Preprint: Jarosław Pokropiński, Jakub Gasiorek, Patryk Kramarczyk, and Lech Madeyski, "SZZ Unleashed-RA-C: An Improved Implementation of the SZZ Algorithm and Empirical Comparison with Existing Open Source Solutions", pp. 181–199. Cham: Springer International Publishing, 2022. DOI: 10.1007/978-3-030-77916-0_7

Preprint: https://madeyski.e-informatyka.pl/download/PokropinskiEtAl22.pdf

SZZ Unleashed-RA-C: An Improved Implementation of the SZZ Algorithm and Empirical Comparison with Existing Open Source Solutions

Jarosław Pokropiński, Jakub Gąsiorek, Patryk Kramarczyk, Lech Madeyski (✉)

Abstract SZZ algorithm is one of the most important algorithms in mining software defects as it allows to create data sets for the sake of software defect prediction. Unfortunately, still very few open source implementations of this algorithm were created. In recent years two interesting open source implementations of SZZ algorithm have been created, which are SZZ Unleashed and OpenSZZ. In this paper we compare how well these implementations perform, as well as propose an improved implementation named SZZ Unleashed-RA-C. The most important features of the proposed algorithm and implementation include: ability to identify and handle refactoring changes when tracing bug-introducing changes (RA functionality), discarding comments and files based on a regular expression, and last but not least the ability of using GitHub as the issue tracker.

1 Introduction

Information about the root cause of a bug and when it was introduced are often missing from issue tracking software. Research in the area of mining software repositories often relies on detailed bug information. Extending the data stored in issue trackers could be of a great value both for researchers and software developers.

Jarosław Pokropiński
Wroclaw University of Science and Technology, Poland, e-mail: 236519@student.pwr.edu.pl

Jakub Gąsiorek
Wroclaw University of Science and Technology, Poland, e-mail: 220890@student.pwr.edu.pl

Patryk Kramarczyk
Wroclaw University of Science and Technology, Poland, e-mail: 257413@student.pwr.edu.pl

Lech Madeyski
Wroclaw University of Science and Technology, Poland, e-mail: lech.madeyski@pwr.edu.pl

One of the most known algorithms used for identifying bug-inducing changes is SZZ proposed by Śliwerski et al. (2005). Its main purpose is to extend the bug report data with commit that first introduced the bug. The SZZ algorithm consists of 2 steps:

- Identification of bug fixing commits using version control system (e.g., git) and issue tracking software (e.g., Jira).
- Identification of a commit causing the bug.

Researchers often rely on the SZZ algorithm to identify bug-introducing changes. Unfortunately, only a few of the SZZ implementations are publicly available. Two of the most popular ones introduced recently are: SZZ Unleashed made by Borg et al. (2019) and OpenSZZ by Lenarduzzi et al. (2020). It is difficult to say which of these implementations produces better results based on the research papers alone, so our aim is to compare them using a validated Defects4J data set specifically prepared for evaluating SZZ implementations and introduced by Neto et al. (2019).

We attempt to improve one of the available open source SZZ implementations with our own ideas, as well as ideas from existing literature.

Our contributions in this paper are as follows:

- Literature review of SZZ publications and open source implementations.
- Extracting a list of improvements to the basic SZZ algorithm on a basis of literature and our own solutions.
- An attempt to propose a new implementation of SZZ algorithm combining improvement ideas (own and existing in literature).
- Empirical comparison of the found SZZ implementations as well as the new one on the same data set.

The remainder of this paper is structured as follows. Section 2 contains a brief overview of the SZZ algorithm, existing implementations found in literature and evaluation of two open source SZZ implementations. Section 3 describes the ways in which topics including validation data set preparation and SZZ algorithm performance evaluation were conducted. In Section 4, we present results of experiments. In Section 5, we answer given research questions and present threats to their validity. Finally, in Section 6, we propose further research, while in Section 7, we draw conclusions.

2 Literature Review

In this section we introduce the SZZ algorithm, discuss and compare existing SZZ algorithm implementations, and pose research questions.

2.1 The SZZ Algorithm

The SZZ algorithm is the most commonly used algorithm for finding bug-introducing changes. Initially, it was developed for the SVN version control system, but has since evolved for repositories using git.

The algorithm consists of two steps. In the first step, SZZ tries to find a bug fixing commit, based on references to bug reports or commit messages containing words like "fix". Modified lines in the source code are then extracted from bug-fixing commits.

