Chapter 9

How to Improve Linking Between Issues and Commits for the Sake of Software Defect Prediction?

1. Introduction

Bug predictions and defect predictions can save a lot of money which otherwise would be spent on bug fixes. Commit logs and bug reports are very often not linked with each other [1] although those links can provide very valuable information which can help in software defect predictions and project evolution. According to bugs prone – by counting the number of bug reports that are matched with them. For the project evolution, those links can generate defect data – for example the number of defects related to various classes in a project. As a result, it is possible to develop defect prediction models for software projects, e.g., [3, 4, 5].

The other problem is misclassification between bugs and non-bugs – many issues which are classified as ’bugs’ refer to maintenance, refactoring or enhancements. Regarding to one of the reports regarding how misclassification impacts bug prediction [1] this problem is very common. Authors of the article [1] have conducted a manual examination of more than 700 issue reports of five open source projects. Their result revealed that 33,8% bug reports were misclassified – being not code fixes, but rather new features, refactorings or documentation updates. There are many simple approaches in matching links with issues, mainly based on simple textual matching. However, there are also three promising approaches (ReLink, MLink and RCLinker) which are based on repository changes and features extraction from bugs and issue trackers metadata. One of the approaches (RCLinker) is based on the machine learning too which is described in the section below.

In Section 2 we have described those three approaches to link issues with commits. Defect Prediction in Software Systems (DePress) [6] Extensible Framework allows building workflows in graphical manner. DePress is based on the KNIME project. The main aim of the DePress Framework is the support for empirical software analysis. It allows collecting, combining and analysing data from various data sources like software repositories or software metrics.

Our work provides the following contributions:

1) We wanted to find a way to link effectively commit logs and bug reports. That is why we have decided to use modified by us RCLinker approach. To achieve this, we have decided to use Defect Prediction in Software Systems and implement our approach of the RCLinker algorithm as a new node in the workflow. We used machine learning too.
2) To validate and check our approach we have tested it on the three projects – two open source projects by Pivotal Software: Spring Data Redis, Spring OSGi and the proprietary commercial project provided by Capgemini.
3) The RClinker approach uses metadata and textual features. We have proposed new features based on the JIRA metadata to check if they will improve the results.

2. Related Work

There are many approaches to linking issues with commit. We have reviewed literature using snowball sampling, described by Wohlin in [7]. Descriptions of the relevant articles are presented in the subsequent subsections.

2.1. ReLink: recovering links between bugs and changes

ReLink [8] is the simplest algorithm on which we will base our work. Traditional approaches for linking issues with commits presuppose that developers are on three properties:

1) Time interval – it is a time difference between commit date and issue modification date. After each fix, developer must update issue in tracking system, so the time difference will be small.
2) Issue owner and commit author – if issue owner is the same as commit author, this issue is probably connected with the commit
3) Text similarity – commit message should be similar to the issue description if they are linked. To normalize text in issue and commit message, ReLink uses the following techniques: removing stop-words, stemming and using synonyms – for example, change "additional" to "extra".

ReLink has a "learning" phase. To learn its model we must follow these steps:

1) Assign a very small value to time interval.
   (a) Assign a very small value to the text similarity threshold.
   (b) Discover links with traditional heuristics for given time interval and similarity threshold, then count number of discovered links and then calculate F-measure using metrics like "Percent of commits that fixes bugs" (more possibilities are below).
   (c) Increase text similarity threshold a little bit
   (d) Repeat steps 3 to 4 until we reach a maximum value of threshold
2) Increase time interval a little bit.
3) Repeat steps 3 to 6 until we reach a maximum value of time interval.
4) Choose threshold for two "properties" with the highest F-measure.
5) Return threshold and time interval.

To start discovering new connections, we must run two algorithms to get proper criteria and then ReLink will "learn" these criteria. After that, ReLink checks links that fulfils criteria. After checking all links, ReLink returns its list.

ReLink discovers up to 26% more links than the traditional approach [8]. It is often used with the following metrics: percent of commits that fixes bugs, percent of files with defects and average time of bug fixing.

2.2. MLink: multi-layered approach for recovering links between bug reports and Fixes

MLink [9] is a multi-layered approach to automatically recover issue links. In comparison to ReLink, it is not only based on the terms-linking method but it checks the changes in the code repository too and tries to link them with the issues metadata.

