 22, 2008 by the author "srowen". Therefore, it is likely that the revision #148 is linked to bug #18 (we also checked the source code and confirmed this link). The commonalities give an indicator of the link between the bug and the change. They provide clues to recover the missing link.

Change Log (Revision: 148; Author: srowen; Date: Jan 22, 2008)
Name of midlet is "ZXingMIDlet", not "ZXingMidlet"!

Bug Report (Issue 18; Status: Fixed)**:**
Reported by herf...@yahoo.com, Jan 11, 2008
Issue 18: does not work on nokia 5300

Comment 1 by project member srowen@gmail.com, Jan 11, 2008
Which version, regular or basic? …
…
Comment 3 by herf...@yahoo.com, Jan 12, 2008
I tested both of them (regular and basic) but in both case it just make an exception…
…
Comment 5 by project member srowen@gmail.com, Jan 22, 2008
…The class is called "ZXingMIDlet" but the exception mentions "ZXingMidlet" (note different capitalization). It looks like the manifest file I wrote gets this wrong. …

Comment 6 by project member srowen@gmail.com, Jan 22, 2008
I think I have fixed this particular problem by correcting…
Can you try the most recent version from…

Figure 1: An example of missing links

Irregular bug reference formats

The traditional heuristics, such as Bachmann and Bernstein's approach [5], search for patterns in change logs using regular expressions (e.g. "issue 681", "bug 239" and so on). However, our manual examination found that developers used many different ways to give bug references in change logs, such as "solve problem 681", "Fixed for #239", "see #149", "for ~~Issue 143~~" ("Issue 143" is strikethrough and marked-up in HTML), etc. Developers may also occasionally make typos such as "fic 239" [23]. The format may even vary across change logs within the same project, as different open source contributors may have their own preferences. It is not easy to develop a single, generic tool to support all possible bug reference formats.

The above analysis motivates us to develop a new link recovery algorithm. In our approach, we identify features of links between bugs and changes, and apply them to recover the missing links.

[3] http://code.google.com/p/zxing/source/detail?r=148

[4] http://code.google.com/p/zxing/issues/detail?id=18

3. RELINK: RECOVERING LINKS BETWEEN BUGS AND CHANGES

In this section, we present ReLink, a new approach to recovering the links between bugs and changes.

3.1 Features of Links

Our approach is based on the identification of features of the links between bugs and changes. First, we identify explicit links, which are links that can be discovered by traditional heuristics. We then analyze the features of these links. Through this analysis, we have identified the following features, which can be later used to recover the missing links:

Time Interval: This is the interval between the time (t_f) when a bug was fixed as given by its bug report and the time (t_c) when the corresponding bug fix was committed at the code repository. The interval ($t_f - t_c$) is a useful feature because it leverages the knowledge that a bug should be fixed after its creation and before its closure. After developers have committed the code change for a bug fix (at time t_c), they are obliged to update the bug report (at time t_f) that the bug has been fixed. Therefore, t_f should be greater than but close to t_c for a linked bug and its committed code change.

Bug Owner and Change Committer: For linked bugs and changes, there exists a mapping between the bug owner and the change committer. Ideally, the committer who makes the bug-fixing changes should be the person who is responsible for fixing the bug. It is also possible that the change committer and the bug owner are different persons, but the mapping between them could be identified by mining the software repositories (which will be discussed in Section 3.2.2).

Text Similarity: This is the textual similarity between bug reports and change logs. For linked bugs and changes, the natural language descriptions in the bug report are often similar to those in the change logs, as they may refer to the same issue and share similar keywords.

3.2 Mining Features of Links

3.2.1 The Interval between the Bug-fixing time and the Change-Commit Time

Although the interval between bug-fixing time and the commit time is a useful feature for mining links, determining the length of such an interval is a nontrivial task. We observed from our empirical studies that developers do not always change the status of bugs to "Fixed" in the bug tracking system immediately after they have committed the bug-fixing changes. For example, we investigated the explicit links of the ZXing project mined by traditional heuristics, and observed when developers change the bugs' status to "Fixed" after the bug-fixing change is committed. We found that for 20% of explicit links, their bug-fixing time and change-commit time differ by more than one day. The bug-fixing time and the change-commit time could be far apart. Our first research challenge is therefore to determine the actual bug-fixing time from each bug report.

With further investigations, we found that most bug comments are related to bug-fixing activities, as developers often post comments to the bug tracking system to report a bug fix and notify the bug reporter. Therefore, we could determine the bug-fixing time according to the time of comments in bug reports. For example, consider the change log and the bug report in Figure 1. The

revision #148 was committed on Jan 22, 2008. There was also a bug-fixing comment made for bug #18 on Jan 22, 2008, by the change committer ("srowen").

We also empirically verified the time interval feature using the explicit links. For ZXing and OpenIntents, on average, each bug received 2-3 comments. For more than 93% of the links, the intervals between bug-comment time and the change-commit time were less than 24 hours. For more than 96% of the links, the intervals were less than one week. The results indicate that in most cases, the change-commit time is close to a bug-comment time.

3.2.2 Mapping between Bug Owners and Change Committers

When bugs are assigned to a certain developer to fix, the developer information P_b is recorded by bug tracking systems. When bug-fixing changes are committed, the committer information P_c is recorded by version control systems. In our empirical studies, we compared the change committers and the bug report owners for linked bugs and changes. We found that they may not have the same names. This is because developers may use different login names in different scenarios. It is also possible that there is an appointed person in the team to confirm bug fixes and change bug status. Table 2 gives some examples for the identified mappings between committers and bug owners.