Step two is the identification of bug inducing changes. The SZZ algorithm uses the blame functionality of the version control system to determine all commits that previously made changes to the same lines of code as bug-fixing commits. These commits are then labelled as potential bug-introducing commits.

SZZ then determines whether these potential bug-introducing commits can be ruled out as bug-introducing or not. Each potential bug-introducing candidate has its commit date compared to the submission date of a corresponding bug report. All candidates, that took place before the creation of the report are considered as bug-introducing. If the commit time is after the bug report submission time, then the candidate is still a suspect, because it could be bug-introducing. This can happen if the change is a partial fix, or it is inducing another bug.

2.2 Existing SZZ Algorithm Implementations

Researchers developed their own versions of the SZZ algorithm. One of them is an SZZ implementation proposed by Neto et al. (2018) which is called the refactoring aware SZZ (RA-SZZ). This implementation introduces ability to identify and handle refactoring changes when tracing bug-introducing changes. RA-SZZ was then compared with another implementation named meta-changes aware SZZ (MA-SZZ) proposed by da Costa et al. (2017) and the original SZZ by Śliwerski et al. (2005).

Refactoring changes detection is usually based on two tools: RefDiff by Silva et al. (2020) and RefactoringMiner by Tsantalis et al. (2018). Comparison of these two tools shows that their precision and recall are similar, with a slight advantage towards RefactoringMiner. We will try to incorporate detection of refactoring changes by utilising the more accurate tool which is RefactoringMiner.

Neto et al. (2019) points out that the overall accuracy of SZZ algorithm increases by 40% if only valid bug-fix lines are used as the input for SZZ.

To the best of our knowledge, the only two open implementations of the SZZ algorithm referenced from literature are SZZ Unleashed[1] by Borg et al. (2019) and OpenSZZ[2] by Lenarduzzi et al. (2020). These two open source SZZ implementations

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

[2] `https://github.com/clowee/OpenSZZ`

have concise `readme` files and an active community (reflected by the amount of GitHub stars and forks, higher for SZZ Unleashed), compared to other repositories.

2.3 Existing implementations comparison

The ultimate goal of our publication was to choose one of the existing implementations of the SZZ algorithm and improve it. As mentioned in Section 2.2 there were two most promising open source implementations. Both of those were tested and the results were compared.

2.3.1 OpenSZZ project evaluation

The OpenSZZ project can be found in two versions: OpenSZZ as a standard Java project and a cloud native version with a surrounding docker infrastructure prepared. The cloud native version was developed after the simple one and has a few differences. At first, let us focus on the non functional aspect of the project. The primary difference between the two versions is the architecture. The so called cloud native version is using the microservices design pattern and the RabbitMQ message broker for the internal communication. One of the biggest issues with it is the log handling of the core SZZ service. By default the logs are forwarded to a few files in the container file system. Unfortunately, for most of the project processing, the used file writer flushes the data only after most of the processing is done. This results with user not being able to track most of the progress in the processed project. SZZ algorithms are known to take a lot of time, especially for highly developed projects. Having any indicator of whether the application is functional is important, especially in those used for research purposes. This leads to another problem the OpenSZZ project has. At first, the project was evaluated with the *commons-bcel* (this project was presented as an example in OpenSZZ) and *unomi* repositories. No issues occurred while running those. Later the *syncope* and *commons-math* projects were run. Unfortunately the runs were unsuccessful and the very basic logging did not give any indicators as to what could be the the cause. The application simply dropped its resource usage and kept running. Further investigation allowed to detect, that the application has ran out of memory. Increasing the heap size allowed the *commons-math* to succeed. 12GB allowed *syncope* to process more data, but was not enough to finish the project processing. Besides the presented issues, the cloud version was easy to set up and work with. OpenSZZ encapsulates the whole SZZ process into a single service, which allows the user to easily start the processing without additional manual steps being required.

The functional performance of the project was evaluated on *commons-bcel* and *unomi* projects. Initial results show that the algorithm is highly susceptible to major refactoring commits. Many resulting bug introducing commits were a major commit changing dozens of thousands lines of code. OpenSZZ also produces pairs that do not have any common files changed between them. This can be observed between *45da20f49abafa125ff4f616e8312b89fbd1f139* (bug introducing commit) and *4d89da4f52f6ae26a4917ba79259e8c89c67eb77* (bug fixing commit) revisions in the *commons-bcel* repository. OpenSZZ attributed the bug introducing change to *src/main/java/org/apache/bcel/classfile/Attribute.java* file which was not changed in the bug introducing commit. Knowing current SZZ algorithm limitations it is highly unlikely for such a situation to be correctly detected. OpenSZZ has found 249 bug introducers in this repository in 3m30s. After discarding issues that were major refactoring commits or any other commits that were unsuccessfully matched to at least ten bug fixing commits this number drops to 24.