MLink uses cascading layers – each layer has a detector with its own set of textual and code features. The layers' input is a filtered set of the candidate links which comes from the previous layer – it means that each detector can be used as a filter. It reduces the amount of the links and passes the set to the next layer. Layers which have filters with higher levels of confidence on accurate detection are applied at earlier levels.

This model consists of six detectors:

1) **Pattern-based detector** – this is similar to ReLink approach – issues metadata and commits logs messages are checked if they contain some typical patterns such as 'fix the issue ...', 'fix the bug ID...' etc.
2) **Filtering layer** – the remaining links from the previous layer are analyzed if they violate time constraint – it means that the commit time for the fix must be between open and close time of the corresponding issue.
3) **Patch-based detector** – it extracts the patch code recommended by the bug reporters or people who have mentioned it in the issues comments.
4) **Name-based detector** – it detects if the entities or other components mentioned in the issue are the same as these which are in the commit log.
5) **Text-based detector** – it is similar to the previous layer but extracts comments in the changed code to and tries to link them with the issue metadata.
6) **Association-based detector** – it is the last layer which is used if the text used in the texts or entities names cannot be matched with the issue (the texts are not similar). It uses association strengths between the terms in the issue and the entity names.

MLink is better than ReLink because it checks and compares not only terms but changes in the code repository too. It achieves high accuracy level: F-score: 87-93%, recall: 85-90%, precision: 82-97% as outlined in [8].

2.3. RCLinker: Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information

RCLinker's [2] authors discovered, that many commits are not containing relevant information in commit messages. It means that if we want to improve linking issues with commits, we must use other contextual information.

RCLinker uses ChangeScribe [10] to generate additional messages about commits. ChangeScribe adds information in following format (real example from [2]):

This change set is mainly composed of:
1. Changes to package org.apache.solr.common.cloud:
(a) Modifications to ClusterState.java:
i. Remove an unused functionality to get shared

Messages, created by ChangeScribe, are then appended to each commit message. RCLinker also uses other contextual information like commit date, issue update date, issue comments' date.

RCLinker uses machine learning – trained Random Forest. Authors defined 9 text features (which are basing mostly on cosine distance between texts) and 11 metadata features (which are basing mostly on issue, commit and comments dates). We use this features to train Random Forest.

Usage of RCLinker is divided into two phases:

1) Learn phase – extending commit messages with ChangeScribe, extracting features (T1 – T9 and M1 – M11) and training model with i.e. Weka implementation of Random Forest.
2) Production phase – extending commit message with ChangeScribe, extracting features (T1 – T9 and M1 – M11) and using on created model to choose proper issues for commit.

RCLinker is much better in case of very poor developers' commit messages. It has approximately 136 % better results of F-measure than MLink, however precision is lower than in MLink.

We have also checked articles that are citing [2, 8] or [9]. Most of them are not related with linking issues with commits.

Empirical Evaluation of Bug Linking [11] is an empirical evaluation with benchmark of ReLink algorithm. It does not propose any new tool. However this article shows that usage of ReLink is reasonable.

In *The Missing Links: Bugs and Bug-fix Commits* [12] there is an analysis of problems with issue-commit linking. Authors used Linkster tool and expert knowledge to check 493 commits and link them to issues. Despite this work did not propose any new tool or algorithm, it is a good article to understand problems in linking issues with commits.

2.4. When do changes induce fixes?

This article [13] describes one of the simplest algorithms which we use in our approach. In this case it is described how to link bugs from the bug database with commits. This method is quite simple – every commit message is split into a stream of tokens (syntactic analysis). Each token could be one of the items: bug number (it is based on a simple regex), a plain number, a keyword such as fixed, defects etc. and a word. After that, the syntactic confidence is being counted – it is always an integer number between values 0 to 2.

This linking method is based on a semantic analysis too. There is also a score if some of the following conditions were resolved: the bug has been resolved as fixed at least once, the bug description is used in the commit message, the author of the commits has been assigned to it or one or more files affected by the commit has been attached to the bug.

In our approach we use pattern matching and semantic analysis too.

