Towards an Automation of the Traceability of Bugs from Development Logs – A Study based on Open Source Software

Bilyaminu Auwal Romo Andrea Capiluppi
Department of Computer Science
Brunel University
London, United Kingdom
{bilyaminu.auwal,andrea.capiluppi}@brunel.ac.uk

ABSTRACT

Context: Information and tracking of defects can be severely incomplete in almost every Open Source project, resulting in a reduced traceability of defects into the development logs (i.e., version control commit logs). In particular, defect data often appears not in sync when considering what developers logged as their actions. Synchronizing or completing the missing data of the bug repositories, with the logs detailing the actions of developers, would benefit various branches of empirical software engineering research: prediction of software faults, software reliability, traceability, software quality, effort and cost estimation, bug prediction and bug fixing.

Objective: To design a framework that automates the process of synchronizing and filling the gaps of the development logs and bug issue data for open source software projects.

Method: We instantiate the framework with a sample of OSS projects from GitHub, and by parsing, linking and filling the gaps found in their bug issue data, and development logs. UML diagrams show the relevant modules that will be used to merge, link and connect the bug issue data with the development data.

Results: Analysing a sample of over 300 OSS projects we observed that around 1/2 of bug-related data is present in either development logs or issue tracker logs: the rest of the data is missing from one or the other source. We designed an automated approach that fills the gaps of either source by making use of the available data, and we successfully mapped all the missing data of the analysed projects, when using one heuristics of annotating bugs. Other heuristics need to be investigated and implemented.

Conclusion: In this paper a framework to synchronise the development logs and bug data used in empirical software engineering was designed to automatically fill the missing parts of development logs and bugs of issue data.

EASE 2015, April 27-29, 2015, Nanjing, China.

http://dx.doi.org/10.1145/2745802.2745833

Categories and Subject Descriptors

D.2.8 [**Empirical Software Engineering**]: Metrics

Keywords

Bug traceability, Bug accuracy, bug-fixing commits,

1. INTRODUCTION

Over the past two decades, there has been a significant interest by software engineering researchers into the analysis and use of empirical data. Open source software (OSS) projects provide a large amount of process and product data, and several tools are available to mine and analyse this data. Utilising this vast amount of data can benefit both OSS and commercial projects: mining and analysing software artefacts, like code, design documents, requirements or bug issues, can offer fundamental contributions for empirical software engineering research.

In particular, bug tracking data can be used to design models for predicting software faults and software reliability; faults and reliability of a software artefact can also be linked to who, when and how changes were made to it. Similarly, the analysis of development logs (version control commit logs) can give important insights of the underlying software quality, by focusing on software developers, their actions and effort in order to build cost estimation models. Such logs, actions and effort can also be used for detecting bug fixing actions, improve bug prediction techniques and increase software quality and reliability[15].

The extraction of version control commit logs and of bugs issue data sets requires tools that (i) automatically mine software projects (or software repositories) and (ii) store the extracted data in specifically built databases, for posterior analysis. Data does not include only source code, but also meta-data, as logs, dates and types of actions performed on specific software artefacts. Developers in OSS communities use these tools as a medium of collaboration and communication, such as reporting bugs or mentioning changes that occur as a result of a bug fix, as well as revising all the commits to a software artefact [4].

Cross-analysing the two sets of bug-related data (version control commit logs and bug tracking data) is related to the traceability of bugs within software development. Such traceability is complicated by the fact that the sources of bug data might not be always in sync or complete [5]. Development logs should form a superset of all development data:

one would expect the data contained in issue trackers to be mirrored in the version control commit logs, and developers to record and distinguish their actions between development and bug fixing. In reality, inconsistency, incompleteness and skewed sets of data might be obtained when combining and cross-analysing the version control commit logs and the issue tracker data [2]. Missing or incomplete data can lead to a biased or untrustable analysis in empirical software engineering research [11].

