SZZ++: Handling bug information in Git source code repositories

M.S. Project Report

Saheel Ram Godhane
University of California
Davis, CA
srgodhane@ucdavis.edu

ABSTRACT

Bug information is widely used as a direct or indirect metric of code quality in software engineering research. It is critical to extract this information in its various forms (the defective lines, number of defects, etc.) from source code repositories efficiently, correctly, and easily. SZZ++ is a tool for extracting such bug information from Git repositories containing source code. SZZ++ can also be used to extract type information and combine it with the raw bug data to generate more meaningful metrics. This report serves as a descriptive manual for the tool.

KEYWORDS

Software Mining, Data Extraction, Workflow

1 INTRODUCTION

Mining online software repositories, stored on code hosting websites such as GitHub, is a common methodology in software engineering research. While the specific aim can vary, generally speaking, researchers try to find patterns in these source code repositories and thereby draw conclusions with the overall goal to further developer productivity. Such mining invariably involves:

- downloading version controlled code histories of one or more projects,
- parsing these histories to extract relevant information,
- constructing suitable metrics from this extracted information,
- using theses metrics in experiments, etc.

SZZ++ was written to automate the first two steps – specifically, extracting bug information in the second step – and can be easily extended to perform further analyses.

In Section 2, we take a look at the original algorithm which inspired our tool and the changes (some being improvements) we have incorporated in our implementation. In Section 3, we give a detailed description of the several tasks SZZ++ can perform for its user, along with usage examples. We also discuss on how SZZ++ can be extended to add problem-specific functionality. In the concluding section, we discuss some of the ways in which the tool itself can be improved.

2 BACKGROUND AND METHODOLOGY

2.1 SZZ: the algorithm

In their 2005 paper, Śliwerski, Zimmermann, and Zeller [5] first discussed an algorithm to automatically locate fix-inducing changes by linking the version history of a project with its bug data. They worked with CVS as the version control system and Bugzilla as the bug tracking system. The basic idea behind their algorithm, henceforth referred to as SZZ, is as follows:

- Start with a bug report in the bug database, indicating a *fixed problem*;
- Extract the associated change from the version archive, which gives us the *location* of the fix;
- Determine the *earlier change* at this location before the bug was reported.

This *earlier change* is a candidate for *fix-inducing changes*. The example in Figure 1 makes it clearer.

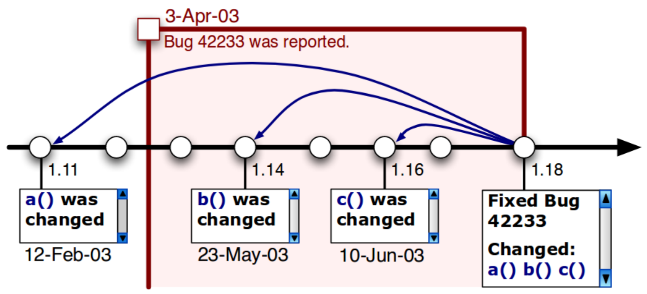


Figure 1: The SZZ algorithm tracking the fix-inducing changes.

Bug ID 42233 was fixed in version 1.18 and the associated changes are tracked back in the version history using the *annotate* command of CVS. Amongst these changes, we can ignore the ones committed in versions 1.14 and 1.16 since they occur after the bug was reported. Whereas the change made in version 1.11 were committed before the bug was reported, hence it is noted as a fix-inducing change.

2.2 SZZ++: extensions and improvements over SZZ

It is important to note that our implementation of the idea presented in the SZZ algorithm has significant differences. Here we discuss some of them.

Development-time bugs: Firstly, SZZ++ has the capability to work with development-time bugs as well as post-production bugs. For the former, since we do not have any bug database during development time, we have to parse GIT commit messages and look for error-related syntactic elements in order to get hold of bug-fix commits. Specifically, we convert each commit message to a bag-of-words, stem the bag-of-words using standard text processing techniques, and then mark the commit as a bug-fix commit if it contains at least one of these keywords: 'error', 'bug', 'fix', 'issue', 'mistake', 'incorrect', 'fault', 'defect', 'flaw', and 'type'. This strategy is based on prior work by Mockus et al. [2], and Ray et al. [3] [4].