Table 2: Examples of the mapping between bug owners and change committers

Bug Owner P_b	Change Committer P_c	Project
dswitkin@gmail.com	dswitkin	ZXing
dswitkin@google.com	dswitkin@google.com	ZXing
srowen@gmail.com	srowen	ZXing
peli0101@googlemail.com	peli0101	Openintents
Will Rowe	wrowe	Apache
Erik Abele	erikabele	Apache

To identify the mappings between bug owners and change committers, we again examined the comments in bug reports. Our empirical studies found that developers often actively discuss bug-related issues and announce bug fixes via the bug tracking system. Therefore, it is likely that one of the commenters is the bug owner who is responsible for bug fixing. For example, in the ZXing project, for 99% of links that are found by traditional heuristics, the developers who committed bug-fixing changes posted bug comments in the bug tracking system. In the OpenIntents project, all developers who fixed bugs posted comments in the bug reports.

3.2.3 The Similarity between Bug Reports and Change Logs

Information Retrieval (IR) technology is commonly used to process textual documents in natural languages. In this project, we treat bug reports and change logs as texts and compare their similarities.

It is expected that for linked bugs and changes, bug reports and change logs exhibit certain similarity. To compute the similarity between bug reports and change logs, we first extract text features. Our approach adopts the Vector Space Model (VSM), a widely used model in IR technology [9]. In VSM, a document is represented as an n dimension vector $<w_1, w_2, w_3, \ldots w_n>$, where n is the number of distinct terms and w_i ($1 \le w_i \le n$) represents the weight of a unique term.

One of the most important issues in text processing is to select appropriate terms to represent the entire documents. In our case, the number of terms in bug reports and change logs could be large. To select the representative terms, we use the following steps to help reduce dimensions:

1) Remove stop words. Stop words are the words that have no strong meaning, such as "a", "an", "the" and so on.
2) Use one term to represent all other terms that have the same stemmer. For example, the tokens "fixing", "fixes", and "fixed" all share the same root "fix", thus we use "fix" to represent the others.
3) Use one term to represent all synonymous words. We apply the tool WordNet [12] to facilitate the selection of the synonymous words. For example, according to the definitions in the WordNet dictionary, the words "additional" and "extra" are synonymous words, therefore we can replace "additional" with "extra".

After selecting the terms in bug reports and change logs, we then calculate the weight for each term. In our approach, we use Term Frequency-Inverse Document Frequency (TFIDF) metric [9] to calculate the weight. The basic idea of TFIDF is that the weight w_i of a term in a document increases with its occurrence frequency in the specific document and decreases with its occurrence frequency in other documents. Formally:

$$w_i = tf_i \times idf_i \qquad (1)$$

where tf_i represents the occurrence frequency of the term t_i in the specific document, and idf_i represents the inverse document frequency, defined as:

$$idf_i = \log \frac{|D|}{|\{d : t_i \in d\}|} \qquad (2)$$

where $|D|$ represents the number of documents and $|\{d: t_i \in d\}|$ represents the number of documents that contain the i-th index term. We use the above formulas to extract features from all bug reports and change logs, and to compute the n dimension vector.

After obtaining the vector space model using TFIDF, we measure the similarity between a bug report and a change log using the Cosine similarity measure [9]:

$$Sim = \frac{\sum_{i=1}^{n} w_{1i} w_{2i}}{\sqrt{\sum_{i=1}^{n} w_{1i}^2 \times \sum_{i=1}^{n} w_{2i}^2}} \qquad (3)$$

It is expected that the linked bug reports and change logs share certain text similarity, thus the larger the Cosine similarity measure, the more a link is likely to be valid.

3.3 Learning Criteria of Features

In this section, we describe how we determine the thresholds of features so that these features can characterize most of the real links. Such thresholds can be treated as criteria for determining links – if the features of an unknown link satisfy the criteria, the link is likely to be valid. Otherwise it is irrelevant and should be removed.

Determining the criteria of features is nontrivial. For example, a lower threshold enables us to relate more fixed bugs and committed changes, but the links identified contain more false

positives. On the other hand, a higher threshold can reduce false positives, but less missing-links could be identified.

To determine the criteria of features, we learn from the explicit links that can be identified through traditional heuristics (*Le*). For the time interval feature and the text similarity feature, their values vary independently. When these two features are applied to *Le*, different values can result in selecting different sets of links and lead to different F-measures. We propose a search-based algorithm, which exhaustively searches for the optimal combination of these two values so that the maximum F-measure can be achieved. Figure 2 shows our algorithm for determining the threshold values of these two features.

DetermineThresholds (*Le*: links between bugs and changes identified by the traditional heuristics)

1 Assign the time interval T with a small initial value T_0
2 Assign the text similarity threshold S with a small initial value S_0
3 Select links in *Le* that satisfy T and S, and compute F-measure
4 Increase S by a small step *s1*
5 Repeat steps 3-4 until the maximum threshold S_m is reached
6 Increase T by a small step *s2*
7 Repeat steps 3-6 until T reaches the maximum threshold T_m
8 Choose the threshold values T_t and S_t that achieve the best F-measure (if ties exist, choose the first occurrence of T_t and S_t)
9 Return T_t and S_t

Figure 2: Determining the thresholds of features

In our experimentation, we specify T_0 =1, *s2*=1, and T_m = 30, which means that we try the time interval from 1 day to 30 days. We specify S_0=0, *s1*=0.01, and S_m = 1, which means that we try from text similarity 0 to 1, with steps of 0.01. In total, our algorithm executes at most 30*100 times, in which we search for the combination of S and T that maximizes F-measure. Based on the identified optimal values T_t and S_t, a criterion is formed: *a link is considered irrelevant if its time interval and text similarity values are above the thresholds*.