This paper proposes a large empirical study that mines the development logs and bug issue data of over 300 OSS projects, hosted on GitHub[1]. The objective is to replicate an initial, exploratory analysis performed on one OSS project [10] with a larger dataset, and to quantify the issue of traceability of issue tracker bugs into development logs. In order to detect the bug fixing activity, we used a subset of the SZZ algorithm [12]; and two data mining tools, the Bicho issue tracker parser [6], and the CVSAnalY development log parser [9]. For the purpose of this research we are only interested in bug IDs that are being mentioned by developers: bug IDs do not necessarily need to be "fixed" or "resolved". Since the tools act on different data sets, they are run independently, produce independent results and fill-in logs and bugs of the same software project in different databases [7].

The second contribution of this paper is to propose a framework not only for "extracting", but also to automatically "syncing" development logs and bugs of issue data supporting multiple Bug Tracking Systems (BTS) and Version Control Systems (VCS). The novelty of the framework, apart from supporting various OSS software repositories, is the ability to synchronise missing development logs (concerning bugs) with data extracted from the BTS, and vice-versa. The aim of such a framework is to assist in mining the complete set of software evolutionary facts throughout the entire life cycle of software projects; to provide complete data to bug detection techniques; to assist the development during the corrective software maintenance; as well as to provide an unbiased dataset for empirical software engineering research on bugs.

The rest of the paper is organised as follows: in Section 2 we discuss the methodology and detail the steps performed to extract bugs and logs data, with a sample of 344 OSS projects from GitHub. In Section 3 we test the scalability of our approach and presents the results obtained within the sample. We then quantify the discrepancies between the development logs stored from the development logs and the bug issue data stored from the issue trackers. Afterwards, we discuss the results in Section 4. In Section 5 we introduce the structure of the framework, which defines and details the guidelines for implementation and the supported VCSs and BTSs, using the CVSAnalY and Bicho tools to store data in their respective databases. We integrate independently running components into the framework, and we identify the entities in Bicho and CVSAnalY databases that could be used to synchronize the bugs and logs of issue data into their respective databases. Also (In Section 5). We highlight the related work and the novelty of our results and framework in Section 6. Finally, Section 7) concludes.

2. METHODOLOGY

[1] `https://github.com/`

In the research reported in this paper, we partially implement the SZZ algorithm [12] to trace bugs and logs within the OSS sample obtained from GitHub. In our formulation, we only look for bugs described by the '#' sign and various numeric values (e.g., #1234), that are linked to the ID of a bug. In its original formulation, the SZZ algorithm also searches for keywords like 'Bug', 'Fixed' and others. A tool was developed to search for these IDs within the two databases, and to combine the results into intersection and union of sets.

The extraction of data and results was achieved in the steps detailed below:

1. Sampling an OSS forge: we extracted the projects' data from the GitHub repository through a crawler developed in Perl, obtaining the sample of OSS projects. The sizing of the sample was done considering a 95% of confidence level and 95% confidence interval, resulting in 344 projects. This randomizing process and selection steps are integrated in the tool that was developed for this research.

2. Obtaining the development logs: the tool is capable to interface with, and execute CVSAnalY and Bicho commands, in order to parse logs and bugs at once. CVSAnalY and Bicho automatically create databases and tables with meta-data, storing all the development logs (version control commit logs) and bug issue data of the sample. Among the tables generated by CVSAnalY, we then specifically query the *scmlog* table, which mentions the number and unique IDs of changes in the version control system. In the presence of a bug ID, the version control commit logs also mentions the bug ID with the #1234 format. For the purpose of this research we are only interested in bug IDs that are being mentioned by developers: bug IDs do not necessarily need to be "fixed" or "resolved". As above, this step is integrated in the tool that was developed for this research.

3. Obtaining the bug tracking data: the second phase in our data preparation process was to execute the Bicho tool to obtain and store all the information contained in the bug trackers of the projects contained in the sample, as well as all the issues reported by the users of a project and confirmed as such by developers. One of the tables created by Bicho is the *issues* and *issues_ext_bugzilla* table where the status ("open", "closed".) or the message accompanying the entry is stored and imported for publication by the relative GitHub tracker. In this way, we queried specifically the *issues_ext_bugzilla* table to obtain the set of unique number and IDs of bugs reported and confirmed by developers. Also this step is integrated in the tool that was developed for this research.