As far as post-production bugs are concerned, SZZ++ has an option to include a bug database complete with time-stamps for each bug, so that it can filter out potential bug-fix change candidates as was done in the original SZZ algorithm. More details are given in the following section.

Git-specific improvements: Secondly, since we are working with GIT instead of CVS, we have access to much more sophisticated tracking of each line of code of our projects. As noted by Kim et al. [1], the original SZZ algorithm has a couple of limitations:

- The tracked lines do not have sufficient function/method annotation information which can lead to some erroneous reporting of bug-fix changes.
- The changes to comments, blank lines, and formatting are flagged as bug-fixes even though clearly they are only cosmetic changes.

The first one is simply a limitation of CVS, the version control system used in the original investigation as it does not care about the function containing each line of code. Whereas the second limitation is a result of not looking into the contents of each changed line while tracking it. As recommended by Kim et al., we have improved upon both these areas in our implementation. Each line in our resultant database has detailed annotation information, including not just containing method but also containing AST block, if any.

Working with snapshots: And finally, SZZ++ allows you to conduct longitudinal experiments. It records bug data for various snapshots/versions throughout the history of the project. This is done in three steps: identifying lines related to a bug, identifying commits that introduced the bug, and mapping the buggy lines to the snapshots of interest.

The first two steps are the same as in SZZ, albeit using development-time bugs. Now, once we know where the buggy lines originate, we use GIT-BLAME, a functionality provided by GIT, along with the '–reverse' option to locate these lines in various snapshots. GIT-BLAME essentially maps the buggy lines to specific snapshots.

Figure 2 taken from [3] illustrates this procedure. Here, c4 is the bug-fix commit and the corresponding buggy lines (color-marked red in the old version of the file) are found to originate from two earlier

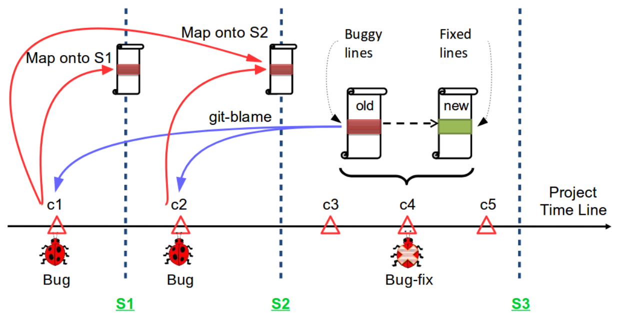


Figure 2: The SZZ++ algorithm mapping buggy lines onto multiple snapshots. Vertical dashed lines are snapshots, named S1...S3, whereas triangles marked on the Project Time Line are commits, named c1...c5.

commits c1 and c2, as indicated by the blue curved arrows. SZZ++ then maps the buggy lines from c1 to S1 and S2, as indicated by the curved arrows. On the other hand, the buggy lines from c2 can only be mapped onto S2 because they don't exist in S1.

At the end of these there steps, we know exactly which lines in each of our snapshots are buggy (and were consequently fixed in later snapshots) and which lines are benign.

3 TOOL DESCRIPTION AND USAGE

In this section we will look at some of the features of SZZ++, how to use them, and how to customize and extend SZZ++. SZZ++ is written mostly in Python with a few helper scripts written in the statistical programming language R.

In order to use SZZ++, one needs to have Python2.7 installed with a few essential packages such as git-python and psycopg2.

3.1 'main.py', the starting point

'main.py', as the name suggests, is the main wrapper around the essential functions of SZZ++. Executing this script without any options prints the Usage Help into the terminal. As seen in Figure 3 on the following page, not providing any arguments results in an exception which reads, "ValueError: 7 or more arguments expected. 0 given."

At the top of the Usage Help is the description of these required command line arguments:

- Path to the output directory
- Name of the project
- Language of the project (c, cpp, java, etc.)
- GITHUB URL of the project
- Interval between snapshots given in number of months
- Number of processing threads you want SZZ++ to create
- The SZZ++ steps you want to execute.

Following this, the Usage Help lists the steps that you can execute with SZZ++. Essentially, these steps are the functions/scripts wrapped inside 'main.py'. Some or all of the command line arguments provided to 'main.py' are used in these steps.