2.3.2 SZZ Unleashed project evaluation

Starting with the non functional performance of the SZZ Unleashed implementation it should be noted that it is separated into a few steps. Each of which needs to be run manually. The same repositories were processed without any issues on default settings. We noticed that for bigger projects such as *syncope*, where the processing time took hours the job partitioning between threads was not perfect. Often half of the available threads have finished work within 1-2 hours while others required few more hours to process all the issues.

Running the *commons-bcel* repository took 6 minutes and produced 708 bug introducer and bug fixer pairs. A huge disadvantage of the SZZ Unleashed project is its output. By default it is a JSON array of arrays containing two strings - commit hashes. Internally the algorithm recognises the file in which the bug fix is placed which results in a huge count of duplicates. Another interesting issue that was noticed is that a single changed line in a change log file (in *commons-bcel* - it is called *changes.xml*) was attributed both to around 80 bug fixing commits and 120 bug inducing ones. Furthermore, the algorithm was susceptible to huge refactoring commits. Creating a list with a count of bug introducing commits assigned to each bug fixing commit shows that there are many issues fixing dozens of commits. Such a case is possible, but highly unlikely to occur. It indicates a higher possibility of the bug fixing commit being incorrectly matched. SZZ Unleashed often detected commits containing changes to the comments only. This can be noticed in this pair from the *unomi* project *0ffc0814f4ff4288b591407afdb0679358249bc* (bug fixing commit), *1d075ec19850466a355ecffc1dfed2da049e25c9* (bug introducing commit). After discarding most common invalid bug inducing commits the count of issues dropped to 458. The same action for bug fixing commits resulted in 468 pairs. After discarding both, the count dropped to 220 and after removing duplicated pairs that could not be validated it was equal to 86.

2.3.3 OpenSZZ and SZZ Unleashed comparison

The major differences between those two projects are the time and memory complexity. OpenSZZ is much faster on bigger repositories, but requires much more memory to process them. OpenSZZ has a friendlier interface for working with it while SZZ Unleashed requires more manual steps. During this process we had an opportunity to get familiar with the internal code base of both of them and our subjective opinion is that the code quality was better in SZZ Unleashed.

Both of the implementations have their constraints and neither of them by default produces reasonably valid results. There is a lot of room for improvement. The performed evaluation resulted in the following improvements that could be implemented:

1. Making SZZ refactoring aware
2. Considering only specified file extensions (ex. ".java").
3. Disregarding deleted lines matching specified pattern.
4. Disregarding fixing and bug-introducing commit pairs if the time between them is greater than 2 years.
5. Adding GitHub issues support as an issue tracker.

Limiting algorithm to `.java` files addresses the problem where configuration files are matched as introducers to fixes that are made in code. Disregarding deleted lines allows to ignore lines that do not contain bug. We wanted to validate impact of time between commits on matching bug introducing commits. As was pointed out in paper by da Costa et al. (2017), it is unlikely, that bug-introducing changes in a project introduce bugs that took years to be discovered. We tried disregarding fixing and bug-introducing commit pairs if the time between them is greater than 2 years. Lastly, to bring SZZ to greater number of projects we proposed to extend existing implementations by adding support for GitHub issues.

2.4 Research Questions

The aim of this paper is to compare existing SZZ implementations using a validated data set and build upon and improve the algorithm which produces better results out of the box. Therefore, we address the following research questions (RQs):

- **RQ1: Which of the two open source implementations (OpenSZZ vs SZZ Unleashed) produces results with higher recall using our data set?**
  We want to compare this two implementations against the validated Defects4J data set by Neto et al. (2019). However, we believe that original SZZ Unleashed contains an error causing the performance to drop significantly which is fixed by a pull request on GitHub: `https://github.com/wogscpar/SZZUnleashed/pull/32`. That is why we have added another sub-RQ:
  - **RQ1.1: Is the available fix for SZZ Unleashed valid?**

- **RQ2: How does detecting and discarding refactoring changes influence the overall recall?**
  We analyse how does adding RefactoringMiner by Tsantalis et al. (2018) affect the performance.
- **RQ3: How does adding our own proposed improvements affect the recall of the selected algorithm?**
  We want to investigate the impact of each individual improvement on the previously selected SZZ implementation (from RQ1).