DetermineMappings (*Le*: links between bugs and changes identified by the traditional heuristics)

```
1  Initialize the set of mappings M=Φ
2  For each link l in Le
3      For each mapping m between l's change committer
         and bug commenter
4        If (m is not in M)
5          Add m to M
6        EndIf
7      EndFor
8  EndFor
9  Return M
```

Figure 3: Determining the mappings between bug owners and change committer

We also learn the mappings between bug owners and change committers from the explicit links identified by the traditional heuristics. We extract all relationships between a bug commenter and a change committer and form the mapping set. The algorithm is described in Figure 3. The criterion for this feature is: *a link is considered irrelevant if none of its bug commenters is mapped to its change committer*.

3.4 Recovering the Missing Links

A large and evolving project tends to contain a lot of fixed bugs and committed changes. This results in a large number of potential links between bugs and changes. Furthermore, there are also many types of relationships between bugs and changes (as illustrated in Figure 4):

- one to one: one bug could be fixed by one change;
- many to many: one bug could be fixed by multiple changes, and one change could fix multiple bugs
- no relationship: a change could be a non-bug fixing change, thus it does not link to any bug.

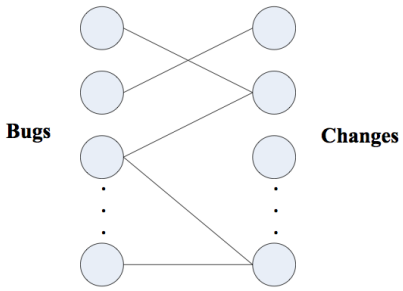

Figure 4: The relationships between bugs and changes

```
1  Store all possible links between bugs and links in L
2  Initialize the set Lr =Φ
3  Mine links Le between bugs and changes using
     the traditional heuristics
4  DetermineThresholds (Le)
5  DetermineMappings(Le)
6  For each link l in (L – Le)
7    If there is mapping between l's bug commenter and l's
       change committer
8      If any of l's bug comment time is within the time
         interval threshold Tt
9        If the text similarity between l's bug report and
           change log is within threshold St
10         add l to Lr
11       EndIf
12     EndIf
13   EndIf
14 EndFor
15 Return Lr + Le
```

Figure 5: The ReLink algorithm

To automatically identify the links between bugs and changes, we propose a link recovery approach, ReLink. ReLink is based on the identified features. The algorithm of ReLink is described in Figure

5: we first determine the satisfaction criteria of features by applying the algorithms described in Figures 2 and 3 (lines 3-5). The criteria include the time interval between the bug-fixing time and the change-commit time, the text similarity threshold, and the mappings between the bug owners and the change committers. ReLink automatically learns the criteria from the explicit links that can be identified by traditional heuristics. For each unknown link that cannot be identified by traditional heuristics (line 6), ReLink checks if the link satisfies all criteria (lines 7-9). If it satisfies, we consider it a valid link (line 10). After checking all the unknown links, ReLink returns the recovered set of links between bugs and changes (line 15). The overall process of ReLink is also illustrated in Figure 6.

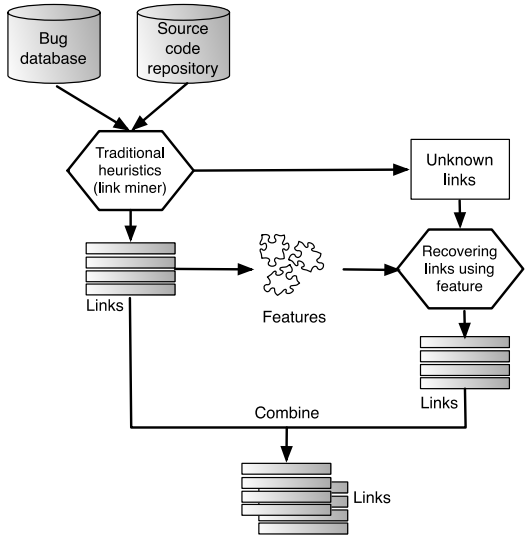

Figure 6: The overall process of ReLink

4. EVALUATION

This section presents evaluation results of ReLink.

4.1 Subject Projects and Experimental Setup

We investigated two projects as shown in Table 1. These two projects (ZXing and OpenIntents) are highly active Android projects hosted by Google Code. We collected their change logs from the SVN repository and bug reports from the Google Code issue tracking system. To obtain the ground truth (the "golden set") of the links, we manually read change logs, bug reports, and the corresponding code changes to establish links between changes and bug reports. The manual annotation was performed by two people: one identifying the links and the other verifying the results.

We also experimented with the Apache dataset that was provided by Bachmann and Bird et al. [6]. In the Apache dataset, the links between defects and bugs were manually annotated by an Apache core developer (Justin Erenkrantz) using the LINKSTER tool [6]. We treated this dataset as a "golden set" and used it to evaluate the performance of ReLink.

To further evaluate the performance of ReLink when the bug IDs are absent from change logs, we conducted a simulation study as follows: we first collect a high quality dataset with most of the links identified, and then intentionally remove bug IDs in 50% of the change logs. Traditional heuristics fail to identify the links associated with these changes due to the absence of the bug IDs. We evaluate the performance of ReLink on recovering these missing links. We choose the Apache project and the Eclipse Memory Analyzer (MAT)[5] project in this simulation experiment as they have a high percentage of linked bugs. To overcome the randomness introduced by the 50% sampling, we perform the simulation experiment 10 times and compute the average results.