4. Data Cleaning: False Positives and True Positives: The fourth step was the cleaning step, before isolating the bug numbers and IDs for both CVSAnalY and Bicho. The query for the '#' sign followed by numeric values in the version control commit logs imported with CVSAnalY produces a large number of false positives in most of the sample of 344 OSS projects we obtained from GitHub. In this case, the messages refer to the pattern searched through the # sign, but

they are all linked to a request of pulling a merge from another distributed repository into the original one under GitHub. These were filtered out automatically and integrated in the tool that was developed for this research.

5. Isolating the bug numbers and IDs: we composed two sets of bug IDs, one from the development logs, and the other from the issue tracking systems. In the development logs, we looked for the bug IDs in the free text descriptions left by developers (and stored in the "scmlog" table). In the bug tracking data, we used the bug IDs as assigned by the developers to the issues reported as bugs. These steps are performed within the developed tool, by querying the appropriate tables and parsing and cleansing the results.

   In addition, we randomly pick few on the sample of 344 OSS projects obtained from GitHub and manually analysed each of the remaining bugs in Bicho and CVSAnalY databases, to make sure that each of the remaining IDs pointed to real bugs before evaluating the union and intersection of the sets. The bug IDs within the dataset obtained through Bicho are always related to bug IDs. In order to evaluate the bug data within the dataset extracted by CVSAnalY, we manually computed the precision and recall of our approach as follows:

$$Precision = \frac{how\ many\ \#s\ refer\ to\ bug}{how\ many\ \#s\ are\ present} \quad (1)$$

$$Recall = \frac{nr\ of\ bug\ activities\ in\ logs}{how\ many\ \#s\ are\ present} \quad (2)$$

   Using the formulas above, we manually found for example that for project ID 47 the precision was 100% and the recall was 67.57% . For project ID 52, we found a lower precision of 68.75% and a recall of 7.273%.

6. Evaluating the union and intersection of the sets: the final step was to evaluate the union and intersection of the sets, per project. Given a set of bug IDs mentioned in the development logs, and the list of bug IDs stored from the issue trackers of a project, we evaluated the intersection (i.e., the common bug IDs) of these two sets, as well as the union of such sets (i.e., the overall set of unique bug IDs jointly held in the two databases). We then formulated a metric (named *Shared Bug Coverage*) to describe how many bug IDs are common in the two databases. Also this final step is integrated into the tool.

The further step that will be integrated in the tool is to automatically populate the databases of bug tracking systems and development logs with missing bug information originated from either information source. We detail how we plan to integrate this further step in the section 5.3 below.

3. RESULTS

In this section we report the analysis on the sample of 344 OSS projects from GitHub. In particular, we report on how many bug IDs are mentioned in the two databases, per project. The two overarching hypotheses that we planned to verify in this research are:

1. bug-related data stored in the issue trackers should be considered as complete;

2. bug-related data is common and shared in both development logs and issue trackers.

In order to summarise the findings from the 344 OSS projects, we produced a box-plot to display the Shared Bug Coverage (SBC) ratio, defined as follows:

$$SBC = \frac{BugIDs(VCS) \cap BugIDs(BTS)}{BugIDs(VCS) \cup BugIDs(BTS)} \quad (3)$$

where Bug IDs(VCS) is the set of bug IDs as found in the development logs (of any project), and Bug IDs(BTS) is the set of bug IDs from the issue trackers (of the same project). This ratio was evaluated per project, and the values are always in the [0..1] interval. The box-plot is shown in Figure 1.

As visible from the box-plot, the set of common bug IDs is in general very low: in around 75% of the projects the common IDs (i.e., intersection of the sets) is no more than 20% of the overall number of detected bug IDs (i.e., the union of the sets). This could mean that in one of the two databases (either development logs or issue tracker data) there is most of the information on bug IDs, and that information is not mirrored in the other database. It could also mean that there is a common subset of bug IDs, but that most of the other IDs are not shared in the two informations sources. Below, we formulate 4 scenarios of bug coverage in the two databases, as we observed in the sample.