3 Methods and Materials

Data set of bugs, as well as the compared SZZ implementations are described in this section.

3.1 Bug data set

To automate comparison of SZZ algorithms we needed a data set of bugs. We chose data set published in Neto et al. (2019), as it was created with evaluation of SZZ algorithms in mind. It consists of data such as bug fixing commit id, bug inducing commit id, path to bug fix and more. Since tested algorithms use git as version control system, and given data set also used SVN, we had to modify the data. Two repositories, *commons-math* and *commons-lang*, were migrated from SVN to git so we modified data set by replacing their SVN revision identifiers with respective git commit hashes. After that, the only remaining project was *jfreechart*, which does not use labels in GitHub issues, so it was removed from the data set. Resulting data set consisted of five projects: *Apache commons-math*, *Apache commons-lang*, *mockito*, *JodaOrg joda-time* and *Google closure-compiler* with corresponding bug fixing git commit hash, bug inducing git commit hash, path to bug fix and additional information. In resulting data set, all repositories were hosted on GitHub. Two repositories: *Apache commons-math* and *Apache commons-lang* used Jira as a issue tracker, while the rest used GitHub issues.

3.2 SZZ Unleashed and OpenSZZ comparison

To compare OpenSZZ and SZZ Unleashed we cloned their repositories and followed instructions in their `README.md` files. To make better comparison, we modified SZZ Unleashed so its output contained information about path of the fix. We ran OpenSZZ, and then SZZ Unleashed on *Apache commons-math* and *Apache commons-lang*. After we got results from both SZZ implementations we filtered them, so they held results only for fixes contained in our data set and then we compared their results with the data set. For each repository and implementation, we counted distinct results consisting of bug fix, bug introducer and bug fix path that appear both in results and our data set. With this data we evaluated performance of implementations using the recall measure.

3.3 Base SZZ algorithm choice

Rodriguez et al. (2018) noticed an issue that most of the researchers studying topics related to the SZZ algorithm tend to create their own implementation from scratch. We agree with this statement and wanted to use an existing solution as a base for our improvements. During the literature review two most promising candidates were chosen: SZZ Unleashed and OpenSZZ.

This process was started by running both of those implementations on repositories *commons-bcel* which by default was used as an example by OpenSZZ and *unomi*. The results were later on analysed by a script that compared both of them. The output of the mentioned script contained information about the count of bug introducer and bug fixer pairs that both implementations have in common, the counts without duplicates and the number of results for each of them: both with and without duplicates.

The next step was to perform additional manual validation of results. During this process we noticed that most of the bug introducers are not valid due to the commits being major releases or refactorings with dozens of thousands of changed lines that had no connection to the bug fix. As this concerned most of the commits in results, we decided to redo the previous steps with those commits filtered out. This resulted in much clearer output and more readable data.

As for the final decision both the steps mentioned above and those from Section 3.2 were used to determine it.

3.4 Github issues

The bug data set described in Section 3.1 contained only two repositories that were using JIRA as an issue tracker. This was the main motivation to extend the used SZZ implementation to allow fetching issues from GitHub issue tracker. An alternative strategy for fetching issues was developed and attached to the scripts handling the first stage of the SZZ algorithm.

3.5 SZZ improvements implementation

After choosing the base for SZZ implementation we implemented our proposed improvements:

1. The first improvement (1) was to make SZZ refactoring aware. We used Refactoring Miner by Tsantalis et al. (2018) to mine repositories for refactorings and ignored lines that were refactorings while building line mapping graph.
2. The second improvement (2) was to run SZZ only on changes in files that contained code as opposed to configuration files. We implemented it by using pattern for files that contained code and ignored all files that did not match it. As we used Java projects we set pattern to `.*\.java`.
3. Another improvement (3) was to ignore line deletions that were deletions of comments while building line mapping graph. To do this we used pattern `\s*\/\/.*|\s*\*.*|\s*\/\*.*` and ignored deletions that matched that pattern.

3.6 SZZ improvements comparison

The article introduces a few possible improvements to the SZZ algorithm. Each of those improvements needs analysis of its performance. For that, we used the data set prepared in Section 3.1 and formula from Section 3.2.

For the SZZ Unleashed implementation, as the created data set can be considered the only used source of truth for the validation, we decided to limit the input data for the bug introducer detection only to those that it contained. This allowed us to get much smaller processing times and clearer results. An additional advantage was that the reproducibility of end results has improved. Having done that, the received issues were used in the second step of the SZZ algorithm. This step was repeated for each improvement that was developed.