Though Figure 3 shows the eighteen steps that can be executed, the crux of SZZ++ are the initial ten steps. Rest of the steps are SZZ++ extensions that were used in recent projects. One of the principles for SZZ++ is to make the data pipelines easier to manage. Thus once someone starts using SZZ++ in their work, their custom list of steps might look different.

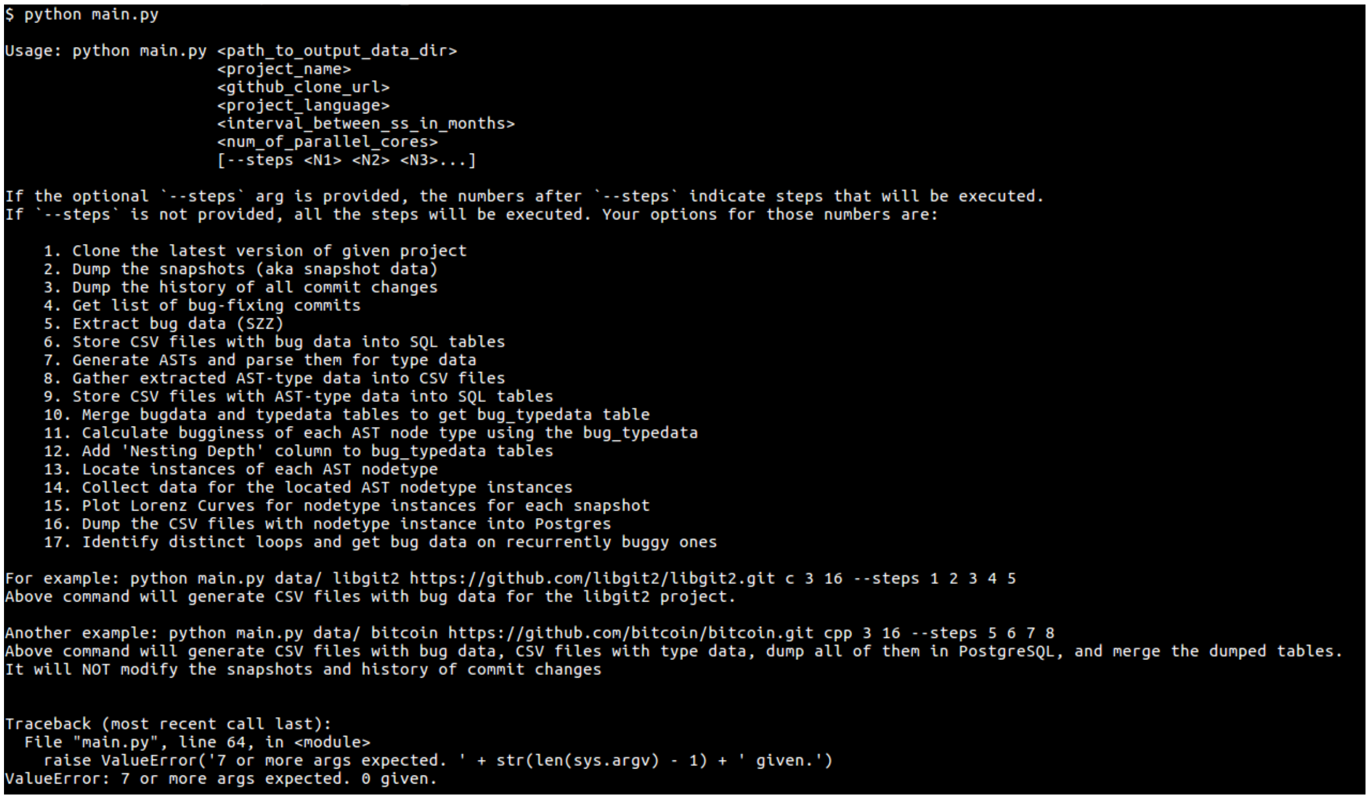
Below the list of steps in the Usage Help are a couple of examples to show how to call 'main.py'. The first example executes the initial five steps for the project LIBGIT2. Below, we look at the details of these individual tasks with the running example of LIBGIT2.

3.2 Steps in SZZ++

Each step in SZZ++ is simply a script that gets called from 'main.py' with appropriate arguments, performs some action, and writes the output to disk. Because of this, generally speaking, each step is dependent on the output of the previous step.