To facilitate the experiments, we have developed a tool for ReLink. The tool automatically collects information from source code repository and bug tracking system, builds the links between bugs and changes, and outputs the identified links. It is developed in Java, and runs on Windows and Linux. It consists of more than 10K lines of code.

4.2 Evaluation Metrics

Our experiments can lead to four kinds of results: a link we identify is a true link (*TP*), a link we identify is not a true link (*FP*), a link we miss is a true link (*FN*), and a link we miss is not a true link (*TN*). We use Recall, Precision, and F-measure as evaluation metrics. The definitions of these metrics are as follows:

- Precision = $\frac{TP}{TP + FP}$

This metric indicates how accurate the experiment result is.

- Recall = $\frac{TP}{TP + FN}$

This metric indicates the coverage of the experiment result.

- F-measure = $\frac{2 \times Precision \times Recall}{Precision + Recall}$

This metric takes the precision and the recall into consideration. It is a combined metric. In this study, we adopt the F1 metric, which weights precision and recall equally [32].

4.3 Evaluation Results

4.3.1 Comparisons to the Golden Set

Table 3 shows the experimental results for the three projects we investigated.

The ZXing project contains 135 fixed bugs. For the traditional approach, we adopted the heuristics proposed by Bachmann and Bernstein [5] and identified links for 57 (42.2%) fixed bugs without any false positives. Our approach, ReLink, identified links for 95 (70.4%) fixed bugs with 10% false positives. In ZXing, there are 143 links between bugs and changes. ReLink can successfully identify 107 links among them, leading to a Recall of 74.8%, which is much higher than that achieved by the traditional heuristics. The precision of ReLink is 0.9, which is marginally lower than that of the traditional approach. In terms of the F-measure, ReLink also outperforms the traditional approach (0.820 vs. 0.651).

For OpenIntents project, the percentage of linked bugs found by ReLink is 4% more than that of the traditional heuristics. There are 129 links between bugs and changes in the project. ReLink successfully recovered 95 of them. Compared to the traditional heuristics, the Recall is improved by almost 10%. Both approaches can achieve high precisions, with ReLink resulting in a better overall F-measure.

[5] http://www.eclipse.org/mat/

Table 3: Evaluation results

Project	Period	Revisions	#Fixed Bugs	Approach	% Linked Bugs	Recovery Links		
						Precision	Recall	F-measure
ZXing	11/2007-12/2010	1-1694	135	Traditional	42.2%	1.0 (69/69)	0.482 (69/143)	0.651
				ReLink	70.4%	0.90 (107/118)	0.748 (107/143)	0.820
OpenIntents	12/2007-12/2010	1-2890	101	Traditional	69.3%	1.0 (87/87)	0.674 (87/129)	0.805
				ReLink	73.3%	1.0 (95/95)	0.731 (95/129)	0.847
Apache	11/2004-4/2008	76294-899841	686	Traditional	77.1%	0.746 (791/1060)	0.764 (791/1035)	0.755
				ReLink	89.8%	0.747 (904/1210)	0.873 (904/1035)	0.805
Apache Simulation	11/2004-4/2008	76294-899841	686	Traditional	38.2%	0.741	0.375	0.498
				ReLink	53.0%	0.682	0.523	0.592
Eclipse MAT Simulation	4/2008-2/2011	10-1070	108	Traditional	49.1%	1	0.418	0.582
				ReLink	96.3%	0.858	0.623	0.718

For Apache project, the golden set identified by the Apache developer contains 1035 links between bugs and changes. ReLink successfully recovered 904 links, while the traditional heuristics recovered only 791 links. ReLink improves the Precision as well, leading to a better F-measure (0.805).

The experimental results confirm that ReLink can achieve better overall performance than the traditional heuristics.

4.3.2 Simulation Study

For the Apache simulation study, traditional heuristics lead to an average recall of 0.375 due to the removal of bug IDs from 50% of change logs. ReLink can achieve a much higher recall of 0.523. This is because ReLink uses feature criteria to identify links. Overall, the traditional heuristics lead to F-measure 0.498, while ReLink achieves F-measure 0.592.

For the Eclipse MAT simulation study, traditional heuristics lead to a low recall of 0.418 due to the removal of bug IDs from 50% of change logs. ReLink can achieve a much higher recall of 0.623. Overall, the traditional heuristics lead to F-measure 0.582, while ReLink achieves F-measure 0.718, a significant improvement.

For the simulation studies, Table 3 only shows the average results of 10 simulations. We also performed the paired-sample t-test to check if the F-measures with ReLink were statistically significantly better than the traditional heuristics. The t-test results confirmed that ReLink can lead to better F-measures than the traditional heuristics, at significance level 0.01. All simulation results confirm that ReLink can recover links with reasonable accuracy.

4.4 Discussion

Traditional heuristics identify links for a certain percentage of bugs, but they fail to identify links for the bugs that have no bug ID references in committed changes. As our experimental results show, ReLink is more effective to recover the links between bugs and changes than the traditional heuristics. ReLink can significantly reduce false negatives (missing links) – Table 3 shows that the recall values of ReLink are 6%-26% higher than those of traditional heuristics. The higher recall values are achieved because ReLink can identify links even the bug IDs are missing from the change logs, or the bug reference formats are irregular. Unlike traditional heuristics, ReLink does not use regular expressions to search for links. Instead, it identifies features of links and checks the satisfiability of unknown links. The performance of ReLink (measured in terms of F-measure) is better than that of traditional methods for all studied projects.