3.1 Scenarios of Bug Coverage

Depending on the configuration found for the two sets of bug IDs, four scenarios can be expected for a software project: they are depicted graphically in Figure 2. The number of projects in our sample that comply with such scenarios are further described below.

Scenario 1 The first scenario that we observed is when the set of bug IDs as found in the issue tracker database has no intersection with the set of bug IDs coming from the development logs. We observed this scenario in 106 projects: for the majority of these projects (81 out of 106), one of the sets of bug IDs is empty, therefore the intersection of the sets will always be empty.

Scenario 2 The second scenario that we observed is when all the bug IDs of either of the sets are contained within the other set: in the theory of sets, the cardinality of the union of the sets is the cardinality of the containing set; while the cardinality of the intersection of the sets is the cardinality of the contained set. We found an overall 145 projects that comply with this scenario, in which one of the bug ID set is a subset of the other: for 111 projects, the bug IDs found in the development logs are a subset of what found in the bug tracking data. In a further 34 projects, it is the opposite: the bug IDs found in the BTS are just a subset of what is found in the development logs.

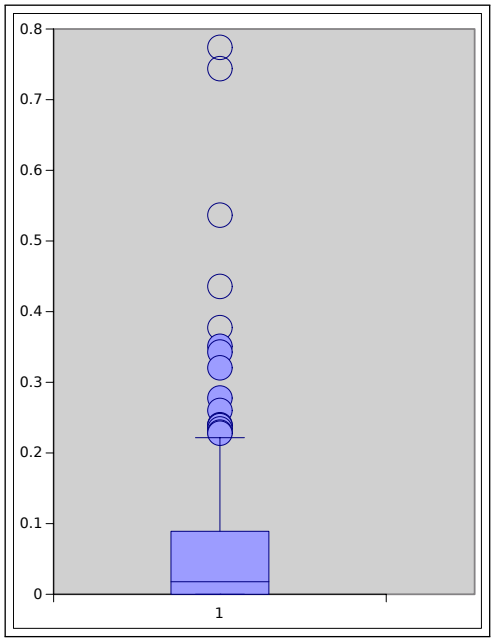

Figure 1: Ratio of bug IDs mentioned in both development logs and bug trackers, per project

Scenario 3 The third scenario is the most common: there is a subset of bug IDs that is common to the two data sources (e.g., the intersection of the sets). Apart from the common IDs, there are also (i) one subset of bug IDs that appear only in the development logs, and (ii) another subset of bug IDs that appears only in the bug tracking system.

Scenario 4 The final scenario is when all the bug IDs are found in the bug tracking system, and the development log: in the set theory language, the union of the sets is equal to the intersection of the sets. This is the ideal scenario, because the bugs are being mirrored exactly in the two databases: unfortunately, we observed this scenario in only 8 projects out of 344, and in all these cases the sets of bug IDs from both development logs and bug tracking issues were empty.

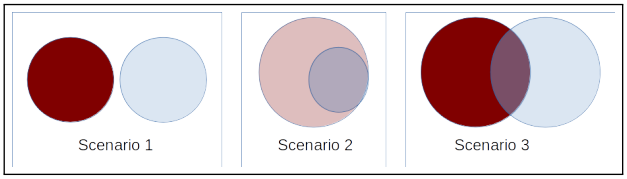

Figure 2: Three scenarios of intersection of sets

4. DISCUSSION

The research presented in this paper has the aim to be an overarching discussion about how data on bugs are being extracted and used to inform studies on bug prediction, bug triaging and identification.

The findings of this paper do not confirm the hypotheses at the basis of this research: bug IDs are not mirrored from bug trackers into development logs and vice-versa. Also, using the set of all bugs from bug tracking systems is not always definitive to describe the overall set of bugs in a software system. Therefore, the traceability of bugs in open source projects could benefit from the integration of two sources of information, one based on the development logs and one based on the bug tracking data.

