

How is Testing Related to Single Statement Bugs?

HABIBUR RAHMAN* and SAQIB AMEEN*, University of Alberta, CA

ACM Reference Format:

Habibur Rahman and Saqib Ameen. 2021. How is Testing Related to Single Statement Bugs?. 1, 1 (April 2021), 6 pages. <https://doi.org/10.1145/nnnnnnnn.1145/nnnnnnnn.nnnnnnnn>

1 INTRODUCTION

Writing unit tests is a common practice in the industry to ensure software quality. Usually meeting certain criteria of test coverage serves as a segway to product release. This is essential because software systems are used in critical areas such as health, security, finance, and space missions. In such settings, a faulty system is not acceptable. International Organization for Standardization (ISO) also mandates testing as a part of the software development life cycle. Despite all efforts, the software system may remain prone to bugs. This raises a question on the effectiveness of testing. Is it even useful?

Multiple studies tried to answer this question under different settings but there seems to be no consensus on it. These studies examined all types of bugs that were found in the systems. Nobody studied the effectiveness of testing on subsets of the bugs found in the software systems. One important subset of bugs is known as Single Statement Bugs (SSBs). They appear in just a single statement and can be fixed by modifying that statement. Those modifications can be as simple as changing a variable name, ordering arguments in a function, changing the return type, and so on. Figure 1 shows an example of a single statement bug where `!=` needs to be replaced with `==`.

```
- if (submittedNode == null || submittedNode.get("values") != null) {  
+ if (submittedNode == null || submittedNode.get("values") == null) {
```

Fig. 1. Example of a single statement bug before and after the fix.

SSBs occur quite often [7] and can be very critical. For example, Apple's return bug resulted in an invalid SSL/TLS connection verification, putting the sensitive data of millions at risk. The normal distribution of the ratio of SSB/all bugs for top 100 open source java projects using the Maven build system on GitHub has a bell curve shown in Figure 2. We can see that the mean lies around > 0.4 . This means there are more than 40% of SSBs in those projects. Mitigating this large chunk of bugs can be useful in improving the overall quality of the software and prevent failures in production. The goal of our study to help practitioners understand the effectiveness of testing for SSBs.

*Both authors contributed equally to this research.

Authors' address: Habibur Rahman, habibur@ualberta.ca; Saqib Ameen, saqib1@ualberta.ca, University of Alberta, Edmonton, AB, CA.

© 2021 Association for Computing Machinery.

This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>.

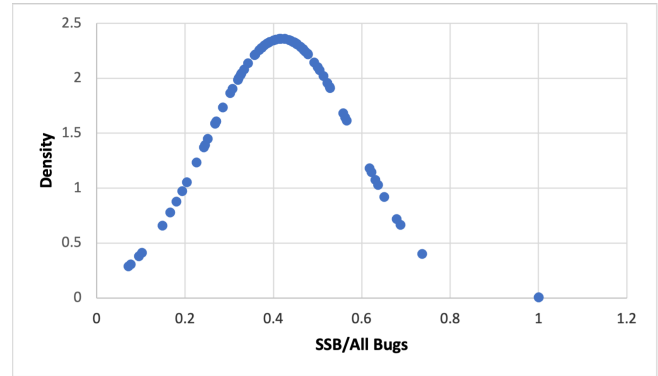


Fig. 2. Distribution of SSBs in the top 100 open source Java projects on GitHub using Maven build system

We intend to understand the effectiveness of testing on SSBs by the correlation between test coverage and single statement bugs. If more test coverage in a project results in fewer SSBs, it shows that testing is helpful and vice versa. This leads to our research question, *Is there a correlation between test coverage and single statement bugs?* We hypothesized that there is a weak to moderate relationship between SSBs and test coverage, i.e., testing is helpful to some extent.

To verify our hypothesis, we analyze the post-unit test bugs in the areas covered and not covered by the testing. If a part is covered by testing, it means it has been executed for its intended purpose and it works fine. In future, if bugs are found in the covered part, it will show that testing did not help here. To check that, we first generate the coverage report at the release time, and see if the bugs found after the release are in covered part or not covered part. Finally, we calculate the percentage of coverage and percentage bugs in the not covered part, to find the correlation between them. It serves as a proxy for the effectiveness of unit testing for SSBs.

For our study, we use the Mining Software Repositories (MSR) 2021 challenge' SSBs dataset[7]. It contains 7824 SSBs from the top 100 Maven-based open source Java projects on GitHub. The important thing about this dataset is that projects can be built and their test can be executed. Which is important for us to be able to generate reports.