3. Experimental Setup

In this section we want to describe why we have decided to use RCLinker approach. According to *MLink* article [9], MLink is better than ReLink by 6-11% in F-score, 4-13% in recall, and 5-8% in precision. In *RCLinker* article [2] authors sustain that RCLinker has gained far much better results in F-measure by 138.66% in comparison to MLink. That is why we have decided to use RCLinker approach.

3.1. Research questions

- RQ1: How effective RCLinker is in recovering missing links between issues and commits? *In this RQ we will check how effective solutions from literature are.*
- RQ2: Is it possible to pick versatile machine learning settings giving good effects for all kinds of projects? *In this RQ we will check how RCLinker's results can be changed when we adjust classifiers' settings. We will try to check if it is possible to gain better results than original authors.*
- RQ3: Will RCLinker will be effective on projects with various difficulty levels? *We will run defect prediction on datasets that vary on number of issues that could be matched with commit logs.*
- RQ4: How to improve RCLinker algorithm? *We will add new metrics based on Jira metadata and evaluate on various machine learning classifiers.*

3.2. Datasets

The research will be performed using two open source projects by Pivotal Software: Spring Data Redis, Spring OSGi and the proprietary commercial project provided by Capgemini.

To check commits and issue linking we have looked through Git history of selected projects to find out if they contain Jira ticket numbers.

Outlined projects were chosen by discovering Spring's projects catalogue. We chose projects which were created recently. The projects were compared with each other minding commits coverage with Jira tags and overall commits number. This dataset will be split into two parts. First one will be used as training set, second one will have its Jira tags removed and used for validation of output.

Spring OSGi will be used as dataset with higher complexity. This data set is bigger and not fully tagged with Jira issues. Specific thing for Spring OSGi commit history is tagging multiple commits with the same Jira issue number. It is two times bigger than Spring Data Redis – consists of over two thousands commits, while Spring Data Redis of around one thousand.

Commercial project provided by Capgemini will be used in final development of the algorithm. The data set used for the research comes from a system produced for one of the biggest automotive companies in Europe and it covers all aspects of car purchasing. The project is developed using agile methodologies. Support and bug fixing is hierarchically organised using Kanban technique as described in [14].

3.3. Knime workflow description

Our main goal is to implement a new DePress plugin. To supply datasets for the plugin (issues from JIRA and commit logs from GitHub) we needed to prepare workspace and provide links to JIRA and GitHub repository. More detailed information about reproduction (i.e. detailed steps of installation) can be found in Appendix A.

3.4. Metrics – model input

We will use two categories of metrics: based on commit message and based on commit metadata. Metrics are similar to those used in *RCLinker* article [2]. We have used them as a model input – independent predictors. As dependent variable we predict if the given pair commit-issue is a true link or not. For a list of notations, used in metrics table, please see Table 1. We will use metrics described in Table 2.

Metrics J1a, J1b, J1c, J1d, J2a, J2b, J3a, J3b, J3c are metrics designed by us, which are based on JIRA changes and metadata.

Model input consists of two additional indicators: *realLink* – valued as 1 if given pair of commit-issue exists in golden set, otherwise 0, and *undersampled RealLink*, which is output of undersampling process described in *RCLinker* article [2]. Golden set is extracted from version control system repository by traversing all the existing commit descriptions and matching issue tracking IDs in them. If such ID is found in description of commit, it is considered as a part of golden set.

During first phase of experiment, we have learned model using Random Forest. Next, we have tried to improve results using also the following classifiers from Weka library: MultilayerPerceptron, BayesNet, NaiveBayes, SGD, AdaBoostM1, RealAdaBoost and changing default parameters' values to get better results. Unfortunately it did not improve the results, so we have decided to use Random Forest classifier.

Table 1. List of notations used in metrics description

Msg	Human-written commit message
Csmsg	Commit message generated by ChangeScribe
cmtDate	Commit date
summary	Summary of an issue
Desc	Description of an issue
Prio	Priority of an issue
noCom	Number of comments in issue
com_i	Comment (1 <= com*i* <= noCom)
words(D)	Number of distinct words in document D
+	text concatenation
reportedDate	Report date of an issue
updatedDate	Last update date of an issue
date (commi)	Date of i^{th} comment in issue

3.5. Prediction model and measures

During experiment we have used a model to predict if the given issue should be linked with the given commit. For each pair (issue, commit) we have analysed if there is a link between them or not.