The essential steps of SZZ++, that is, the scripts that extract bug and type data from given projects employ parallel processing wherever the tasks are embarrassingly parallel. As mentioned before, this behavior is controlled by the sixth argument to 'main.py'.

Some of the steps might seem to be doing rather insignificant tasks, but this is a deliberate choice to increase modularity. We will now discuss what



```
$ python main.py

Usage: python main.py <path_to_output_data_dir>
                      <project_name>
                      <github_clone_url>
                      <project_language>
                      <interval_between_ss_in_months>
                      <num_of_parallel_cores>
                      [--steps <N1> <N2> <N3>...]

If the optional `--steps` arg is provided, the numbers after `--steps` indicate steps that will be executed.
If `--steps` is not provided, all the steps will be executed. Your options for those numbers are:

    1. Clone the latest version of given project
    2. Dump the snapshots (aka snapshot data)
    3. Dump the history of all commit changes
    4. Get list of bug-fixing commits
    5. Extract bug data (SZZ)
    6. Store CSV files with bug data into SQL tables
    7. Generate ASTs and parse them for type data
    8. Gather extracted AST-type data into CSV files
    9. Store CSV files with AST-type data into SQL tables
    10. Merge bugdata and typedata tables to get bug_typedata table
    11. Calculate bugginess of each AST node type using the bug_typedata
    12. Add 'Nesting Depth' column to bug_typedata tables
    13. Locate instances of each AST nodetype
    14. Collect data for the located AST nodetype instances
    15. Plot Lorenz Curves for nodetype instances for each snapshot
    16. Dump the CSV files with nodetype instance into Postgres
    17. Identify distinct loops and get bug data on recurrently buggy ones

For example: python main.py data/ libgit2 https://github.com/libgit2/libgit2.git c 3 16 --steps 1 2 3 4 5
Above command will generate CSV files with bug data for the libgit2 project.

Another example: python main.py data/ bitcoin https://github.com/bitcoin/bitcoin.git cpp 3 16 --steps 5 6 7 8
Above command will generate CSV files with bug data, CSV files with type data, dump all of them in PostgreSQL, and merge the dumped tables.
It will NOT modify the snapshots and history of commit changes


Traceback (most recent call last):
  File "main.py", line 64, in <module>
    raise ValueError('7 or more args expected. ' + str(len(sys.argv) - 1) + ' given.')
ValueError: 7 or more args expected. 0 given.
```

Figure 3: Executing 'main.py' brings up the Usage Help for SZZ++.

some of these steps do and what to expect when you run them:

1. Clone the given project: Quite simply, this script check if the project exists at the provided GITHUB URL, and tries to download a copy of its latest version.

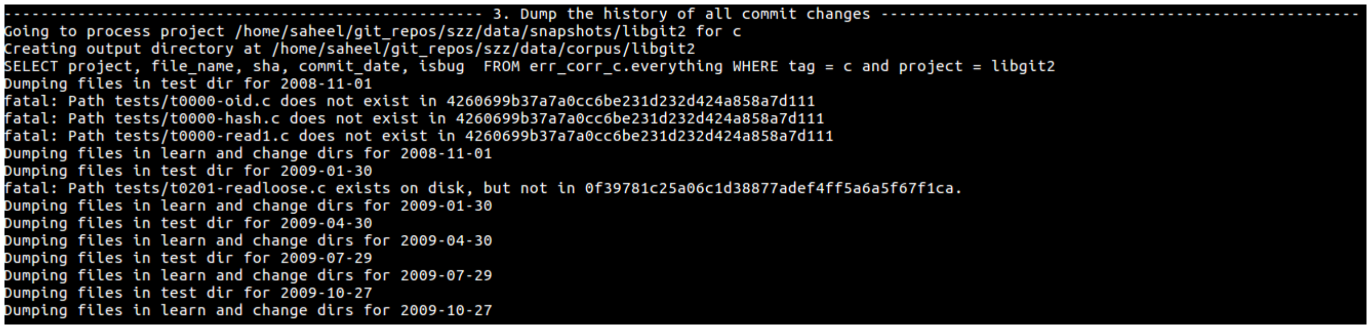
2. Dump the snapshots: This step will check out different versions of the project every N months, where N is a user-given argument. If you only care about the first and last version, then N should equal the entire duration of the project. You can manually delete some versions/snapshots from the output of this step, if it suits your purpose. The next step only considers and works on the snapshots present in the output directory after.