The four scenarios as described show an overarching problem in traceability of bugs, which can be described as the "expressiveness" of an information source. Development logs should be expressive enough to closely follow the opening, fixing and closing of a bug; and update afterwards the bug tracking system as a proof of what was achieved during the development itself.

What we found from our sample of projects is that there is never a perfect match in what is recorded by developers in the different databases: what is more worrying, the information source that is intended primarily to track defects and their resolution is often found missing some pieces of information that are instead recorded in the development logs.

To be truly effective, our (and others') approach of tracing bugs into development logs should be integrated into a framework that not only detects and stores the discrepancies in the traceability of bugs into the development logs. The framework should also provide a means to fill the missing data in one data source, if that was to be found in the other data source. In the next section we present our framework, that has the aim to integrate different types of repositories, and various approaches to bug notations.

5. STRUCTURE OF THE FRAMEWORK

In this section, the structure of the framework is discussed, comprising six modules: The Bug Tracking Issue System, Control Version System, Bicho, CVSAnalY, SCMLog and Issues. Figure 3 below depicts the components in a UML notation that will be instantiated of the final implementation. On the other hand, Figure 4 shows the architectural overview of the framework. The next subsections describe the main components, what has been achieved so far, and what is currently missing.

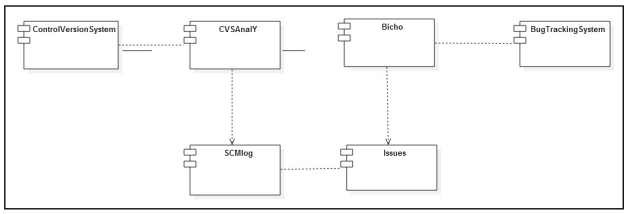

Figure 3: UML Diagram of Components

5.1 Issues-Tracker Parser through Bicho

This component provides an interface in which the interaction between Bicho and any Bug Tracking System is defined. The interface that must be implemented by each client when mining data from issue trackers is the Issues interface. Thus, the interface would enable the interaction between Bicho and the supported Bug Tracking Systems. Some BTSs

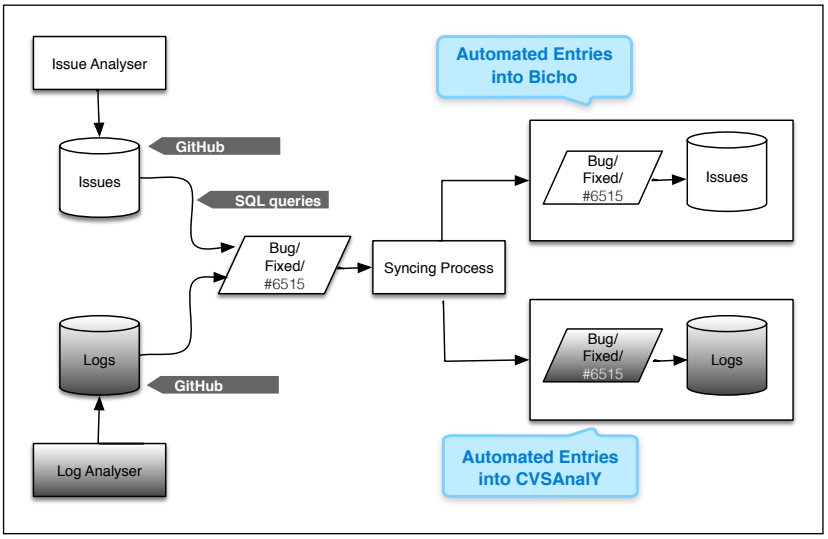

Figure 4: Architectural Overview of the framework

are currently supported by our framework: JIRA, Bugzilla, GitHub, SourceForge, Launchpad and Allura. Among these systems only GitHub requires the user to authenticate their identity using the logging credentials already registered on GitHub, before it allows any interaction or communication.