As an output of program returns a list of linked issues with commits – pairs *(issue, commit)*. We have evaluated the result in the matter of measures such as precision, recall and F-measure.

4. Results

In this section are presented results which we have achieved by using RCLinker approach without the ChangeScribe tool. In the first two tables can be found results for open source projects, in the last one – for the commercial Capgemini project.

Table 2. Used textual and metadata metrics

	Text features
T_1	cos(summary + desc + COM_i, msg + csmsg)
T_2	$average(cos(I_a, C_b)), I_a \in \{summary, desc, COM_i\}, C_b \in \{msg, csmsg\}$
T_3	$max(cos(I_a, C_b)), I_a \in \{summary, desc, COM_i\}, C_b \in \{msg, csmsg\}$
T_4	T_2/T_3
T_5	T_2/T_1
T_6	$\lvert words(summary + desc + \sum COM_i) \cap words(msg + csmsg)\rvert$
T_7	$\frac{\lvert words(summary+desc+\sum COM_i) \cap words(msg+csmsg)\rvert}{\lvert words(summary+desc+\sum COM_i) \cup words(msg+csmsg)\rvert}$
T_8	$\frac{\lvert words(summary+desc+\sum COM_i) \cap words(msg+csmsg)\rvert}{\lvert words(summary+desc+\sum COM_i)\rvert}$
T_9	$\frac{\lvert words(summary+desc+\sum COM_i) \cap words(msg+csmsg)\rvert}{\lvert words(msg+csmsg)\rvert}$
	Metadata features
M_1	Number of changed files in the commit that are mentioned in the issue text
M_2	$\frac{M_1}{noCom+prio}$
M_3	1 if the issue reporter is also the committer. Otherwise, 0
M_4	1 if the committer posts comments in the issue. Otherwise 0
M_5	1 if the commit date is between the report date and the last updated date of the issue. Otherwise, 0
M_6	cmtDate - reportedDate
M_7	updatedDate - cmtDate
M_8	M_6/M_7
M_9	$\min_{1 \ldots noCom} \lvert cmtDate - date(com_i)\rvert$
M_{10}	$\lvert cmtDate - date(COM_{noCom})\rvert$
M_{11}	$\lvert cmtDate - date(COM_{noCom-1})\rvert$
	JIRA features
J_{1a}	1 if issue status is RESOLVED or CLOSED. Otherwise 0
J_{1b}	1 if issue status is IN PROGRESS. Otherwise 0
J_{1c}	1 if issue status is REOPENED. Otherwise 0
J_{1d}	1 if issue status is OPEN. Otherwise 0
J_{2a}	1 if commit and issue are similar (classes from commit and from issue description). Otherwise 0
J_{2b}	1 if commit and comment are similar (classes from commit and classes from comment - for example stacktraces etc.). Otherwise 0
J_{3a}	1 if assignee from the issue is the same as the commiter. Otherwise 0
J_{3b}	1 if issue reporter is the same as the commiter. Otherwise 0
J_{3c}	1 if author of the issue comment is the same as the commiter. Otherwise 0

4.1. Spring Data Redis

Results for Spring Data Redis project are presented in Table 3.

Table 3. Spring Data Redis evaluation results

Description	**Recall**	**Precision**	**F-measure**
no cross validation, 2 nearest neighbours	0.63	0.17	0.27
10 iterational cross validation, 10 nearest neighbours	0.52	0.10	0.16
10 iterational cross validation, 2 nearest neighbours	0.48	0.38	0.42
15 iterational cross validation, 2 nearest neighbours	0.46	0.37	0.40
10 iterational cross validation, 1 nearest neighbour	0.41	0.54	0.47
10 iterational cross validation, 0 nearest neighbours	**0.38**	**0.79**	**0.51**

The best achieved results processing Spring Data Redis project were recall: 0.38, precision 0.79 giving F-measure at point of 0.51.

During evaluation it turned out that producing nearest neighbours actually does not impact results in the positive way. It makes recall slightly rise, but with cost of huge precision drop.