Still ReLink may introduce false negatives. For example, Table 3 shows that ReLink missed 25.2% of links in ZXing and 26.9% links in OpenIntents. We found two major reasons for false negatives. One reason is that there are no similar keywords between bug reports and change logs for some links, causing very low text similarity. Thus ReLink discards these links as irrelevant. The other reason is that some bugs have long intervals between the change-commit time and the bug-fixing time. As a result, these bugs fail to satisfy the time interval threshold. Figure 7 shows such an example in OpenIntents. The change for fixing the bug 217 was committed on Aug 22, 2009. The status of the bug 217 was not updated to "Fixed" until Sep 22, 2009. Note that, all these missing links are not identifiable by traditional heuristics either.

Change Log (Revision: 2293; Author: rmceoin; Date: Aug 22, 2009)
OI Safe: large patch, added Search activty, completely redid autolock, added preference 'Lock on screen lock' with default of true.…

Bug Report (Issue 217; Status: fixed)**:**
What steps will reproduce the problem?
1: Unlock OI Safe…
2: Leave screen on PassEdit
…
After step 6, screen should auto-lock.

Comment 1 by project member rmceoin@gmail.com,Mar 10,2009
…Also fails with PassView.

Comment 2 by project member rmceoin@gmail.com,Sep 22,2009
…Status: Fixed

Figure 7: An example of false negative cases

Change Log (Revision: 99398; Author: trawick; Date: Apr 17, 2003)
merge this fix into 2.0.46-dev:\n\n … PR 18649
[Justin Erenkrantz, Jeff Trawick]

Bug Report (bug 18649; Status: closed)**:**

…enable-layout broken in 2.0.45. In 2.0.45, --enable-layout no longer works…

Figure 8: An example of false positive cases

Bachmann et al. [6] also discovered that, some bugs in Apache projects are actually "bugs incognito", i.e., these bugs were only discussed in developer mailing list and were not stored/tracked using bug track systems. Like traditional heuristics, ReLink cannot discover links for this type of bugs either. Mining mailing lists and recovering more links remain as future work.

ReLink may also introduce false positives. We manually examined these false positives in Table 3. We found that the false positives of ReLink in the Apache project overlapped largely with those identified by traditional heuristics. These links were incorrectly identified due to reasons such as merge of changes. Figure 8 shows an example of false positive case in Apache introduced by the merge of changes. Revision 99398 was actually a merge of the bug fix (PR 18649) found in version 2.0.45 into a new version (2.0.46).

There are ways to reduce ReLink's number of false positives. For example, we can modify the ReLink algorithm (before line 15 in Figure 5) to let it double-check the links *Le* mined by the traditional heuristics - if a link *l* in *Le* does not satisfy the feature criteria, ReLink considers *l* an irrelevant link and removes it. We have compared the performance of the revised ReLink (called ReLink-), ReLink and traditional heuristics for the Apache project. The results are shown in Figure 9. ReLink- can achieve better precision than the other two methods. However, the recall of ReLink- is lower because of the tradeoff between recall and precision. Overall, ReLink-'s F-measure is better than that of the traditional heuristics, but slightly worse than ReLink. Investigating techniques that can further reduce the number of false positives and at the same time improve the recall value will be our important future work.

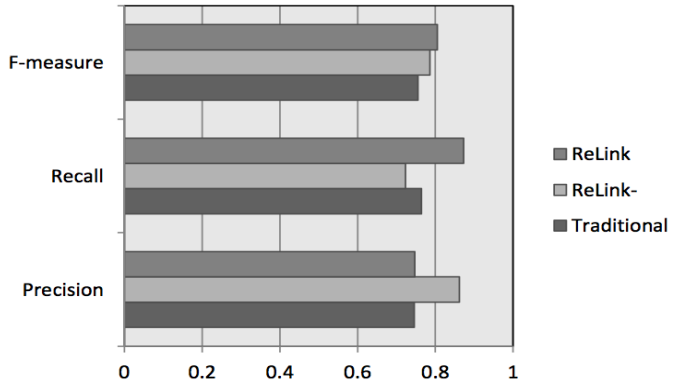

Figure 9: The performance of ReLink-

5. Practical Effects of ReLink

We have shown that ReLink can identify the links between bugs and changes more accurately than the traditional heuristics. The immediate next question would be the practical effects of the results. In this section, we show the implications of our results on software maintenance studies, especially on maintainability measurement and defect prediction.

5.1 Implications of Results on Maintainability Measurement

Many maintenance metrics have been proposed to understand maintenance activities and measure software maintainability. For example:

- The percentage of bug-fixing changes [15, 16, 21]: changes can be classified into many categories such as corrective changes (for bug fixing), adaptive changes (e.g., for accommodating new features), and perfective changes (e.g., for refactoring). Knowing the percentage of bug-fixing changes can help understand maintenance efforts spent on bug-fixing activities.
- The percentage of buggy files [33]: this is the percentage of defective files (files containing at least one bug). It can help measure software quality.
- Mean Time to Fix [22]: this is to measure the average time a team spends on fixing a bug. It can help measure a maintenance team's bug-fixing ability. The time spent for fixing a bug is calculated as the interval between bug report open time and bug-fixing change commit time.