Results were analysed with a formula mentioned above. In addition to that the processing time was measured. It should be noted that for the SZZ Unleashed it is smaller due to processing of a limited amount of issues.

It is also worth mentioning that we did not test version that limited SZZ to `.java` files because our data set consists only of such files.

3.7 SZZ Unleashed fix impact

During the development an open pull request was noticed in the SZZ Unleashed GitHub repository. Interestingly, its title contained the *fatal bug* phrase. The fix concerned a variable used as a list iterator. The first step of the impact analysis of the potential bug was estimated by a manual review of the code it related to. Afterwards, the fixed version was run just as others described in Section 3.6. Both those steps were enough to make a final decision about whether it was valid.

4 Results

Firstly we observed, as seen in Table 1, that all tested implementations produced bad results for `closure-compiler` so we omitted them in further experiments. The issue with this repository concerns the lack of issues from the GitHub issue tracker that are present in the used data set. Conducted research focuses mostly on the second step of the SZZ algorithm and this repository produces no data that could be supplied to it.

Table 1 Closure compiler results

algorithm version	repository	time	matches	size data	size results
SZZ Unleashed	closure-compiler	0s	0	124	0
SZZ Unleashed fixed	closure-compiler	0s	0	124	0
SZZ Unleashed-RA	closure-compiler	0s	0	124	0
SZZ Unleashed-C	closure-compiler	0s	0	124	0
SZZ Unleashed-T	closure-compiler	0s	0	124	0

Table 2 SZZ Unleashed and OpenSZZ comparison results

algorithm version	repository	time	matches	size data	size results
SZZ Unleashed	commons-lang	11m 54s	8	64	150
SZZ Unleashed	commons-math	15m 55s	28	107	238
OpenSZZ	commons-lang	9m 18s	0	64	7
OpenSZZ	commons-math	15m 42s	0	107	0

Using data in Table 2, we measured the performance of SZZ Unleashed ($\frac{8+28}{150+238} = 0.0928$, i.e., 9.28%) and performance of OpenSZZ (0%) on these two repositories on the given data set. With these results we decided to use SZZ Unleashed as a base for improvements.

4.1 SZZ Unleashed fix impact results

To the best of our knowledge, original SZZ Unleashed algorithm contains a bug described in Section 3.7. A proper analysis was performed to validate its impact.

Using data in Table 3, we measured that performance of SZZ Unleashed is 8.19% and performance of SZZ Unleashed with fix is 14.21%.

Table 3 SZZ Unleashed fix impact results

algorithm version	repository	time	matches	size data	size results
SZZ Unleashed	commons-lang	16s	8	64	150
SZZ Unleashed fixed	commons-lang	6s	34	64	188
SZZ Unleashed	commons-math	20s	28	107	238
SZZ Unleashed fixed	commons-math	1m 1s	52	107	357
SZZ Unleashed	mockito	3s	5	59	82
SZZ Unleashed fixed	mockito	2s	10	59	119
SZZ Unleashed	joda-time	1s	3	29	67
SZZ Unleashed fixed	joda-time	2s	6	29	54

That result and analysis of the part of the code containing described bug gave us enough confidence to assume that proposed fix is valid and further tests were performed on the fixed version.

After performing our research, the proposed fix (`https://github.com/wogscpar/SZZUnleashed/pull/32`) was merged into the repository, which means that our assumptions were correct in the context of the results we have got.

4.2 Proposed improvements impact results

In this section we present the impact of our proposed improvements, which are as follows:

- Making SZZ refactoring aware (RA) (1).
- Considering only specified file extensions (ex. ".java") (2).
- Disregarding deleted lines matching specified pattern (3).
- Disregarding fixing and bug-introducing commit pairs if the time between them is greater than 2 years (4).
- Adding GitHub issues support as an issue tracker (5).