Using cross validation instead of random splitting data set into two fixed-size subsets improved the result. When no cross validation was used, with two nearest neighbours generated, there were F-measure equal to 0.27. With the same nearest neighbour setting and cross validation used, the F-measure raised to level of 0.42.

Increasing number of cross validation iterations did not bring significant improvement comparing to extended computation time needed to process data. Adding 5 iterations enhanced F-measure by 0.02.

4.2. Spring OSGi

Results for Spring OSGi project are presented in Table 4.

Table 4. Spring OSGi evaluation results

Description	Recall	Precision	F-measure
10 iterational cross validation, 1 nearest neighbour	0.19	0.22	0.20
10 iterational cross validation, 0 nearest neighbours	0.17	0.58	0.26
15 iterational cross validation, 0 nearest neighbours	**0.19**	**0.60**	**0.29**

In comparison to Spring Data Redis, Spring OSGi has far more less correct commits descriptions. While Spring Data Redis has almost all of them well described, Spring OSGi has more or less 50%. That is why, the results are much worse. Slightly better result we got by increasing the number of iterations.

4.3. Capgemini project

Results for the commercial Capgemini project are presented in Table 5.

Table 5. Capgemini project evaluation results

Description	Recall	Precision	F-measure
10 iterational cross validation, 2 nearest neighbours	0.58	0.41	0.48
10 iterational cross validation, 1 nearest neighbour	0.54	0.56	0.55
15 iterational cross validation, 0 nearest neighbours	0.49	0.86	0.62
10 iterational cross validation, 0 nearest neighbours	**0.49**	**0.88**	**0.63**

Proprietary commercial project provided by Capgemini has quite good results. First of all the dataset with commits and issues was not too big – this was a period of 6 months. The other thing why results are good is caused by good described commits' messages – about 95% has good commit description.

In comparison to Spring Data Redis – the best results were achieved when nearest neighbours equalled 0. The recall and F-measure raised significantly: recall from 0,56 to 0,88 and F-measure from 0,55 to 0,63.

5. Discussion

In this section, we have described why we have not used ChangeScribe in our implementation of the RCLinker algorithm. ChangeScribe caused many performance and implementation problems which are described below.

5.1. General discussion

As we can see in the results, RCLinker algorithm performs the best in Capgemini proprietary project. Also results for the Spring Data Redis are still quite good, however they are much worse than in the original RCLinker [2] approach.

5.2. Problems

During implementation of RCLinker algorithm we have encountered many performance and implementation problems.

First problem was with executing ChangeScribe in non-eclipse environment. ChangeScribe was not describing properly all changes. We wrote to ChangeScribe's authors and created an issue on the GitHub repository. They helped us and gave access to special, modified version of ChangeDistiller.

Second problem was memory complexity of ChangeDistiller. We have tried to run application with various heap sizes, however even 15 GB of RAM was not enough.

5.3. Validity threats

Golden set is extracted from version control system repository by pattern matching potential issue IDs in commits' description. This is a threat to validity since there may be some mistakenly tagged descriptions and the golden set acquired this way may not be full. There is no other fully credible way of achieving such a golden set in projects evaluated by us. Manual creation of golden set would be extremely time consuming and would not plausibility of it would be arguable as well.

6. Conclusions

Connections between issues and commits are very valuable in defect prediction. Unfortunately commit logs are often missing clear disclosure of these links.

Our implementation of RCLinker was able to achieve results with F-measure equal to 0.62 on the commercial project. This is promising result, but is not enough for enterprise use of this tool. The result indicates need for further development of the algorithm itself.

Due to problems with ChangeScribe results are inconclusive. Comparing to the RCLinker paper our implementation achieves significantly worse results. There is a possibility, that using ChangeScribe, the results would be comparable to original RCLinker's evaluation.

We wanted to check which features are significant and important for the results. T1, T2 and T3 were good indicators when it comes to textual relevance between commits and issues. Features T4 and T5 are normalized forms of T1-T3 respectively and that is why we supposed that they may be not very essential — we have checked this assumption using different classifiers described below. Features T6-T7 were used to compute the number of common words between issue and commit bringing new information to the classifier so they are relevant for classifier. Because T8 and T9 are the ratio of T6 to the number of distinct word in issue and commit we consider them as not useful. Metadata features M9, M10, M11 were based on the comments and dates between them and did not provide valuable information for machine learning algorithms. After evaluation we consider JIRA features J1a, J1b, J1c, J1d, J2a, J2b, J3a, J3b, J3c as not relevant, as they were not improving result of machine learning.

