Data Collection for Software Defect Prediction – an Exploratory Case Study of Open Source Software Projects

Goran Mauša*, Tihana Galinac Grbac * and Bojana Dalbelo Bašić**
* Faculty of Engineering, University of Rijeka, Rijeka, Croatia
** Faculty of Electrical Engineering and Computing, University of Zagreb, Zagreb, Croatia
goran.mausa@riteh.hr, tihana.galinac@riteh.hr and bojana.dalbelo@fer.hr

Abstract – Software Defect Prediction (SDP) empirical studies are highly biased with the quality of data and widely suffer from limited generalizations. The main reasons are the lack of data and its systematic data collection procedures. Our research aims at producing the first systematically defined data collection procedure for SDP datasets that are obtained by linking separate development repositories. This paper is the first step to achieving that objective, performing an exploratory study. We review the existing literature on approaches and tools used in the collection of SDP datasets, derive a detailed collection procedure and test it in this exploratory study. We quantify the bias that may be caused by the issues we identified and we review 35 tools for software product metrics collection. The most critical issues are many-to-many relation between bug-file links, duplicated bug-file links and the issue of untraceable bugs. Our research provides more detailed, experience based data collection procedure, crucial for further development of SDP body of knowledge. Furthermore, our findings enabled us to develop the automatic data collection tool.

I. INTRODUCTION

The research community is in need of a systematically defined data collection procedure for Software Defect Prediction (SDP) studies. In order to achieve this goal, we need to be aware of all the issues that could lead to data bias. Data collection from freely available open source projects is becoming increasingly popular, so we decided to reveal as many issues that are present in their development repositories.

A great number of empirical studies tried to indentify the best approach to SDP and most of them are collected in several systematic review [1,2,3,4]. However, there is still work to be done on reproducibility of empirical studies based on datasets retrieved from development repositories [5]. Detailed and replicable data collection procedures have to accompany the datasets in order to achieve greater generalization of SDP studies [1]. Furthermore, it is known that SDP repositories often suffer from data quality issues like: outliers, missing values, redundant or irrelevant attributes and instances, overlapping classes, contradictory samples, data shifting, imbalance of classes or replicability [6], [7]. Thus, we need to start systematically build a knowledge base on reusable research datasets. In this paper, we want to stress on role of human introduced bias in data collection procedure because it is developers who make entries into development repositories. We also recognize the importance of reusable tools for automated data collection that could play significant role in systematically building of uniformly obtained datasets. Freely available reusable tools, that have open source software with clearly defined data collection mechanism implemented, could form a base for wider discussion of data collection issues and its relations to bias. It is important that all research in the area of SDP collects the data according to a standardized procedure in order to perform repeatable and verifiable case studies that will benefit the body of knowledge. Thus, we aim to develop a rigorous systematic data collection plan that could be reused by other researchers. We also plan to quantify the main sources of bias. That kind of information would improve comparability between studies and generalization ability. Building a detailed data collection procedure and addressing all the possible issues is the first and crucial step in that process. Moreover, we believe the procedure should also be explained clearly enough to be applicable in industrial research as well as in academic that most often rely on open source software [8].

The paper is structured according to the guidelines given by Runeson and Höst [9] and Kitchenham et al. [10]. Section II puts our research into context and presents the related work. Section III explains in detail experimental planning where we organized the objective of our research into several research questions, presented the subject of the experiment and the means to perform it, defined the tasks that are going to be executed and methods how to do it, described the experiment design and prepared the analyzes and validation procedures that will enable us to answer the research questions. Section IV describes the execution of experiment. Finally, we present the results and answers to each research question in Section V and conclude the paper in Section VII.

II. BACKGROUND AND RELATED WORK

The importance of building a verifiable, repeatable and accurate data collection plan was recognized and put into several steps by Basili and Weiss [11]. Our research goal is to create most often used type of SDP datasets

using such a data collection plan. We analyzed all the 36 papers that were selected for final analysis in [1] because those were the ones that contained description of data collection procedure. However, we identified there is a lack of systematic unified mechanism in that area. There is a number of SDP studies that contain a section describing their data collection. However, there are no guidelines for reporting the crucial details of data collection process. Hence, it is not not performed systematically and one could not compare and quantify bias introduced with the data collection process.