After the experiment, we have found that there is a weak to moderate correlation between the test coverage and SSBs. This shows that testing is effective to some extent in reducing the SSBs.

2 BACKGROUND AND TERMINOLOGY

Test coverage is the percentage of lines of code executed by the tests for a project. We measure testing in the form of test coverage. A project that has 87% coverage means 87 out of 100 lines of code has been executed by the test cases of that project. In this writing, the word 'coverage' refers to test coverage.

In statistics, correlation is the statistical relationship between two random variables or bivariate data. Correlation is measured in terms of the correlation coefficient that ranges from -1 to 1 where -1 means negative correlation, 0 means no correlation, and 1 means a positive correlation between two variables. The correlation coefficient is useful for hypothesis testing. Based on the value of the correlation coefficient, a hypothesis can be accepted or rejected. There are several methods to calculate this coefficient, Pearson correlation coefficient is one of them. It can be expressed as the equation below:

$$r_{xy} = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 (y_i - \bar{y})^2}} \quad (1)$$

Where:

r_{xy} – the correlation coefficient of the linear relationship between the variables x and y

x_i – the values of the x-variable in a sample

\bar{x} – the mean of the values of the x-variable

y_i – the values of the y-variable in a sample

\bar{y} – the mean of the values of the y-variable

Percentage test coverage, and number of bugs in the not covered part are the variable for our case.

3 RELATED WORK

Our work overlaps with two niches of the software engineering literature. First one is related to analyzing the relation of test coverage and its effectiveness. They utilize different techniques, under different conditions to study the usefulness of test coverage. The second one is related to the usage of SSB for an empirical study or a software application.

In the first kind of work, more or less, researchers aim to find the correlation between test coverage and the occurrence of bugs. To date, no consensus exists in the community on the usefulness of test coverage. Some studies tend to agree, while others disagree. For example, Gren and Antinyan [5] and Antinyan et al. [1] found a weak to no correlation between the test coverage and post-unit test defects in their separate studies and concluded that test coverage was not helpful. Both of them conducted the study on a single, but large-scale industrial project. The former one did not reveal the details of the project, while the latter one worked with Ericson. Ericson mainly deals with telecommunications and networking software around the world. One aspect of these studies which is similar to our work is that they rely on actual bugs found after unit tests and real test cases written by developers. In contrast to our case, both of them considered all types of bugs, not just the SSBs. The methodology of Gren and Antinyan [5] also differs from our approach, they evaluate coverage in terms of files and only consider the files with either 100% coverage or no coverage at all. While Antinyan et al. [1] considered the overall coverage, which is similar to our approach. Due to the small sample size and specific software niche, the results of both studies cannot be generalized for other studies.

On the other end of the spectrum, Inozemtseva and Holmes [6] found a weak to moderate correlation between the test coverage and its effectiveness, when the test suite size was controlled for. They conducted the study on five large open-source Java projects

powered by the Ant build system and during the study. Contrary to our study, they used synthetic test suits and mutation testing to generate faulty programs. While they are a good approximation, they do not necessarily reflect the real scenarios and are limited by the algorithms behind those tools. Furthermore, they considered an extra variable, i.e., the test suite size, and varied it throughout the study while generating test cases. In our approach, we do not rely on any tests or bug generators. In another study, Namin and Andrews [8] also considered the role of test suite size in addition to the coverage on the effectiveness of unit tests. They found that by increasing the coverage, indirectly, the test suite size is increased, which increases the effectiveness of tests. For varying, but controlled test suite sizes, they found a high to weak correlation between the coverage and test effectiveness. Similar to the aforementioned study, they also relied on self-generated test cases and considered only the Siemen suite of seven (C, C++) programs. Our technique is similar to what Bach et al. [2] have used in their study related to the impact of coverage on bug density. But we use a simpler approach since we only have SSBs. In comparison to our study, they considered all types of bugs and analyzed a single project, i.e., SAP HANA, and found a positive impact of coverage on bugs density. In general, the literature work on this topic differs from our work in one or more of the following areas:

- A very small number of projects were considered.
- Artificially generated test cases were used.
- Synthetic bugs were introduced in the system.
- All types of bugs were considered.

Furthermore, in studies, done in collaboration with the industry, the details of the analyzed software were not released. Which is a barrier in generalizing or understanding the results from those studies. There could be additional internal factors and software engineering practices leading to those results. Lastly, our dataset differs from those used in the aforementioned studies.