Table 4 SZZ Unleashed proposed improvements impact results

algorithm version	repository	time	matches	size data	size results
SZZ Unleashed fixed	commons-lang	6s	34	64	188
SZZ Unleashed-RA	commons-lang	40s	34	64	186
SZZ Unleashed-C	commons-lang	5s	33	64	114
SZZ Unleashed-T	commons-lang	5s	17	64	81
SZZ Unleashed-RA-C	commons-lang	33s	33	64	106
SZZ Unleashed fixed	commons-math	1m 1s	52	107	357
SZZ Unleashed-RA	commons-math	20m 41s	52	107	349
SZZ Unleashed-C	commons-math	1m 39s	52	107	308
SZZ Unleashed-T	commons-math	49s	40	107	266
SZZ Unleashed-RA-C	commons-math	20m 59s	52	107	302
SZZ Unleashed fixed	mockito	2s	10	59	119
SZZ Unleashed-RA	mockito	45s	10	59	277
SZZ Unleashed-C	mockito	45s	10	59	265
SZZ Unleashed-T	mockito	43s	5	59	176
SZZ Unleashed-RA-C	mockito	44s	10	59	251
SZZ Unleashed fixed	joda-time	2s	6	29	54
SZZ Unleashed-RA	joda-time	9s	6	29	54
SZZ Unleashed-C	joda-time	2s	6	29	47
SZZ Unleashed-T	joda-time	2s	0	29	30
SZZ Unleashed-RA-C	joda-time	6s	6	29	47

where:
SZZ Unleashed fixed - SZZ Unleashed with the fix applied,
SZZ Unleashed-RA - "Refactoring Aware" version of SZZ Unleashed with RefactoringMiner added,
SZZ Unleashed-C - SZZ Unleashed version disregarding deleted lines which are comments. The *C* letter stands for *comments*. SZZ Unleashed-T - SZZ Unleashed version disregarding fixing and bug-introducing commit pairs if the time (T) between them is greater than 2 years,
SZZ Unleashed-RA-C - version containing both RefactoringMiner and disregarding deleted lines matching specified pattern.

Table 4 shows the performance of the base version of SZZ Unleashed with fix is 14.21% on 4 repositories. Our proposed improvements affect the results as follows:

- SZZ Unleashed-RA - 14.41% performance,
- SZZ Unleashed-C - 17.75% performance,
- SZZ Unleashed-T - 13.66% performance,
- SZZ Unleashed-RA-C - 18.20% performance.

5 Discussion

We received no results for any of the SZZ implementation for `closure-compiler`. That situation seems to be caused in the first phase of the SZZ algorithm and might suggest that GitHub implementation of this phase should be improved. It is worth mentioning that this result would be useful when improving the first phase of the algorithm. Since we did not implement such improvements as it was the only project with issues and others using GitHub as issue tracker did return valid results, we decided that those results bring little to comparison and we omitted them in further experiments.

RQ1: Which of the two open source implementations (OpenSZZ, SZZ Unleashed) produces better results?

Answer to RQ1

While comparing SZZ Unleashed and OpenSZZ we observed that for bug fixes in our data set OpenSZZ generated only 7 results for `commons-lang` and none for `commons-math`, while all results were incorrect which led to its performance being evaluated at 0%. Performance of SZZ Unleashed was 9.28% so its performance is better than OpenSZZ. It is worth mentioning, that there is a large space for improvement.

RQ1.1: Is the available fix for SZZ Unleashed valid?

Answer to RQ1.1

Version of SZZ Unleashed with fix performed about 1.74 times better which is a concrete proof that the fix is valid.

RQ2: How does detecting and discarding refactoring changes influence the overall performance?

Answer to RQ2

Adding information about refactoring changes caused the performance of the algorithm to increase from 11.65% to 11.79%. However this comes with a disadvantage of RefactoringMiner only being able to detect refactoring of Java projects, which limits us to those type of projects in the future.

RQ3: How does adding our own proposed improvements affect performance of the selected algorithm?

Answer to RQ3

Results show that our proposed improvements on their own positively affect the performance, apart from the only case of the SZZ Unleashed-T version, which caused the performance to drop. Overall, the best performing version of the algorithm was SZZ Unleashed-C, which discards lines matching specified pattern. It caused the performance to increase from 11.65% to 13.89%. However, combining two versions - SZZ Unleashed-RA and SZZ Unleashed-C - led to even greater improvements, with the performance of **SZZ Unleashed-RA-C** increased to 14.16%.

5.1 Threats to validity

Neto et al. (2019) created data set using bugs from Defects4J (Just et al., 2014) database. Defects4J maintainers create this database with strict rules for each bug. Each bug in database has the following properties:

1. Issue filed in the corresponding issue tracker, and issue tracker identifier mentioned in the fixing commit message.
2. Fixed in a single commit.
3. Minimised: the Defects4J maintainers manually pruned out irrelevant changes in the commit (e.g., refactorings or feature additions).
4. Fixed by modifying the source code (as opposed to configuration files, documentation, or test files).
5. A triggering test exists that failed before the fix and passes after the fix – the test failure is not random or dependent on the test execution order.