Nevertheless, there were some attempts in that direction. A very good example of planned validation procedure can be found in [12]. They analyzed possible threats to validity, prepared and performed a manual check of a subset of potentially flawed data and evaluated the soundness and completeness of their data collection procedure. A detailed plan of analysis that should confirm the appropriateness of a novel approach for the data collection procedure in the phase of linking bugs to changes is present in [13]. They prepared a sort of benchmarking data set, manually collected and validated using observer triangulation. They named it "*the ground truth*" and used it to compare their novel approach to the traditional one in terms of precision, recall and F-measure evaluation metrics. We believe this approach should be used more often when introducing novel concepts because with the use of statistical tests of significance it could accelerate the diversification of true improvements. Confirmation to our statements comes from research presented in [14] that used the same "ground truth" data set and in [12] that even improved and enlarged such a data set. One commonly identified issue is the *bug linking* issue when collecting the data from two separated repositories of bugs and source code changes is addresed in [15]. The paper [16] identified the bug-feature bias and commit feature bias in defect datasets.

To the best of our knowledge, this is the first research intended to find all other issues present in the data collection procedure that could lead to biased software defect prediction datasets. We want to identify all the critical information that may be useful in comparison of different studies. It is important to be systematic because otherwise we could unintentionally include a source of bias and not be able to evaluate its impact. In our study we used students to manually collect the datasets and we draw conclusions from their data collection process. The similar approach is mentioned as in [16] but no systematic approach for the analysis of manually collected datasets is proposed. In this paper we conduct a systematic experiment which aims to identify all the traps in data collection process that lead to bias. In the remainder of the paper we refer to them as **issues**.

III. EXPERIMENTAL PLANNING

A. Research Goal

Our research goal is to create a systematic data collection procedure for SDP data sets that will enable us to develop automated data collection tool. To achieve this goal, we follow the process of data collection procedure development presented in Figure 1. This paper presents the first step of that process. Its purpose is to reveal and quantify all the issues that stand in the way of creating the data collection procedure, and to find the appropriate software metrics tool.

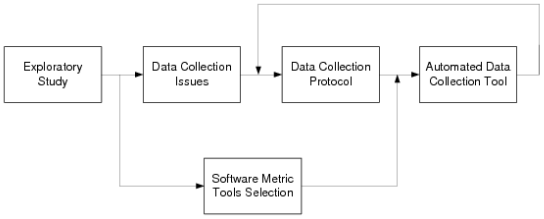

Figure 1. Research plan

B. Research Questions

Our research goal is divided into two research questions that will guide our exploratory study:

RQ1. Which issues affect the data collection procedure and how could they be overcome?

RQ2. Which software metrics tools are best suited for automatic collection?

Giving answer to **RQ1** is the primary purpose of this study. We hope to acquire as much experience and insight into the issues of data collection process as possible. Answering **RQ2** will help us develop an automated data collection tool. There is a number of software metrics (SM) tools present on the Internet and in related work so we need to review them and implement the ones that satisfy certain criteria.

C. Experimental Subjects

The participants that conduct the data collection are 12 students of the Faculty of Engineering at the University of Rijeka, Croatia, enlisted in the course of Software Engineering Management, obligatory in the first year of graduate Computer Science study program. This is an appropriate course because it teaches the students the importance of proper and accurate data in the decision making processes in the evolution of a large and complex software product. The participants had no previous experience with SM tools, little experience with bug repository use and mixed experience with source code management repositories. That is why they were all introduced to the concepts of these repositories in preparation exercises and seminars. Meanwhile, we omitted the fact that this is an exploratory study and that the observer triangulation is used.