The second part of the literature, which uses SSB mainly differs from our work in terms of intended implications. To the best of our knowledge, at this point, the use of SSB in literature is limited to program repair. There is no study related to test coverage and its effectiveness. Chen et al. [3] used a combination of Bugs2Fix [9] and CodRep [4] single statement dataset to propose a novel learning-based program repair technique. Their dataset is also different from the one we are using. Their dataset is not intended for building projects or running tests as there is no information on the project's build system or even if they can be run or not. Those factors are important for us to be able to generate the coverage reports. Another study by Karampatsis and Sutton [7] also uses SSBs. In their study, they provided a new dataset on SSBs and all types of bugs in the top 100 open-source Java projects on GitHub and attempted to find the frequency of occurrence of SSBs. They found out that SSBs occur with a frequency of one bug per 1600-2500 lines of code. Even more important contribution of this paper is that the projects in their dataset use the Maven build system and can be built. If there are tests in the project, they can be executed. We use their dataset of SSB and real test cases present in those projects to conduct this study on the effectiveness of coverage on SSBs.

4 METHODOLOGY

This section describes the detailed methodology used to process the information from the dataset and curate the desired data for evaluation. It starts with the description of the dataset for a better context and then goes through each step describing its purpose, input, and output.

Our main goal is to find out if there are more bugs or fewer bugs in the covered part, after unit tests were written. For this purpose, an overview of our methodology is shown in Figure 3. It consists of two major steps. In step (A), we generate the coverage reports. In the second step (B), with the help of those coverage reports, we try to find if a certain bug belongs to the covered part or not. At the end of this whole process, we have percentage coverage for each project, and the percentage covered of SSBs in the covered as well as uncovered parts. We prune the resulting data for any outliers and then use Pearson's correlation to find our results.

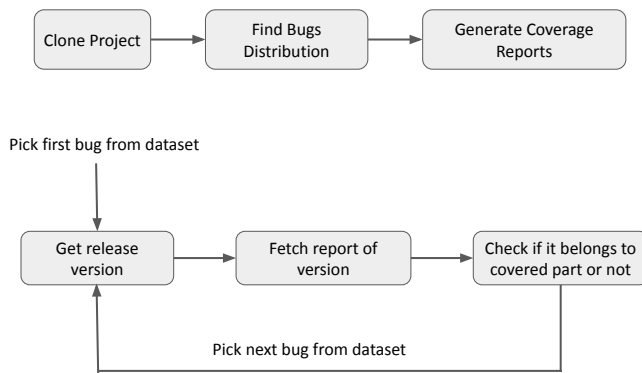


Fig. 3. An overview of the complete methodology.

4.1 Dataset

For our study, we used the ManySSTuBs4J[7] dataset. This corpus consists of simple fixes to SSBs found in top open source Java projects on GitHub. It has two different variants. The first one contains the top 100 open source Java projects which use Maven as a build system. They can be built, and tests in those projects can be executed. The second one contains the top 1000 open source Java projects. In this larger variant, the projects use different build systems and they may or may not be built. For our study, a natural choice is to use the former variant due to testability. Table 1, shows the dataset statistics.

Projects	Bug Commits	Buggy stmts	Bugs per Commit	SSBs
100 Java Maven	12598	25539	2.03	7824
1000 Java	86771	153652	1.77	51537

Table 1. Statistics of the ManySSTuBs4J dataset.

Figure 4, shows an example of the ManySSTuBs4J dataset instance. In this dataset snapshot, we can see that it contains a bug type,

project name, fix commit SHA, parent commit SHA, Bug Line number, and other properties. For our study, we have used the project name, fix commit SHA, parent commit SHA, and bug line numbers from each of the data items. In the *projectName* property, the part before the period represents the organization name or user name and the part after the period represents the name of project. It comes handy while cloning the repos for processing in the later sections. For the sake of brevity, we omitted some parts in the snapshot, which are represented by three dots.

```

{
  "bugType": "CHANGE_OPERATOR",
  "fixCommitSHA1": "aa90e04b5e6b7f6d46dde16867196329568324e",
  "fixCommitParentSHA1": "46d3a4007fe1418d53baabc16dec39275079684b",
  "bugFilePath": "/java/org/./GetRuntimeFormDefinitionCmd.java",
  "fixPatch": "...",
  "projectName": "Activiti.Activiti",
  "bugLineNum": 184,
  "bugNodeStartChar": 8444,
  "bugNodeLength": 35,
  "fixLineNum": 184,
  "fixNodeStartChar": 8444,
  "fixNodeLength": 35,
  "sourceBeforeFix": "submittedNode.get(\"values\") != null",
  "sourceAfterFix": "submittedNode.get(\"values\") == null"
}

```

Fig. 4. A snapshot of the ManySSTuBs4J dataset instance.