3. Dump the commit change history: This step extracts all the changes committed to the project history between each pair of snapshots. By a commit change we mean the old and new versions of files changed in a commit. Figure 4 shows this step being performed for a few snapshots of the LIBGIT2 project.

4. Get the list of bug-fix commits: Getting hold of the list of bug-fix commits depends on the type of bugs we are dealing with – development-time or post-production. For development-time bugs, the user can run the 'R/scripts/bug_predict.R' script provided as part of SZZ++. Whereas, for post-production bugs the list can easily be obtained from a bug database like BUGZILLA. Since this detail depends on the specific software project being used, we cannot automate it as part of SZZ++.

As of now it is assumed the user has a way of fetching the list of bug-fix commits. Thus, this step simply fetches the list of bug-fix commits that the user has stored in their database.

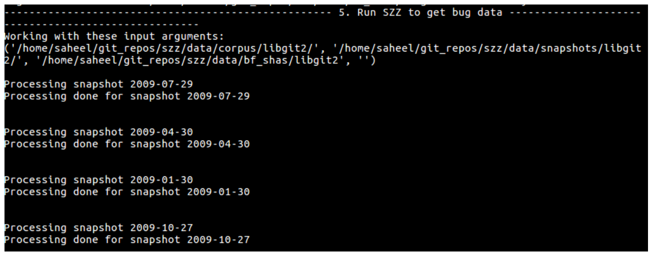
5 and 6. Extract and save bug data: This is the quintessential step in the list where bug data



```
------------------------------------------------- 3. Dump the history of all commit changes --------------------------------------------------
Going to process project /home/saheel/git_repos/szz/data/snapshots/libgit2 for c
Creating output directory at /home/saheel/git_repos/szz/data/corpus/libgit2
SELECT project, file_name, sha, commit_date, isbug  FROM err_corr_c.everything WHERE tag = c and project = libgit2
Dumping files in test dir for 2008-11-01
fatal: Path tests/t0000-oid.c does not exist in 4260699b37a7a0cc6be231d232d424a858a7d111
fatal: Path tests/t0000-hash.c does not exist in 4260699b37a7a0cc6be231d232d424a858a7d111
fatal: Path tests/t0000-read1.c does not exist in 4260699b37a7a0cc6be231d232d424a858a7d111
Dumping files in learn and change dirs for 2008-11-01
Dumping files in test dir for 2009-01-30
fatal: Path tests/t0201-readloose.c exists on disk, but not in 0f39781c25a06c1d38877adef4ff5a6a5f67f1ca.
Dumping files in learn and change dirs for 2009-01-30
Dumping files in test dir for 2009-04-30
Dumping files in learn and change dirs for 2009-04-30
Dumping files in test dir for 2009-07-29
Dumping files in learn and change dirs for 2009-07-29
Dumping files in test dir for 2009-10-27
Dumping files in learn and change dirs for 2009-10-27
```

Figure 4: Saving various snapshots (aka versions) of the project to disk. Test files are ignored.

is extracted using the SZZ algorithm. The output consists of multiple CSV files, one for each snapshot and a single merged CSV file consisting bug data for all the snapshots. This single CSV file is named 'ss_bugdata.csv' and resides in the '/data/corpus/libgit2' directory in case of the project Libgit2.



```
------------------------------------------------- 5. Run SZZ to get bug data --------------------
------------------------------
Working with these input arguments:
('/home/saheel/git_repos/szz/data/corpus/libgit2/', '/home/saheel/git_repos/szz/data/snapshots/libgit
2/', '/home/saheel/git_repos/szz/data/bf_shas/libgit2', '')

Processing snapshot 2009-07-29
Processing done for snapshot 2009-07-29


Processing snapshot 2009-04-30
Processing done for snapshot 2009-04-30


Processing snapshot 2009-01-30
Processing done for snapshot 2009-01-30


Processing snapshot 2009-10-27
Processing done for snapshot 2009-10-27
```

Figure 5: SZZ in action. Collecting bug data for a few snapshots is shown.