Due to the above mentioned constrains, experiments on this data set may give results not reflecting the real world conditions. Another threat to validity of this research is the number of projects available in our data set. Since `closure-compiler` was omitted in the experiments, we ended up with only four repositories and the OpenSZZ was evaluated (in automated way) only on two projects as it is only compatible with JIRA issue tracker.

As one of our proposed improvements was to include only specified file extensions in the process of detecting the bug introducing commits, we have performed our experiments only on Java projects and "*.java" files. The addition of RefactoringMiner also limits us to Java projects only. It is possible to run the algorithm on other projects, by not using the RefactoringMiner and running the program without the "-fp" flag, which excludes files based on specified file pattern, but it has not been tested.

6 Future research

One of the limitations of the improved version is the GitHub issue fetcher. It is possible that the lack of direct interpretation of the id, which is a number, might affect the output of bug fixing commits detection. The issue id has the same format as the one of a GitHub pull request. Merging a pull request usually creates a commit with message containing its id and matching the regular expression used for detecting bug fixing commits by issue id: *"#ID"*.

SZZ uses annotation graphs, which (as claimed by Williams and Spacco (2008a)) are imprecise at tracking lines across large hunks of modified lines. One possible solution to this issue would be to replace them with line-number mappings proposed by the same authors (Williams and Spacco (2008b)).

To better evaluate the SZZ implementations we would need a bigger data set of bugs containing code that is not limited to Java only.

What is also worth exploring is the time between bug-fixing and bug-introducing commits and finding a perfect time frame, where the results are still valid, and no correct pairs are rejected.

7 Conclusions

The publication presents improved versions of the SZZ Unleashed algorithm including the most promising SZZ Unleashed-RA-C. Few of the most important features are: refactoring awareness (RA functionality), discarding comments and files based on a regular expression and the possibility of using GitHub as the issue tracker. Those changes result in a much better performance compared to the base version. Nevertheless, there is still a large space for improvement. The paper introducing the original SZZ algorithm (Śliwerski et al., 2005) received a huge number of citations and was awarded as the most influential paper at MSR conference in 2015. Hence, we believe that the improved version would provide benefits for many researchers.

8 Appendix: Research reproduction

This section presents steps required to reproduce the presented research. Details, code and data are available at `https://github.com/pwr-pbrwio/PBR20M1/blob/master/reproduction.md`.

8.1 Dependencies installation

Dependencies include:

- git
- java 8
- python 3

8.1.1 Dependencies installation on windows and macos

Download and install dependencies:

1. git: `https://git-scm.com/`
2. java: `https://www.oracle.com/java/technologies/javase-jre8-downloads.html`
3. python: `https://www.python.org/`

8.1.2 Dependencies installation on linux

1. sudo apt update
2. sudo apt install git
3. sudo apt-get install openjdk-8-jre
4. sudo apt-get install python3

8.2 Steps to reproduce

Requirements:

You will need a GitHub personal access token (`https://help.github.com/en/github/authenticating-to-github/creating-a-personal-access-token`). Place it in `Scripts/token.txt`. It is used for projects using GitHub as issue tracker.

8.2.1 SZZUnleashed: with and without improvements

On windows replace `python3` with `python` and `pip3` with `pip`.

1. Prepare SZZ:

```
git clone https://github.com/pwr-pbrwio/PBR20M1
cd PBR20M1
pip3 install -r requirements.txt
cd ..

```

2. Get repository from data set (example of `commons-lang`):

```
mkdir commons-lang
cd commons-lang
git clone https://github.com/apache/commons-lang.git

```

3. Download project issues (filtered with data set)
   If you are using Jira as the issue tracker:

```
python3 ../PBR20M1/Scripts/getNetoIssues.py --owner "apache"
--repo "commons-lang" --tag "lang" --repoPath "./commons-lang
" --jira "issues.apache.org/jira"
```

   If you are using GitHub as the issue tracker (e.g., for mockito):

```
python3 ../PBR20M1/Scripts/getNetoIssues.py --owner "mockito"
 --repo "mockito" --repoPath "./mockito" --fetchStrategy
github
```

4. Run the SZZ algorithm:

```
java -jar "../PBR20M1/Scripts/unleashed/szz.jar" -i ".temp/
issue_list.json" -r "./commons-lang" -d=3  -fix -ra -up -mt -
fp
```