Table 4. The comparisons of measurement data

Project	Linking approaches	%Bug-fixing changes	%Buggy files	Mean Time to Fix(days)
ZXing	Golden	8.1% (138/1694)	29.6% (118/399)	7.5
	Traditional	4.0% (67/1694)	14.8% (59/399)	10.2
	ReLink	6.3% (107/1694)	20.8% (83/399)	7.3
OpenIntents	Golden	4.2% 121/2890	4.9% (36/742)	25.1
	Traditional	2.9% (83/2890)	2.6% (19/742)	21.2
	ReLink	3.3% (94/2890)	4.0% (30/742)	25.1
Apache	Golden	2.3% (976/43167)	50.5% (98/194)	159.7
	Traditional	1.7% (753/43167)	47.9% (93/194)	178.9
	ReLink	2.0% (866/43167)	52.1% (101/194)	153.9

These metrics can be derived from the links between bugs and changes. After identifying the links, we know which bugs/changes are linked and which are not. Therefore we can collect the corresponding data and compute the metrics. As the quality of links collected by ReLink is higher than that collected by the traditional heuristics, we can obtain better measurement data for the above-mentioned metrics. Table 4 shows the measurement data we collected for the studied projects using the traditional heuristics and ReLink. We can see that the measures derived from links obtained by ReLink are closer to the actual ones, while the measures obtained by traditional heuristics contain larger discrepancies. For example, for ZXing, the actual percentage of bug-fixing changes is 8.1%. The measures derived from ReLink

and traditional heuristics are 6.3% and 4.0% respectively. The ZXing team's actual mean time to fix value is 7.5 days per bug. The measures derived from ReLink and traditional heuristics are 7.3 and 10.2 respectively. Clearly ReLink leads to more accurate measurement, thus it helps project teams better understand and plan their maintenance activities.

5.2 Defect Prediction with Relink

To measure the impact of ReLink on software defect prediction, we built commonly-used file level defect prediction models using defect data collected from traditional heuristics and ReLink, and measured the performance of the models to predict ground truth defects.

5.2.1 Data Collection

We collected metric data and defect data for the studied projects, and built a classification model to predict the defect-proneness of the files. The metric data includes file-level static code complexity measures collected by the *Understand for Java/Understand for C++* tool[6], such as lines of code, cyclomatic complexity, average lines of comments, etc. The defect data contains buggy instances (files) collected through analyzing the identified links between bugs and changes.

The *buggy* and *clean* labels are assigned based on links between change logs and bugs. If a commit is linked to a bug, we assume the files in the commit are buggy. To label files, we use the links obtained from three different approaches: traditional heuristics, ReLink, and the ground truth. Table 5 summarizes the datasets used in this experiment. The datasets contain different percentages of buggy files. For example, the ZXing project contains 399 files, among which 29.6% of the files are buggy in the "Golden" (ground truth) dataset, 14.8% files are buggy in the "Traditional" dataset, and 20.8% files are buggy in the "ReLink" dataset.

Table 5. Defect Prediction Dataset

Subject	Version	# of files	# of metrics	Linking approaches	% of buggy
ZXing	1.6	399	41	Traditional	14.8%
				ReLink	20.8%
				Golden	29.6%
Open-Intents	Revision 1088~2073	56	41	Traditional	10.7%
				ReLink	28.6%
				Golden	39.3%
Apache	2.0	194	60	Traditional	57.2%
				ReLink	46.9%
				Golden	50.5%

5.2.2 Prediction and Evaluation Models

After collecting the metric data and buggy labels, we built a classification model using a popular machine-learning algorithm, Decision Tree (J48 in the Weka implementation [31, 32]).

We adopted the 10-fold cross validation technique to train and evaluate the model. When the 9-fold is used as a training set, we used the labels from three datasets: Traditional, ReLink, and Golden. However, for the 1-fold (test set), we used the labels obtained from the Golden set, which are manually recovered ground truth links (Figure 10). Note that the ultimate goal of defect prediction is to identify ground truth (actual) defects.

Since the performance may vary based on instances in each fold, we ran this 10-fold cross validation model 100 times (100 10-fold), and computed the average performance. To measure the prediction model's performance, we used the standard measures such as Precision, Recall, and F-measure [32].

5.2.3 Results

Table 6 presents the prediction results using three different approaches. The prediction performance of using ReLink is much better than that of using traditional heuristics. For example, the F-measure with Traditional is 0.257 for OpenIntents, while the F-measure with ReLink is 0.706, a significant improvement. Similarly, for ZXing, the F-measure with ReLink is 0.325, while F-measure with Traditional is only 0.171. For Apache, ReLink also improved the prediction accuracy.

We have also performed the paired-sample t-test to check if the improvements in F-measures are statistically significant. The t-test results confirmed that they are statistically significant, at the 99% confidence level (p-value < 0.01).

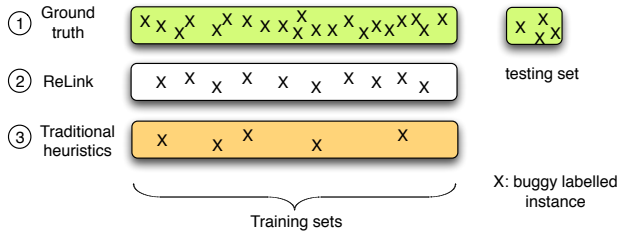

Figure 10: An example of false positive cases

Table 6. Prediction Results (Decision Tree)

Subject	Linking approaches	Precision	Recall	F-measure
ZXing	Traditional	0.346	0.114	0.171
	ReLink	0.432	0.261	0.325
	Golden	0.476	0.435	0.454
Open-Intents	Traditional	0.405	0.188	0.257
	ReLink	0.779	0.645	0.706
	Golden	0.742	0.683	0.711
Apache	Traditional	0.672	0.727	0.698
	ReLink	0.716	0.748	0.731
	Golden	0.709	0.713	0.711

These results clearly indicate that the defect information collected by ReLink improves the defect prediction performance. However, we noticed that the improvement varies across projects. For example, ReLink significantly improved the prediction accuracy of ZXing, while the improvement of Apache seems marginal. There are a few possible explanations for these results. First, as we showed in Table 3, the quality of Apache change logs is decent. As a result, traditional heuristics can mine links with reasonable accuracy. Basically, the quality of Apache defect information derived by traditional heuristics and that by ReLink do not have huge differences. Note that ReLink is still able to improve the prediction accuracy because of the more accurate defect information derived.