4.2 Splitting the Dataset By Projects

The corpus of the dataset is quite large. While processing a single project, we should not be enumerating through the whole dataset. To ease the search, we split the dataset based on the project name. We processed the whole dataset to sort it by the project name and then saved data of each project in separate JSON files. In each of the new datasets, they contain bugs from the same project.

4.3 Cloning Projects

To clone the projects, first we constructed the repository link for each project by using the *projectName* property given in the dataset. Then we used those links to clone the projects from GitHub in our local machines for further processing. The whole process was automated through a Python script.

4.4 Finding the Distribution of Bugs

This is a crucial step. The bugs listed in the ManySSTuBs4J[7] dataset span multiple years. For example, in the case of JUnit4, there are bugs from the year 2002 till 2017. To check, if a bug was discovered in the covered portion or not, we need to get the coverage report at the release time before that bug was found. Since we target a certain coverage before each release, the release time serves as a good point to take the coverage report. But it is very difficult and time-consuming to generate reports for such a long time for various reasons. Firstly, it is hard to resolve the dependencies to execute the test cases, and there could be a very large number of releases in such a long period. Secondly, for the very old versions, it hard

to integrate Jacoco to begin with. Jacoco is the tool that we use to generate the coverage reports.

To resolve this, we find the distribution of the bugs with respect to years. Then we identify the high bug density area to generate the reports for that portion. It serves as a good proxy for overall project. For example, Figure 5 shows an example of such distribution in the form of a bar graph for a project in the dataset, named Jedis. It is clear from the image that it is more useful to generate the reports for the releases between 2013 and 2017 than any other portion.

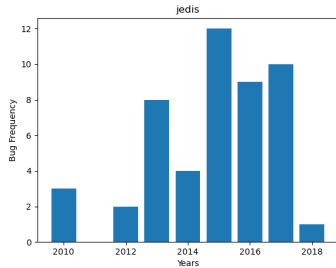


Fig. 5. Distribution of SSBs in project Jedis

For some projects, it was not possible to generate the reports for all the identified regions for a few reasons. For some versions the dependencies could not be resolved, for a few jar files were missing, some used very old versions of Maven, and in a few cases, there were no tests. Some projects had very huge test suites, taking more than an hour to execute the tests and generate the reports. In such cases, it was not possible to generate reports for a good number of versions. Another advantage of this approach is that we end up generating coverage reports for the consecutive releases. In consecutive releases, coverage does not change much. As a result, when we can take the average of coverage to indicate the percentage coverage in the project.

4.5 Generating Coverage Reports

Once we identified the versions using the bug density distribution, we manually checked out each version and resolved the dependencies to generate the coverage reports. Most of the projects did not include Jacoco to generate the reports. In such projects, we had to figure out the appropriate version for each release before generating the report. Each project also has a different structure and test runner to execute the unit tests. That also needed to be taken into account to be able to generate the reports.

We evaluated more than 50 projects from the dataset during our study. However, we were able to generate reports properly for around 17 projects only. It is mainly due to failure in resolving one or more issues mentioned above. All the generated reports and files with fixes for each project are available as a part of the recreation package for this study in the GitHub repo of the project artifacts.

4.6 Getting Release Number

Once we have all the reports for the projects. The next step (B) of our methodology begins, where we have to analyze if find the ratio

of bugs in the covered and uncovered parts. This whole process is automated with Python and Shell scripts.

For this purpose, we used the separated dataset of projects. For each project, we enumerated through its dataset, going through each SSB one by one. For each SSB, we identified its release version and then used the report generated for that version to identify if that bug was in the covered part or not. This process is repeated for all the bugs in the dataset against that project and the results are recorded.

4.7 Fetching Report for Particular Release

At this stage, we fetch those reports based on the release version. During the manual report generation process, we saved the *xml* file as well as the *html* file for the Jacoco report. The former contains the details about the covered and uncovered lines at a granular level while the latter contains the percentage coverage for the test suite.