D. Experimetnal Material

The main experimental material are: software development repositories, SM tools, and data collection forms. There are two **development repositories** crucial to the SDP data collection: the bug tracking (BT) repository and the source code management (SCM) repository. The Eclipse open source community is the most often used subject in SDP data collection so we used its largest projects in our experiment: JDT, PDE, Platform, BIRT and Mylyn. We use the term large projects for those that contain houndreds or thousands of

source code files and have been developed through at least a decade of evolution. The Eclipse community uses Bugzilla as its BT and GIT as its SCM repository. Bugzilla is a Web-based general-purpose bug tracker that allows developers to keep track of both defects that cause loss of functionality and requirements' requests. GIT is a distributed version control and source code management system with complete history and full version tracking capabilities. The **SM tools** are required in the second phase of our experiment for calculation of various product metrics. Further details are given in section IV-A. We use five **data collection forms** in the data collection procedure. Each form is prepared and explained in details for each of the tasks demanded from the participants to solve:

Form 1 is designed to collect the pieces of information from the BT repository: Bug ID, Product, Version, Summary (Full), Priority, Assignee, Number of Comments, Opened, Changed.

Form 2 is designed to document the bug-commit linking output: Bug ID, Found (yes/no), Commit ID.

Form 3 is designed to collect the pieces of information related to the linked files from the SCM repository: File path, Bug ID, Insertion, Deletion

Form 4 is designed to collect all the metrics obtained by SM tools both before and after the bug fix: File path, Product Metrics (m_1, m_2, ..., m_n).

Form 5 is designed as the final dataset form that containes all the metrics and the number of bugs for each file: File path, Metrics (m_1, m_2, ..., m_n), Number of bugs, Release.

E. Tasks and Methods

We need to perform several **tasks** to answer our research questions:

1. Examine the appropriateness of SM tools for our data collection procedure
2. Execute the data collection procedure manually and automatically
3. Analyze the collected data and examine the sources of bias

The first task gives us the answer to the **RQ2**. It needs to be done prior to data collection because it is an investigation of its own and the results are implemented in the second task. The manual execution of the data collection reduces initial researcher bias and gives insights into the issues that need to be addressed. Making the data collection precess automatic could reveal some additional issues. Furthermore, the manual and automatic data collection provide us with valuable data for evaluation of the issues and for answering the **RQ1**. In order to conduct all of these tasks, we use the following **methods**:

1. **Seminars** for reviewing the SM tools
2. **Written Instructions with Examples** for each step of the data collection procedure
3. **Data analysis and validation methods** of the results obtained during the experiment

The participants are given a preselected subset of the SM tools for reviewing and they are involved in the execution of data collection procedure. The authors of this paper perform the literature and Internet survey to locate all the SM tools, form a list of the most representative range of product metrics, find the candidate tools for further examination through seminars, plan the data collect procedure, prepare the written instructions and examples for each step of the data collection procedure, and determine the data analysis and validation techniques that we explain in more details in the following section.

F. Experimental Design

The manual collection is executed to identify all the issues present in our data collection process. The task scheduling is performed so that it allows a validation of the performance of different SM tools and the data itself early enough to enable timely corrective action on data collection procedure. Since participants could be the source of bias as well, we assign each project and each tool to more than one participant, i.e. following the principle of observer triangulation [13]. The participants are given personalized tasks in 4 exercises with 52 steps and explained by 21 screen-shot figures. The data collection procedure is divided into several phases. Each phase, along with its issues is presented in Figure 2. We describe them in more details and give a proposition how to solve them in the remainder of this subsection.

SM Tool Selection: The initial phase is to select the SM tools for collection of software metrics that represent the independent variables in SDP. Since we decided to analyze Eclipse projects and collect the most often used software metrics, the tool needs to be able to analyze large projects written in java and give reports for calculated software product metrics on file or class level of granularity.

Issue 1.1 The SM tools requirements are:

1. **Availability of tool** - for in-use testing
2. **Programming language** – java
3. **Level of granularity** – file / class level
4. **Software metrics** – product metrics
5. **Usability - input** – capable for large projects
6. **Usability - output** – *csv*, *html* or *xml* reports

Project Selection: The initial phase also needs to select the open source project that will be the source of SDP data set. The issues that affect this initial phase are:

Issue 2.1 The project needs to be present both in the BT and in the SCM repository

Issue 2.2 The project's releases from BT must be present as release tags in SCM repository

The presented issues are rather self-explanatory. We need to be able to access the source code, the bugs and the corresponding releases in both repositories. We evaluate these issues by choosing a great number of projects from the chosen Eclipse community, and analyze which projects contain release designations in both repositrories.