For each buggy line found in the snapshots, SZZ++ records several metadata: project_name, file_name, commit_sha, line_num, is_new, is_buggy, bugfix_snapshot, bugfix_file_name, bugfix_sha, bugfix_line_num. The last five columns record information on where the buggy line was eventually fixed; rest of the columns is information about the buggy line itself. Figure 5 shows the SZZ step being run for our project Libgit2.

Step 6 simply stores these CSV files in a SQL database, which is more suited for further processing. The details of the SQL database can be edited in the 'src/szz/dump_bugdata_into_psql_table.py' script.

7, 8, and 9. Generate and store type data: These three steps generate Abstract Syntax Trees (ASTs) for the snapshots of our interest, determine the type of each line in these snapshots, and store all this type in CSV files and later into the SQL database.

These steps are not quite part of the SZZ algorithm but add a considerable amount of metadata to the buggy lines as well as non-buggy lines in the codebase.

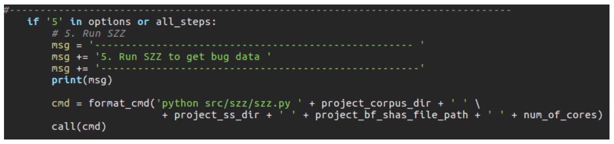
10. Merge type and bug data: This steps simply merges the generated SQL tables for type and bug data for each line in the snapshots. Since the tables may end up containing millions of rows, it is a fairly time consuming step. An rough estimate of processing time is given at the beginning of the SQL merge.

At the end of this merge, we have the following columns for each row in the table (which denotes a line in some snapshot of the project): project, snapshot, file_name, sha, line_num, line_type, parent1, parent2, parent3, parent4, parent5, parents_all, ast_depth, loop_nesting_depth, is_buggy, bf_ss, bf_file_name, bf_sha, bf_line_num. Apart from the bug data columns, now we have the type of each line, its parents in the AST, and its nesting depth.

Steps 13 through 16 optionally add containing AST block and containing function information to this set.

3.3 Customization and Extension of SZZ++

Mapping onto single or multiple snapshots: SZZ++ also provides an option to map the buggy lines onto all the snapshots after the bug was introduced and before it was fixed, OR only onto the snapshot nearest to the bug-introducing commit. This option allows the user more control over what kind of snapshot bug data should be collected. To enable this option, the user can check out the 'blame_on_only_one_ss' branch of the repository.



```
#-------------------------------------------------------------------------------
    if '5' in options or all_steps:
        # 5. Run SZZ
        msg = '------------------------------------------------------ '
        msg += '5. Run SZZ to get bug data '
        msg += '-------------------------------------------------------'
        print(msg)

        cmd = format_cmd('python src/szz/szz.py ' + project_corpus_dir + ' ' \
                         + project_ss_dir + ' ' + project_bf_shas_file_path + ' ' + num_of_cores)
        call(cmd)
```

Figure 6: A look inside 'main.py' showing the wrapper code for Step 5.

Adding custom steps: Figure 6 shows the code for Step 5 – Generate bug data (SZZ) – as it appears in 'main.py'. You can easily call your own scripts by adding such small snippets of code to 'main.py'.

4 CONCLUSIONS

We present SZZ++, a tool for gathering and processing bug and type data for software projects. SZZ++ is modular, parallel, customizable, and extensible. It can be used for pipe-lining the essential data collection steps of any software engineering research project that involves mining version controlled codebases.

REFERENCES

[1] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. 2006. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on.* IEEE, 81–90.

[2] Audris Mockus and Lawrence G. Votta. 2000. Identifying Reasons for Software Changes Using Historic Databases. In *Proceedings of the International Conference on Software Maintenance (ICSM'00) (ICSM '00).* IEEE Computer Society, Washington, DC, USA, 120–. http://dl.acm.org/citation.cfm?id=850948.853410

[3] Baishakhi Ray, Vincent Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar Devanbu. 2016. On the naturalness of buggy code. In *Proceedings of the 38th International Conference on Software Engineering.* ACM, 428–439.

[4] Baishakhi Ray, Daryl Posnett, Vladimir Filkov, and Premkumar Devanbu. 2014. A Large Scale Study of Programming Languages and Code Quality in Github. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014).* ACM, New York, NY, USA, 155–165. https://doi.org/10.1145/2635868.2635922

[5] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do changes induce fixes?. In *ACM sigsoft software engineering notes*, Vol. 30. ACM, 1–5.