We parsed the *xml* file using the MiniDom Python API to identify in which part the bug occurred and then parsed the *html* file with BeautifulSoup (an open source library for scrapping) to extract the percentage coverage for that particular version.

4.8 Identifying SSBs in Covered and Not Covered Parts

In this section, we explain how exactly we identified and counted the number of bugs in covered and not covered part from the *jacoco.xml*. The *xml* report file contains information like missed instructions, missed branches, as well as covered instructions and covered branches. For the bugs, where the covered instructions and covered branches were zero, they corresponded to the uncovered part and vice versa.

After enumerating the whole dataset, and repeating the aforementioned step for each of them, we end up with the total number of bugs in the covered and not covered part. Using those numbers, we find the percentage of bugs in the uncovered part.

4.9 Calculating the Correlation Coefficient

The data collected till this stage is shown in Table 2. It does not reflect all the data from section 4.8, rather it represents the data after removing the outliers. Outliers projects included a very small number of bugs (less than 5) and all of them were either in the covered part or the uncovered part. The reason such outliers occurred, despite identifying the high bug density releases in section 4.1, is that we could not generate reports for many versions in those parts.

We used this data to calculate the correlation coefficient between the percentage of SSBs that are not covered part and the average percentage coverage. For calculating correlation, we used the Equation 1. Based on this correlation coefficient, we verified the hypothesis. The results and their implications, based on the result of this step, are discussed in detail in Section 5.

5 RESULTS

Previous section explains how we used the SStuBs to collect the data from GitHub, used Jacoco to generate the coverage reports, and processed them to get the desired data. In this section, using that data, we try to answer our research question.

5.1 How is Testing Related to SSBs?

In our research question, we asked if there a correlation between test coverage and SSBs. The purpose of this question is to find out if increasing the coverage can be “actually” helpful in reducing the SSBs. Table 2 shows the data collected to answer this question. The

Project Name	Percentage Coverage	Bugs in Not Covered (%)
alibaba.druid	41.67	58.33
alibaba.fastjson	50	50
AsyncHttpClient.async-http-client	40	60
brettwouldridge.HikariCP	5.88	94.12
Bukkit.Bukkit	68	32
cucumber.cucumber-jvm	50	50
google.auto	19.23	80.769
google.closure-compiler	52.38	47.62
google.guice	35	65
jhy.jsoup	45.45	54.55
junit-team.junit	18.75	81.25
mybatis.mybatis-3	16.67	83.33

Table 2. Data collected about projects on percentage coverage and percent-age of SSBs in not covered part

first column shows the project name, and the second column shows the average percentage test coverage, while the third column shows the percentage of SSBs in the not covered part. We are considering the bugs in the non-covered part as they provide an easy way to understand the effectiveness of coverage. A higher percentage of them indicates that coverage is effective and vice versa. In our data, we have a mixed percentage. We used this data to find the correlation coefficient (r) between the percentage of coverage and the percentage of SSBs. The r -value turned out to be 0.40, which translates into a weak to moderate positive correlation between them. It shows that high coverage helps mitigate the SSBs. This correlation is shown in 6. We can see that increased covered is related to greater percentage of bugs in the not covered part and testing is useful here.

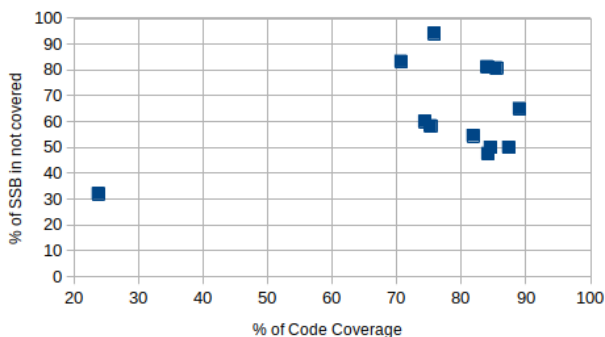


Fig. 6. Correlation of the percentage of bugs in not covered part and the average of percentage test coverage

RQ 1 Answer. *Our results suggest that there is a positive weak to moderate correlation between the unit test coverage and the number of SSBs found in the not covered parts. The coverage seems to be effective for SSBs, to some degree.*

Our study has a few important implications. Firstly, it is one of its kind to explore the effectiveness of coverage on the SSBs and the results indicate that it's a promising research area to further investigate. Further studies will help better understand the nature of this relationship under different settings and we might be able to reach a consensus about the unit test effectiveness. Secondly, knowing that testing is helpful, will help the practitioners allocate resources and prioritize testing more effectively. Which in turn, can help improve the software quality.