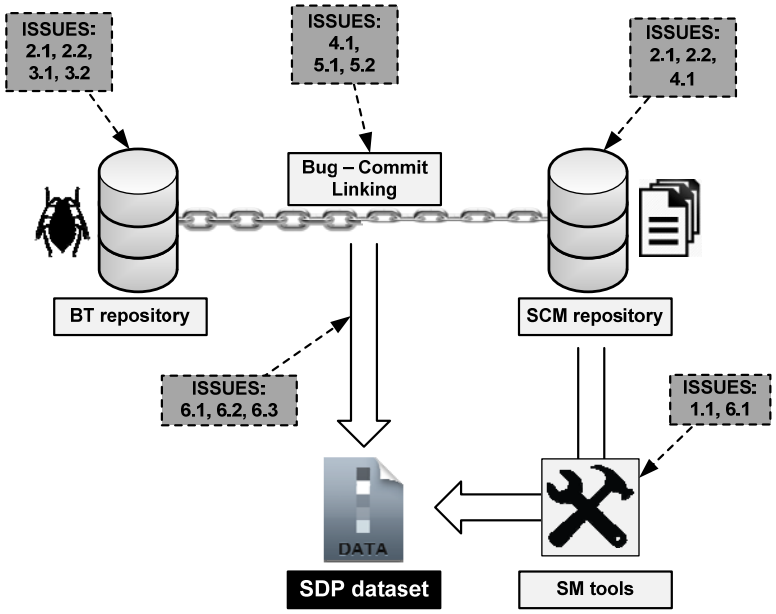

Figure 2. The impact of identified issues on each data collection phase

Bug Repository Collection: The first data collection phase is to find defects from the Bugzilla repository and collect all the valuable pieces of information.

Issue 3.1 The bug must be relevant to SDP
Issue 3.2 The bug should be solved

BT repositories typically store bugs and enhancements. Furhteremore, bugs may relate to minor typographical errors. The bugs relevant to SDP are the ones that represent a loss of functionality which is indicated by severity level above trivial. Since we need to linking the bug to source code, the bugs need to be solved in order to be present in the commits from the SCM repository. A BT tabular report, as presented in **From 1**, for all releases of each project is required with following parameters:

- Bug Status: resolved, verified and closed
- Bug Resolution: fixed
- Bug Severity: blocker, critical, major, normal and minor

This way we bypass all the unconfirmed, invalid or duplicated defects, defects that are not solved or that could not be solved and requirement requests.

We evaluate the impact of these issues by comparing the number of bugs that satisfy our parameters to the number of bugs that satisfy opposite parameters (Bug Status: unconfirmed, new, assigned and reopened; Bug resolition: invalid, wontfix, duplicate, worksforme, moved and not_eclipse; Bug severity: enhancement, and trivial).

SCM Repository Collection: The second data collection phase is to find the SCM repository for the chosen project in GIT. We need stable releases of the project for future analysis of software metrics. If issues 2.1 and 2.2 are both satisfied, there remains only one issue, related mostly to large projects and we evaluate it by counting the number of such projects.

Issue 4.1 The project may be divided into several sub-repositories within the foundation's top level GIT repository

Bug - Commit Linking: The most important data collection phase is to link the BT and SCM repositories because it yields the number of defects per software module. Every source code change is stored within commits and each commit has the developers description of the change within a message. The linking is done by searching the Bug ID in the commit message. We use **Form 2** to record successful linking.

Issue 5.1 The Bug ID number may be a part of larger block of characters
Issue 5.2 The Bug ID may be present in more than one commit

The issue of varying length of the Bug ID presents an obstacle to automatic data collection. It is important to search for the exact match of the Bug ID. We evaluate its impact by calculating the percentage of the Bug IDs we can link with the simple search and the strict search we mentioned. The second issue is because of the nature of complex bugs that require a lot of change, possibly in different subrepositories, and because of the loose development rules in the community. We evaluate it by comparing the number of linked commits and the number of bugs.

Source Code Analysis: This phase consists of calculating the product metrics of all the source code files in each release of the chosen project. The SM tools are used for that purpose. All the product metrics obtained by these tools are collected in **Form 4**.

SDP Dataset Generation: The final phase of our data collection procedure is to construct the final SDP dataset in the **Form 5**, combining the output of the previous two phases.