   Where flags -fix -ra -up -mt -fp are optional: -fix enables fix -ra runs SZZ with refactoring awareness -up removes comments -mt limits time between commits to 2 years -fp limits SZZ to .java files
5. Get results:

```
python3 ../PBR20M1/Scripts/measurePos.py --repoName="commons-
lang"
```

8.2.2 OpenSZZ

Dependency requirements – the following software is required:

- docker
- docker-compose

Usage:

1. Clone the OpenSZZ repo (`https://github.com/clowee/OpenSZZ-Cloud-Native`).
2. Publication was prepared on version 533b4911710753e76c78c02c02ca10707a74e05b. Make sure the correct version is used.
3. Increase the heap size by adding the `JVM_OPTS` environmental variable to the web service in the docker-compose.yml file. Note that using docker on windows or linux might require increasing the total memory assigned to docker in the docker settings. Example:

```
web:
build: ./core
ports:
- "${PORTRANGE_FROM}-${PORTRANGE_TO}:8080"
networks:
  - spring-cloud-network
  depends_on:
  - rabbitmq
  volumes:
  - /var/run/docker.sock:/var/run/docker.sock
  environment:
  - JVM_OPTS=-Xmx12g -Xms12g -XX:MaxPermSize=1024m

```

4. Follow the OpenSZZ readme file (`https://github.com/clowee/OpenSZZ-Cloud-Native`) for instructions on starting the application and running repositories.
5. Rename results to `BugInducingCommits.csv`.
6. Analyse results:

```
python3 ../PBR20M1/Scripts/openSzzAcc.py --repoName="commons-lang"

```

References

Borg M, Svensson O, Berg K, Hansson D (2019) SZZ unleashed: an open implementation of the SZZ algorithm-featuring example usage in a study of just-in-time bug prediction for the Jenkins project. In: Proceedings of the 3rd ACM SIGSOFT International Workshop on Machine Learning Techniques for Software Quality Evaluation, pp 7–12

da Costa DA, McIntosh S, Shang W, Kulesza U, Coelho R, Hassan AE (2017) A Framework for Evaluating the Results of the SZZ Approach for Identifying Bug-Introducing Changes. IEEE Transactions on Software Engineering 43(7):641–657

Just R, Jalali D, Ernst MD (2014) Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis, pp 437–440

Lenarduzzi V, Palomba F, Taibi D, Tamburri D (2020) OpenSZZ: A Free, Open-Source, Web-Accessible Implementation of the SZZ Algorithm

Neto EC, da Costa DA, Kulesza U (2018) The impact of refactoring changes on the SZZ algorithm: An empirical study. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp 380–390

Neto EC, da Costa DA, Kulesza U (2019) Revisiting and Improving SZZ Implementations. In: 2019 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE, pp 1–12

Rodriguez G, Robles G, Gonzalez-Barahona J (2018) Reproducibility and Credibility in Empirical Software Engineering: A Case Study based on a Systematic Literature Review of the use of the SZZ algorithm. Information and Software Technology DOI 10.1016/j.infsof.2018.03.009

Silva D, Silva J, De Souza Santos GJ, Terra R, Valente MTO (2020) RefDiff 2.0: A Multi-language Refactoring Detection Tool. IEEE Transactions on Software Engineering pp 1–1, DOI 10.1109/TSE.2020.2968072

Śliwerski J, Zimmermann T, Zeller A (2005) When do changes induce fixes? ACM sigsoft software engineering notes 30(4):1–5

Tsantalis N, Mansouri M, Eshkevari LM, Mazinanian D, Dig D (2018) Accurate and Efficient Refactoring Detection in Commit History. In: Proceedings of the 40th International Conference on Software Engineering, ACM, New York, NY, USA, ICSE '18, pp 483–494, DOI 10.1145/3180155.3180206, URL http://doi.acm.org/10.1145/3180155.3180206

Williams C, Spacco J (2008a) SZZ Revisited: Verifying When Changes Induce Fixes. In: Proceedings of the 2008 Workshop on Defects in Large Software Systems, Association for Computing Machinery, New York, NY, USA, DEFECTS '08, p 32–36, DOI 10.1145/1390817.1390826, URL https://doi.org/10.1145/1390817.1390826

Williams CC, Spacco JW (2008b) Branching and Merging in the Repository. In: Proceedings of the 2008 International Working Conference on Mining Software Repositories, Association for Computing Machinery, New York, NY, USA, MSR '08, p 19–22, DOI 10.1145/1370750.1370754, URL https://doi.org/10.1145/1370750.1370754