ReLink significantly improved the prediction performance of OpenIntents. As we showed in Table 3, ReLink is able to recover

[6] http://www.scitools.com/

many links missed by traditional heuristics. In addition, the number of instances of OpenIntents is small. As we also found in a recent study [18], prediction models for a subject with a small number of instances are more sensitive to noise. Since ReLink provides much accurate defect information, the prediction performance of OpenIntents is significantly improved.

6. THREATS TO VALIDITY

There are potential threats to the validity of our work:

- For the two Android projects, the golden sets of links were collected by us manually. To assure their quality, two people were involved: one annotating the links and the other verifying them. However, it is difficult to guarantee that the golden sets do not contain any false negatives or false positives. Even for the Apache dataset, which was manually examined by an Apache core developer, its quality is not completely assured because the developer may not recall all bug-fixing activities happened several months or years ago.
- Our approach is based on the assumption that the descriptions of bugs and committed changes are similar. In our investigation, we found that many open source projects hosted by Google Code follow the features we described. However, for projects that do not satisfy the assumption, our approach would be under threat.
- All datasets used in our experiments were collected from open source projects. We need to evaluate the performance of ReLink on commercial projects. This remains as future work.

7. RELATED WORK

Real-world data are often noisy, which may affect interpretations and models derived from the data. The data quality problem has also been observed by some software engineering researchers. For example, Mockus [20] noted that in many realistic scenarios the data quality is low (e.g., some change data could be missing), which could affect the outcome of an empirical study. Myrtveit et al. [24] and Strike et al. [29] also noticed the problem of missing and incomplete data in software effort estimation. In [1, 10, 11], authors analyzed the quality of bug reports and change logs. Liebchen and Sheppard [19] investigated the data quality issue and found that among hundreds of software engineering papers only four suggested that the data quality issue may affect their analysis results. In this paper, we focus on software process data and propose methods to improve data quality.

To improve the correctness of identified links between defects and changes, Fisher et al. [13, 14] discussed the confidence of links. They took file names specified in the bug tracking databases and change logs into account. If the files exist in both of them, the confidence of the links will be higher. Śliwerski et al. [28] proposed to verify the links by semantic analysis. A link is valid if it satisfies one of the conditions such as whether the bug has been resolved as Fixed at least once, whether the short description of the bug report is contained in the committed change log, etc. Bachmann et al. [5] improved Fischer's approach by excluding all false-positive numbers that have a defined format. They validated the linked bug report by checking the relationship between the bug-fixing date and the change commit date.

Although many researchers proposed various kinds of approaches to verify the links, the quality of the software process data is still not good enough. Bachmann and Bird et al. [6, 7] showed strong evidence that, only a fraction of bug fixes are explicitly labeled in the source code repository, and there exist systematic biases in finding the links. The bias could affect the effectiveness of the studies based on the process data such as software defect prediction. Nguyen et al. [25] also found that biases existed in a commercial project (IBM Jazz) that enforced strict development guidelines. In order to find the missing links, Bird et al. [8] developed the tool "LINKSTER" to facilitate identification of links manually. However, their tool requires manual effort, and thus is difficult to scale up.

Our work is also related to the traceability analysis among software artifacts. Many researchers have proposed information retrieval (IR) based techniques to recover traceability links between source code and text documents (such as development journals, error logs, and emails) [2, 3, 4]. Runeson et al. [26] and Wang et al. [30] applied IR techniques to compare text similarity between two bug reports, thereby identifying potentially duplicate bug reports. In our work, we apply IR techniques to compare text similarity between bug reports and change logs, thereby recovering traceability links between bugs and changes.

8. CONCLUSIONS AND FUTURE WORK

To automatically collect links between bugs and changes, traditional heuristics look for explicit links to bugs in change logs. Recent studies have shown that traditional heuristics could be biased, since developers may not leave explicit links in change logs. In this paper, we have proposed ReLink, an automatic link recovery algorithm. ReLink is based on automatically learned feature criteria from explicit links. Our experimental results have shown that ReLink is able to recover links reliably. We have also shown that the more accurate defect information collected by ReLink has positive impacts on the follow-up software maintenance studies, including defect prediction models.

Our experimental results confirm that defect information collected using traditional heuristics should be used with caution. It is desirable to use link recovery algorithms and to perform careful inspection of the collected defect information. ReLink is the first step toward this direction.

In future more research on recovering links is needed in order to obtain more accurate defect data. We will also apply our approach to industrial projects to evaluate its usefulness.

Our tool and the experimental data used in this paper are available at:

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

ACKNOWLEDGEMENTS

This research is supported by the Hong Kong RGC/GRF grant 612108, the NSFC grant 61073006, and the Tsinghua University research project 2010THZ0. We thank Christian Bird for providing us with the annotated Apache dataset, and Lichao Liu for validating the golden set data for Android projects.

REFERENCES

[1] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *ICSE'09*, pages 298–308, May 2009.

[2] G. Antoniol, G. Canfora, G. Casazza, A. De Lucia, and E. Merlo. 2002. Recovering Traceability Links between Code and Documentation, *IEEE Trans. Softw. Eng. 28*, 10 October 2002, 970-983.