6 THREATS TO THE VALIDITY

In this section, we discuss some of the possible threats to the validity of our study. As described in section 4, we considered multiple versions of each project and wrote our scripts for processing and some tasks automation. This involvement of the human factor equates to the possibility of human error in the process. There could be a mismatch in the versions, and we could have made some unintentional mistakes in the code, however, throughout the process we also did manual verification to reduce this risk. The results of our study are also limited by the SStuBs(sstubs) dataset we used. While it includes a good number and range of projects from different domains, it is limited by the language type and build system. All the projects use the Maven build system and are developed in Java. The results obtained from the study might not apply to the other type system, languages, or even other build systems.

Another important aspect to consider is that all the reports are generated manually by checking out each version one by one, resolving dependencies, and configuring Jacoco. It is not only a tedious manual process but also makes it highly susceptible to error. However, throughout the process, we did manual verification to mitigate this threat as much as possible. Another threat is that the projects included in the dataset are the top open-source Java projects. They are maintained for more than a decade by the open-source community and used by tons of organizations and developers. Through their feedback and community involvement, they have matured over the years. Since we only consider the high-density bugs areas where we have tests written and we can generate reports, they might not reflect their whole project life cycle. Lastly, since we only considered open source projects, our results might not apply to the close source projects.

7 CONCLUSION AND FUTURE WORK

We tried to find the effectiveness of the unit testing for the SSBs. Instead of synthetic tests or faults, we relied on the real bugs found in the software systems and the real tests written at the time of their releases. We used a good number of projects so that the result could be generalized for the Maven based Java projects. All the projects are open source and can be explored to understand their nature. We found that to some extent, fewer bugs were found in the covered parts as compared to the uncovered parts. Which shows

that coverage should be improved in software systems to reduce the number of SSBs. For the companies, it implies that testing should be prioritized since it is helpful in ensuring the software quality.

Apart from the aforementioned implications, our study paves the way for potential future work. We identified a few directions which can be explored in the future to better understand the effect of coverage on the SSBs. We identified in related work (Section 4) that some studies added more variables, such as test suite size, while studying the effectiveness of testing. A similar approach can be taken with SSBs to see if the correlation still holds or changes. Secondly, the scope of our study is limited to only one build system, i.e, Maven, and Java language. Which is mainly due to the availability of buildable and testable softwares in Maven along with the SSBs dataset. Similar studies can be conducted with other build systems or even other languages. Lastly, all the projects included in our study were open source projects. It would be interesting to see the results in closed source projects.

REFERENCES

- [1] V. Antinyan, J. Derehag, A. Sandberg, and M. Staron. 2018. Mythical Unit Test Coverage. *IEEE Software* 35, 3 (2018), 73–79. <https://doi.org/10.1109/MS.2017.3281318>
- [2] T. Bach, A. Andrzejak, R. Pannemans, and D. Lo. 2017. The Impact of Coverage on Bug Density in a Large Industrial Software Project. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 307–313. <https://doi.org/10.1109/ESEM.2017.44>
- [3] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2020. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2020), 1–1. <https://doi.org/10.1109/tse.2019.2940179>
- [4] Zimin Chen and Martin Monperrus. 2018. The CodRep Machine Learning on Source Code Competition. arXiv:1807.03200 [cs.SE]
- [5] Lucas Gren and Vard Antinyan. 2017. On the Relation Between Unit Testing and Code Quality. *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)* (Aug 2017). <https://doi.org/10.1109/seaa.2017.36>
- [6] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering (Hyderabad, India) (ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. <https://doi.org/10.1145/2568225.2568271>
- [7] Rafael-Michael Karampatsis and Charles Sutton. 2020. How Often Do Single-Statement Bugs Occur? The ManySSuBs4J Dataset. In *Proceedings of the 17th International Conference on Mining Software Repositories (Seoul, Republic of Korea) (MSR '20)*. Association for Computing Machinery, New York, NY, USA, 573–577. <https://doi.org/10.1145/3379597.3387491>
- [8] Akbar Siami Namin and James H. Andrews. 2009. The Influence of Size and Coverage on Test Suite Effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (Chicago, IL, USA) (ISSTA '09)*. Association for Computing Machinery, New York, NY, USA, 57–68. <https://doi.org/10.1145/1572272.1572280>
- [9] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages. <https://doi.org/10.1145/3340544>