5.2 Development Log Parser through CVSAnalY

This component defines the interaction between CVSAnalY and any Version Control Systems. In addition, the SCMLog component serves as the main entry point where development logs are stored as extracted by CVSAnalY. In this way, the SCMLog interface must be implemented, in order to allow communication with any VCS. Currently, the framework supports the interaction with other VCS such as Git, CVS and Subversion.

However, one of the main obstacles among the supported CVS is that Git requires authentication by the clients or user before CVSAnalY can point to a repository in Git to extract and stored development logs. User-name and password need to be entered, in order to allow a communication between CVSAnalY and the Git (CVS). As a result, this paper implemented this framework only in its static interaction with Git: users need to first specify their logging credentials for authentication in GitHub in order to extract data by CVSAnalY from the remote VCS and stored development logs into a database generated by CVSAnalY.

5.3 Syncing Process

Observing the tables of Bicho and CVSAnalY and their attributes, we propose to use bug-related data in either database to fill the missing data as detected in the other database. Any bug IDs and attributes stored by CVSAnalY (but not found by Bicho) could be used to fill the *summary* and other attributes in the Bicho database. In consequence, automating the integration of development log data with issue tracker data (and vice-versa) will require the use of meta-data contained in the 'scmlog' table (populated by CVSAnalY) to be copied in the 'issues' table (populated by Bicho). Figure 5 shows which attributes could be used from either table to fill the gaps in the other table.

Bicho (issues table)

Column	Type
Id	Int
Tracker_id	Int
Submitted_by	Int unsigned
Assigned_to	Int unsigned
Submitted_on	dataTime
Issue	varchar
Summary	varchar

CVSAnalY (scmlog table)

Column	Type
Id	Int
Repository_id	Int
Author_id	Int
Committer_id	Int
date	dateTime
Rev	mediumText
Message	longText

Figure 5: Corresponding fields linked in Bicho and CVSAnalY

6. RELATED WORK

In this section, we report the related work that developed methods to retrieve bug-related data. We also report the tools that trace the bug-fixing commits to the bug traces in the issue trackers.

A framework to sync development logs and bugs of issues data has been proposed and designed this is by utilising and improving existing framework developed in the past by [7] [3] [14] [1] [13] [1] [8] Which are all attempts to integrate and identified missing links between logs and bugs issues data and thus provide researchers in empirical software engineering with a unified framework for integrating Bug Tracking Issues Systems and Version Control System for mining software repositories. In the same way our novelty, lies but with an attempt to synchronised either the missing version control commit logs or bugs issues data of software projects retrieved by these tools (Bicho and CVSAnalY) and stored into their respective databases.

The Buco reporter, developed by Ligu et al. [8], is an extensible framework that mirrors the development logs and the bug tracking data, and it generates a complete set of evolutionary facts and metrics about a given OSS projects. Buco accurately traces development logs and bugs, but it was not designed and developed to synchronise the missing development logs and bugs, if discrepancies were found. The

contribution of the presented research is a complete framework to synchronize the missing development logs and bugs, supporting various repositories and bug tracing algorithms and approaches.

The Linkster tool involves a series of steps to retrieve, parse as well as convert and link the data sources [3]. As a result, it requires significant manual effort to analyze recovered links which might be much more accurate. On the other hand, RELINK [14] collects information automatically from the source code repository and bug tracking system, builds the resulting information linked between bugs/issues or logs and output the identified links. In general, both these tools require a large amount of interaction but they recover missing logs and bugs/issues accurately. Our approach completes these tools by filling the missing data in either database in an automatic way.

7. CONCLUSION AND FUTURE WORK

This short paper presents the results of an extended quantitative analysis on a sample of 344 OSS projects, and how the bug-related data is stored in the development logs and the bug tracking logs. The set of bug IDs from the development logs was compared to the set of bug IDs as found in the issue tracker systems. The objective of this research was to ascertain how much discrepancy is visible when considering these two sources of information, and whether either could be considered as a complete and credible set of data regarding bug issues.