Concluding the revision we decided to leave significant metrics T1, T2, T3, T6, T7, M1-M8 and not use features: T4, T5, T8, T9, M9, M10, M11, J1a, J1b, J1c, J1d, J2a, J2b, J3a, J3b, J3c. We have tested it on the Spring Data Redis dataset. The results were comparable: Recall: 0,36, Precision: 0,79, F-measure: 0,49. With the previous features set we got: Recall: 0,38, Precision: 0,79 and F-measure: 0,51.

We have also tested our implementation using the following classifiers from Weka library: MultilayerPerceptron, BayesNet, NaiveBayes, SGD, AdaBoostM1, RealAdaBoost and Random Forest. However, Random Forest, used in original paper [2], gave us the best results. We have checked how various parameters will change the results. For Random Forest, the best parameter set is: Max Depth: unlimited, number of trees: 10.

7. Future works

We were not able to gain such a good results as described in the article about RCLinker [2]. We suppose that a tool which will be similar to ChangeScribe can improve the results. Additional features which were based on JIRA metadata did not improve the results. It is likely that a tool which generates additional messages about commits will give significant information about changes in the repository code — it will be possible to create a new set of features. The decision to create a new tool instead of using ChangeScribe is associated with the problems described in the subsection 5.2.

References

[1] K. Herzig, S. Just, and A. Zeller. It's Not a Bug, It's a Feature: How Misclassification Impacts Bug Prediction, In: Proceedings of the 2013 International Conference on Software Engineering, ICSE '13, Piscataway, NJ, USA, pp. 392–401, IEEE Press, 2013.

[2] T.-D. B. Le, M. Linares-Vásquez, D. Lo, and D. Poshyvanyk, RCLinker. Automated Linking of Issue Reports and Commits Leveraging Rich Contextual Information, In: Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC '15, Piscataway, NJ, USA, pp. 36–47, IEEE Press, 2015.

[3] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A General Software Defect-Proneness Prediction Framework, IEEE Transactions in Software Engineering, Vol. 37, pp. 356–370, 2011.

[4] L. Madeyski and M. Jureczko. Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study, Software Quality Journal, Vol. 23, no. 3, pp. 393–422, 2015.
[5] M. Jureczko and L. Madeyski. Cross–project defect prediction with respect to code ownership model: An empirical study, e-Informatica Software Engineering Journal, Vol. 9, no. 1, pp. 21–35, 2015.
[6] L. Madeyski and M. Majchrzak. Software Measurement and Defect Prediction with De-Press Extensible Framework, Foundations of Computing and Decision Sciences, Vol. 39, no. 4, pp. 249–270, 2014.
[7] C. Wohlin. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering, In: Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering, EASE ’14, New York, NY, USA, pp. 38:1–38:10, ACM, 2014.
[8] 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, ESEC/FSE ’11, New York, NY, USA, pp. 15–25, ACM, 2011.
[9] 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 the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering, FSE ’12, New York, NY, USA, pp. 63:1–63:11, ACM, 2012.
[10] ChangeScribe, https://github.com/SEMERU-WM/ChangeScribe, 2016.
[11] T. F. Bissyandé, F. Thung, S. Wang, D. Lo, L. Jiang, and L. Réveillère. Empirical Evaluation of Bug Linking, In: 17th European Conference on Software Maintenance and Reengineering (CSMR), pp. 89–98, 2013.
[12] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein. The Missing Links: Bugs and Bug-fix Commits, In: Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE ’10, New York, NY, USA, pp. 97–106, ACM, 2010.
[13] J. Sliwerski, T. Zimmermann, and A. Zeller. When Do Changes Induce Fixes?, In: Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR ’05, New York, NY, USA, pp. 1–5, ACM, 2005.
[14] M. Majchrzak and L. Stilger. Experience Report: Introducing Kanban Into Automotive Software Project, From Requirements to Software: Research and Practice, Scientific Papers of the Polish Information Processing Society Scientific Council, pp. 15–32, 2015.