[3] A. Bacchelli, M. D'Ambros, M. Lanza, R. Robbes, Benchmarking Lightweight Techniques to Link E-Mails and Source Code. In *WCRE'09*, Lille, France, pp. 205-214, Oct 2009.

[4] A. Bacchelli, M. Lanza, and R. Robbes, Linking e-mails and source code artifacts. In *ICSE '10*, Vol. 1. ACM, New York, NY, USA, 375-384.

[5] A. Bachmann and A. Bernstein. Software process data quality and characteristics - a historical view on open and closed source projects. In *IWPSE-Evol'09*, pages 119-128, Amsterdam, The Netherlands, August 2009.

[6] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, The Missing Links: Bugs and Bug-fix Commits. In *FSE'10*, 97-106, Santa Fe, New Mexico, USA, Nov 2010.

[7] C. Bird, A. Bachmann, E.Aune, J. Duffy, A. Bernstein, V.Filkov, and P. Devanbu, Fair and balanced?: bias in bug-fixing datasets. In *ESEC/FSE'09*, Aug. 2009, 121-130.

[8] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, LINKSTER: enabling efficient manual inspection and annotation of mined data. In *FSE'10*, 369-370, Santa Fe, New Mexico, USA, Nov 2010.

[9] R. Baeza-Yates and B. Ribeiro-Neto, *Modern Information Retrieval*, Addison Wesley, 1999.

[10] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim. Duplicate bug reports considered harmful... really? In *ICSM'08*, pages 337–345, October 2008.

[11] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller. Open-source change logs. *Emp. Softw. Eng*., 9(3):197–210, 2004.

[12] C. Fellbaum, *WordNet: An Electronic Lexical Database*, Cambridge, MA: MIT Press, 1998.

[13] M. Fischer, M. Pinzger, and H. Gall. Analyzing and relating bug report data for feature tracking. In *WCRE'03*, pages 90-99, Victoria, Canada, November 2003.

[14] M. Fischer, M. Pinzger, and H. C. Gall. Populating a release history database from version control and bug tracking systems. In *ICSM'03*, pages 23-32, Amsterdam, Netherlands, September 2003.

[15] A. Hindle, D. M. German, R. C. Holt: What do large commits tell us?: a taxonomical study of large commits. In *MSR 2008*, pp. 99-108, May 2008.

[16] S. Kim, T. Zimmermann, K. Pan and E. Whitehead Jr., Automatic Identification of Bug-Introducing Changes. In *ASE'06*, Tokyo, Japan, September 2006.

[17] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *ICSE'07*, pages 489–498, Washington, DC, USA, 2007.

[18] S. Kim, H. Zhang, R. Wu and L. Gong, Dealing with Noise in Defect Prediction. In *ICSE'11*, Honolulu, Hawaii, USA, May 2011.

[19] G. Liebchen and M. Shepperd. Data sets and data quality in software engineering. In *PROMISE'08*, 39–44, May 2008.

[20] A. Mockus, Missing Data in Software Engineering, *Empirical Methods in Software Engineering*. The MIT Press, 2000.

[21] A. Mockus and L. G. Votta, Identifying Reasons for Software Changes Using Historic Databases. In *ICSM 2000*, San Jose, CA, USA, 2000, pp. 120-130.

[22] 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*., 11(3):309–346, 2002.

[23] A. Murgia, G. Concas, M. Marchesi, R. Tonelli, A machine learning approach for text categorization of fixing-issue commits on CVS. In *ESEM 2010*, Bolzano-Bozen, Italy, Sep 2010.

[24] I. Myrtveit, E. Stensrud, and U. H. Olsson. Analyzing Data Sets with Missing Data: An Empirical Evaluation of Imputation Methods and Likelihood-Based Methods. *IEEE Trans. on Software Engineering,* 27(11), pp.999-1013, 2001.

[25] T. H. D. Nguyen, B. Adams, A. E. Hassan, A Case Study of Bias in Bug-Fix Datasets. In *WCRE'10*, pp. 259-268.

[26] P. Runeson, M. Alexanderson, O. Nyholm, Detection of Duplicate Defect Reports Using Natural Language Processing. In *ICSE'07,* 499-510, May 2007.

[27] A. Schroter, T. Zimmermann, R. Premraj, and A. Zeller. If your bug database could talk... In *ICSE'06*, pages 18–20, Rio de Janeiro, Brazil, September 2006.

[28] J. Sliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *MSR'05*, pages 24-28, Saint Louis, Missouri, USA, May 2005. ACM.

[29] K. Strike, K. E. Emam, and N. Madhavji. Software Cost Estimation with Incomplete Data. *IEEE Trans. on Software Engineering*, 27(10), pp.890-908, 2001.

[30] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, An approach to detecting duplicate bug reports using natural language and execution information. In *ICSE'08*, pages 461-470, Leipzig, Germany, 2008

[31] WEKA: http://www.cs.waikato.ac.nz/ml/weka/

[32] I.H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementation*, second ed., Morgan Kaufmann, 2005.

[33] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *PROMISE'07*, pages 1-9, Minneapolis, Minnesota, USA, May 2007.

[34] T. Zimmermann and P. Weissgerber. Preprocessing cvs data for Fine-grained analysis. In *MSR'04*, pages 2-6, Edinburgh, Scotland, UK, May 2004.

[35] H. Zhang and R. Wu, Sampling Program Quality, *Proc. ICSM 2010*, Timisoara, Romania, Sep 2010, pp. 1-10.

[36] H. Zhang, An Investigation of the Relationships between Lines of Code and Defects. In *ICSM'09*, Edmonton, Canada, September 2009, pp. 274-28.