We found that over half of the projects sampled have a portion of bug IDs mentioned in one source (either development logs or bug tracking logs) but not in the other. We also found that the intersection of "common" or shared bug IDs is very low (around 20% for some 75% of the projects in the sample), while in some extreme cases projects hold distinct, not shared set of IDs in either database.

Furthermore, we also presented a framework for detecting and automatically synchronising missing bug-related data from these sources. The extraction of data and detection of discrepancies is partly completed: future work comprises the implementation of the full SZZ algorithm. The syncing process has been laid out and it will be also part of future work.

8. AKNOWLEDGMENTS

The authors would like to thank Dr T Hall for the extensive comments and support on various aspects of this work; and Dr G Robles for the feedback given on an earlier version of this paper.

9. REFERENCES

[1] P. Anbalagan and M. Vouk. On mining data across software repositories. *2009 6th IEEE International Working Conference on Mining Software Repositories*, 2009.

[2] J. Aranda and G. Venolia. The secret life of bugs: Going past the errors and omissions in software repositories. In *Proceedings of the 31st International Conference on Software Engineering*, pages 298–308. IEEE Computer Society, 2009.

[3] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein. Linkster: enabling efficient manual inspection and annotation of mined data. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 369–370. ACM, 2010.

[4] H. Hayashi, A. Ihara, A. Monden, and K.-i. Matsumoto. Why is collaboration needed in oss projects? a case study of eclipse project. In *Proceedings of the 2013 International Workshop on Social Software Engineering*, pages 17–20. ACM, 2013.

[5] H. Hemmati, S. Nadi, O. Baysal, O. Kononenko, W. Wang, R. Holmes, and M. Godfrey. The msr cookbook: Mining a decade of research. *2013 10th Working Conference on Mining Software Repositories (MSR)*, 2013.

[6] I. Herraiz, D. Izquierdo-Cortazar, F. Rivas-Hernández, J. Gonzalez-Barahona, G. Robles, S. Duenas-Dominguez, C. Garcia-Campos, J. F. Gato, and L. Tovar. Flossmetrics: Free/libre/open source software metrics. In *Software Maintenance and Reengineering, 2009. CSMR'09. 13th European Conference on*, pages 281–284. IEEE, 2009.

[7] M. Legenhausen, S. Pielicke, J. Ruhmkorf, H. Wendel, and A. Schreiber. Repoguard: a framework for integration of development tools with source code repositories. In *Global Software Engineering, 2009. ICGSE 2009. Fourth IEEE International Conference on*, pages 328–331. IEEE, 2009.

[8] E. Ligu, T. Chaikalis, and A. Chatzigeorgiou. Buco reporter: Mining software and bug repositories. page 121, 2013. Retrieved January 23, 2015 from http://ceur-ws.org/Vol-1036/p121-Ligu.pdf.

[9] G. Robles, S. Koch, J. M. GonzÃąlez-Barahona, and J. Carlos. Remote analysis and measurement of libre software systems by means of the cvsanaly tool. In *In Proceedings of the 2nd ICSE Workshop on Remote Analysis and Measurement of Software Systems (RAMSS)*, pages 51–55, 2004.

[10] B. A. Romo, A. Capiluppi, and T. Hall. Filling the gaps of development logs and bug issue data. In *Proceedings of The International Symposium on Open Collaboration*, page 8. ACM, 2014.

[11] M. Shepperd. How do i know whether to trust a research result? *Software, IEEE*, 32(1):106–109, 2015.

[12] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? *ACM SIGSOFT Software Engineering Notes*, 30(4):1–5, 2005.

[13] A. Sureka, S. Lal, and L. Agarwal. Applying fellegi-sunter (fs) model for traceability link recovery between bug databases and version archives. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 146–153. IEEE, 2011.

[14] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European conference on Foundations of Software Engineering*, pages 15–25. ACM, 2011.

[15] H. Yang, C. Wang, Q. Shi, Y. Feng, and Z. Chen. Bug inducing analysis to prevent fault prone bug fixes. 2014. Retrieved February 15, 2015 from http://software.nju.edu.cn/zychen/paper/2014SEKE1.pdf.