Issue 6.1 The file identifier needs to be unique for the *Source Code Analysis* and the *Bug - Commit Linking*
Issue 6.2 File – Bug cardinality is many-to-many
Issue 6.3 Single file can be changed in multiple commits that are linked to one bug

The unique identifier of any file present in one release is its *File Path*. In order to cope with issue 6.1, the file path was extracted from commits in the same way it was extracted in the output of SM tools. Neglecting this issue would make it very difficult to connect the previous two phases. Due to issue 6.2 and 6.3, we count the number of distinct *File Path* and *Bug ID* (FB) ordered pairs in **Form 3**. To evaluate the potential impact of issue 6.2, we compare the number of FB pairs with the number of linked files and bugs. The impact of issue 6.3 will be evaluated through the number of FB duplicates.

IV. RESULTS

The literature and Internet survey found 35 SM tools (12 from [8] and additional 23 from the Internet search) and we analyzed them all. The tools were compared against demands presented in subsection III-F. In the first stage, we analyzed the tool descriptions. In the second stage, we anylzed the remaining tools more thoroughly in usage on a sample project through participants' seminars. In the third stage, we tested the tools in large scale manual data collection. 19 tools passed the first review stage, 5 tools passed the second and only 2 remained in the end. Here is a full list of tools we analyzed: iPLASMA, inFusion, inCode, CodeCover, JavaNCSS, ckjm, Understand, Borland Together, CodeSonar Static Analysis Tool, CodePRO Analytix, Metrics 1.3.8, Moose, JavaMetrics, Testwell CMT ++ (CMTJava), JDepend, Dependency Finder, JarAnalizer, CCCC, Source Code Metrics, Classycle, Sonar, Resource Standard Metrics, Jhawk, Jtest, SonarGraph, PMD, JDiff, CodeCount, LocMetrics, CodeAnalyzer, JMT, Jmetric, ES2, Xradar, Essential Metrics. Table 1 presents 7 categories of disqualifying factors we applied and the number of tools reject due to each of them.

The tools that were tested in use are: CodePRO Analytix, Metrics 1.3.8, Source Code Metrics, Jhawk and LocMetrics. Metrics 1.3.8. required projects to be built and Source Code Metrics required the restart of NetBeans for each following analysis. A great number of analyses dictated to exclude these two tools. We did not manage to automate the Code PRO Analytix and since it was incapable of analyzing whole project releases it was discarted as well. The LOC Metrics[1] proved to be a fast tool offering a sample of 10 simple product metrics and the JHawk[2] enriched the set of metrics with additional 40. The complete set of software product metrics provided by these two tools is wider than in any other related study.

[1] http://www.locmetrics.com/
[2] http://www.virtualmachinery.com/jhawkprod.htm

TABLE I. SM TOOLS REVIEW - ISSUE 1.1

Requirement	*Important characteristics*	*Rejection rate*
1	Availability of tool	5
2	Programming language - java	0
3	Level of granularity - files	7
4	Software metrics - product - too few metrics	4 4
5	Usability - large input	3
6	Usability - output reports	1
7	Other - older version of another Other - not a tool, a platform Other - trial version too limited	2 2 2

The evaluation of data collection issues is presented in tables 2, 3, 4, 5 and 6. Table 2 reveals that not all of the Eclipse projects are present in the GIT repository. We analyzed 85 different Eclipse projects from BT repository. We found 76 of them in SCM repository, indicating that the issue 2.1 is rarely present. Furtheremore, for 51 projects we managed to find release tags in SCM that correspond to release numbers from BT. It shows the naming convention of releases are not always consistent and that issue 2.2 could impact greater number of projects. Finally, we found that 37 projects are distributed in more than one sub-repository, showing that it is important to be aware of issue 4.1. Table 3 presents the importance of choosing the appropriate parameters while searching for bugs in the BT repository. We compare the number of bugs collected by Our parameters (subsection III-F-Bug repository collection) and by other parameters. We can expect a completely different SDP dataset if we do not limit the *Bug Resolution* to fixed only. On the other hand, bugs with incomplete data (expected by **Form 1**) are rare and could cause insignificant bias. The inclusion of bugs of opposite severity may cause from 10% - 15% bias in the number of bugs. For the Mylyn project, the bias rises to 45%. Table 4 demonstrates the need for a precise search criterion for establishing the bug - commit links. A great number of incorrectly linked bugs may occur if we simply search for the Bug ID as a series of digits (Simple Search) and not defining its surrounding characters (Strict search). The bias in the linking rate from 7% - 40% proves issue 5.1 to be very important. Table 5 reveals that a single bug ID may be linked to more than one commit and that it is highly important to be aware of issue 5.2. Table 6 reveals that the number of FB pairs is much greater than the number of files that were linked to bugs, proving that the majority of bugs is located in a smaller part of the software. We also notice that the number of linked files can be both greater (PDE, BIRT, Mylyn) or lower (JDT, Platform) than the number of linked bugs, proving that issue 6.2 needs to be taken into account. The number of FB pair duplicates is presented in table 7. We performed additional manual root cause analysis of a subset of FB duplicates. We discovered that some FB pairs even occurred more than twice and their cause was:

TABLE II. EVALUATION - ISSUES 2.1, 2.2 AND 4.1

Demand	*Important characteristics*	*Acceptance rate*
-	Total projects analyzed	85
1	Present in BR and SCM repository	76
2	Corresponding release tags	51
3	>1 SCM subrepository	37

TABLE III. EVALUATION - ISSUES 3.1, AND 3.2

Bug requirement	**Number of bugs per project**				
	JDT	***PDE***	***Platform***	***BIRT***	***Mylyn***
Our paramaters	20353	7318	38511	13071	3492
Opposite severity	2875	895	4604	1243	1581
Opposite status	0	0	0	0	0
Opposite resolution	16488	3570	32582	3059	1909
Incomplete bug data	64	26	108	7	1

TABLE IV. EVALUATION - ISSUE 5.1

Linking Technique	**Linking rate per project**				
	JDT	***PDE***	***Platform***	***BIRT***	***Mylyn***
Bugs:	18696	6822	34641	8101	2739
Simple Search:	80.5%	65.8%	80.7%	59.8%	52.3%
Strict Search:	72.0%	58.6%	71.2%	18.9%	16.1%

• There are both a bug test and a bug fix reported for the same FB pair

• There are 2 bug fixing commits reported in the SCM for the same FB pair

• There had been reported one or two temporary bug fixes before the final bug fix was reported

• There are both a bug fix and a revert bug fix reported for the same FB pair

All the reasons we found show us that only the final bug fixing commit should be regarded as relevant

V. THREATS TO VALIDITY

There are two major threats to validity to our experiment: the specificity of data (only several largest Eclipse projects written in java stored in Bugzilla and GIT) and the possibility of human error from the participants. However, observer triangulation and the partial automation of the process reduced this the human error influence and most recent experiments showed the procedure can be also successfully used for other projects from other foundations, like Mozilla or Apache. In contrast to usual data collection procedures, we welcomed the differences in collected data from different participants because those were indicators of issues we need to address.

VI. CONLUSION

The goal of this exploratory research was to identify as many issues that may cause bias in the SDP data collection procedure. Our survey of related research and other sources in this area available on the Internet resulted in a preliminary data collection protocol. That protocol, along with detailed descriptions was given to 12 participants of our experiment and they were assigned 5 major Eclipse projects. Their output data was then analyzed and validated through a number of metrics and manual inspections.

TABLE V. EVALUATION - ISSUE 5.2

Number of	**Multiple bug-commit links per project**				
	JDT	***PDE***	***Platform***	***BIRT***	***Mylyn***
Bugs linked:	13468	3995	24663	1532	441
Commits linked:	17667	7573	29669	2127	3002

TABLE VI. EVALUATION - ISSUE 6.2

Number of	**Multiple file-bug links per project**				
	JDT	***PDE***	***Platform***	***BIRT***	***Mylyn***
Bugs linked:	13468	3995	24663	1532	441
Files linked:	12080	6629	15084	4273	975
Bug-file links:	25476	16595	23784	6663	1258

TABLE VII. EVALUATION - ISSUE 6.3

Number of	**Duplicated file-bug links per project**				
	JDT	***PDE***	***Platform***	***BIRT***	***Mylyn***
FB duplicates:	768	1233	903	4	0

As an answer to our **RQ1,** we identified and quantified several important issues that may become a significant source of bias in the SDP dataset. The collection starts with the choice of proper open source projects that contain their data in the BT and the SCM repositories. Some larger projects contain several SCM subrepositories. Relevant bugs are the resolved, verified and closed ones with the resolution fixed and severity level above trivial because SDP requires only those that caused a loss in functionality. Linking between bugs and commits must be done in a carefully defined strict manner because otherwise we might obtain bias in linking rate. Multiple links between bugs and commits (i.e. files) and duplicated file-bug links can be expected. We also answered our **RQ2**. After we thoroughly analyzed 35 SM tools, we identified *LOC Metrics* and *JHawk* as the most appropriate for our automatic data collection tool [17]. These two SM tools provide us with a respectable subset of 50 product metrics and are suitable for automating the data collection process.

Finally we find evidence that human error is widely present in development repositories. Allthough the Bug ID is a unique identifier of a software defect, it cannot be always linked to the commit that made it fixed. Therefore, a certain number of bugs remains untraceable and becomes an issue in the phase of determining the non-fault-prone files as well as the fault-prone ones. However, there are indications that some versions suffer a lot less from this problem and a more detailed analysis is intended for our future work.

ACKNOWLEDGMENT

The work presented in this paper is supported by the University of Rijeka research grant Grant 13.09.2.2.16.

REFERENCES

[1] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov. 2012.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *J. Syst. Softw.*, 83(1):2–17, Jan. 2010.

[3] T. M. Khoshgoftaar and N. Seliya. Comparative assessment of software quality classification techniques: An empirical case study. *Empirical Software Engineering*, 9(3):229–257, 2004.

[4] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Softw. Engg.*, 17(4-5):531–577, Aug. 2012.

[5] J. M. González-Barahona and G. Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Softw. Engg.*, 17(1-2):75–89, Feb. 2012.

[6] D. Rodriguez, I. Herraiz, and R. Harrison. On software engineering repositories and their open problems. In *Proceedings of RAISE '12*, pages 52–56, 2012.

[7] M. J. Shepperd, Q. Song, Z. Sun, and C. Mair. Data quality: Some comments on the nasa software defect datasets. *IEEE Trans. Software Eng.*, 39(9):1208–1215, 2013.

[8] T. Galinac Grbac, P. Runeson, and D. Huljenic. A second replicated quantitative analysis of fault distributions in complex software systems. *IEEE Trans. Softw. Eng.*, 39(4):462–476, Apr. 2013.

[9] P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Softw. Engg.*, 14(2):131–164, Apr. 2009.

[10] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. E. Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Trans. Softw. Eng.*, 28(8):721–734, Aug. 2002.

[11] V. R. Basili and D. Weiss. A methodology for collecting valid software engineering data. *IEEE Computer Society Trans. Software Engineering*, 10(6):728–738, 1984.

[12] T. F. Bissyande, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Reveillere. Empirical evaluation of bug linking. In *Proceedings of CSMR '13*, pages 89–98, Washington, DC, USA, 2013. IEEE Computer Society.

[13] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: Recovering links between bugs and changes. In *Proceedings of ESEC/FSE '11*, pages 15–25, New York, NY, USA, 2011. ACM.M. Young, The Technical Writer's Handbook. Mill Valley, CA: University Science, 1989.

[14] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Multi-layered approach for recovering links between bug reports and fixes. In *Proceedings of FSE '12*, pages 63:1–63:11, New York, NY, USA, 2012. ACM.

[15] A. Sureka, S. Lal, and L. Agarwal. Applying fellegi-sunter (fs) model for traceability link recovery between bug databases and version archives. In *Proceedings of APSEC '11*, pages 146–153, Washington, DC, USA, 2011. IEEE Computer Society.

[16] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: Bias in bug-fix datasets. In *Proceedings of ESEC/FSE '09*, pages 121–130, New York, NY, USA, 2009. ACM.

[17] G. Mauša, T. Galinac Grbac, and B. Dalbelo Bašić. Software defect prediction with bug-code analyzer - a data collection tool demo. In Proc. of SoftCOM '14, 2014.