How Often Do Single-Statement Bugs Occur? The ManySStuBs4J Dataset

Rafael - Michael Karampatsis
School of Informatics
The University of Edinburgh
Edinburgh, UK
R.M.Karampatsis@sms.ed.ac.uk

Charles Sutton
Google AI, The University of Edinburgh
and The Alan Turing Institute
charlessutton@google.com

***Abstract*—Program repair is an important but difficult software engineering problem. One way to achieve a "sweet spot" of low false positive rates, while maintaining high enough recall to be usable, is to focus on repairing classes of simple bugs, such as bugs with single statement fixes, or that match a small set of bug templates (Long and Rinard, 2016; Pradel and Sen, 2018) However, it is very difficult to estimate the recall of repair techniques based on templates or based on repairing simple bugs, as there are no datasets about how often the associated bugs occur in code. To fill this gap, we provide two versions of the dataset containing 24412 and 153751 single statement bug-fix changes mined from 100 popular open-source Java Maven projects and from 1000 popular open-source Java projects respectively, annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. We also administer a repository of Maven dependencies for the 100 projects dataset to facilitate tools that require building the projects. We hope that this dataset will prove a resource both for future work in automatic program repair and also for future studies in empirical software engineering. In an initial analysis, we find that for both datasets about 33% of the simple bug fixes match the templates, indicating that a remarkable number of single-statement bugs can be repaired with a relatively small set of templates. Further, we find that SStuBs appear with a frequency of about one bug per 1600-2500 lines of code (as measured by the size of the project's latest version), allowing researchers to make an informed case about the potential impact of improved program repair methods.**

***Index Terms*—Program Repair, Mining Software Repositories, Datasets**

I. Introduction

Fixing bugs in programs, that is, program repair, is one of the core tasks in software maintenance, but requires effort to analyze failed executions, locate the cause of the fault, synthesize a bug fix and validate that the fault has been corrected without introducing new ones [1]. Automatic program repair [2]–[5] attempts to alleviate most of the manual effort of locating and repairing faults. However, a major concern in industry is that linters and program repair methods approaches are required to have high precision without risking achieving high enough recall. As an industrial example Google's Tricorder [6] enforces a false positive rate $< 10\%$.

This work was supported in part by the EPSRC Centre for Doctoral Training in Data Science, funded by the UK Engineering and Physical Sciences Research Council (grant EP/L016427/1) and the University of Edinburgh.

One way to find a "sweet spot" of maintaining high precision with adequate recall is to focus on repairing types of simple bugs, such as one-line bugs, or bugs that fall into a small set of templates, such as mutation operators [2] or other types of predefined templates [3], [4], [7]. However, these have been evaluated on either a relatively small numbers of projects, e.g. 69 defects in 8 applications or on synthetic data. Because of this lack of data, it has not previously been possible to estimate the *recall* of a set of repair templates, that is, the percentage of real-world bugs that can be repaired by one of the templates. Simultaneously to the current work, a larger dataset of one-line bugs has been mined [8], but even this dataset does not attempt to classify bugs into templates.

Aiming to fill this gap, we provide a dataset containing 24412 single-statement bug-fix changes mined from 100 popular open-source Java Maven projects as well as a larger one containing 153751 single-statement bug-fix changes mined from 1000 popular open-source Java projects, annotated by whether they match any of a set of 16 bug templates, inspired by state-of-the-art program repair techniques. The chosen templates aim at extracting bugs that compile both before and after repair as such can be quite tedious to manually spot, yet their fixes are so simple that many developers would call them "stupid" upon realization. We will refer onwards to these bugs as "simple stupid bugs" (SStuBs)[1] and the corresponding dataset as the ManySStuBs4J dataset [2]. Automatic repair of SStuBs is potentially an intermediate step toward more general program repair tools, while already being useful to developers. We also think that SStuBs might be a good start for the evaluation of machine learning based fault localization and repair methods.

An additional distinctive feature of our dataset is that the smaller version is restricted to projects that can be built without manual effort using Maven, which allows for evaluating repair techniques that require building a project and/or running the test suite for projects that provide one. In an initial analysis, we find that 32.05% in the smaller version dataset and 33.52% in the larger version of all of the single-statement bugs that we

[1]The acronym is intended to reflect the fact that, for the authors at least, finding such a bug can feel much like stubbing one's toe.

[2]We distinguish between the two versions by simply referring to them as the small and large version

mine match at least one of the SStuB templates resulting in 9639 and 64026 SStuB instances respectively. This indicates that a remarkable number of singe-statement bugs can be repaired with a relatively small set of templates. In further analysis we also estimated the frequency in lines of code with which these pattern based and general single-statement bugs appear. This estimation is based on the size of the project's latest version and reveals that in the smaller dataset version SStuBs appear with a frequency of about 1 per 1600 lines of code and 1 per 2500 lines of code for the large version. We hope that this dataset can serve as a valuable resource not only for future work in program repair but also for future studies in empirical software engineering.

II. Methodology

We next describe the methodology we employed to build the dataset. Our data generation tools along with documentation and detailed instruction for how to use them are available in a public GitHub repository[3] and the extracted dataset is publicly available via the Edinburgh DataShare DOI: https://doi.org/10.7488/ds/2528.

A. Selecting Appropriate Java Projects

In order to mine a high quality dataset we opted to selecting high popularity projects. For the small version of the dataset we selected the 100 most popular open source Java Maven [9] projects from GitHub as of April 1, 2017. To allow evaluation of repair tools that might require building the projects, we selected only Maven ones because it is easy to automatically download the required dependencies for every project and build it. In contrast, manual downloading of dependencies would require an immense amount of human effort. We also provide these dependencies in a Maven repository. To create a ranking for the projects we downloaded the MySQL dump of GHTorrent [10] up to 1/4/2017. A project's popularity is determined by computing the sum of z-scores of its forks and watchers. Lastly we pulled the projects' head commit by 28/1/2019 and considered commits until that date. The same approach was used to rank projects for the larger version. However, the ranking was calculated using a later dump of GHTorrent from 1/1/2019.

B. Classifying Commits as Bug-Fixing or not

For every project our tool searches historically through all of its commits to locate bug fixing ones. To decide if a commit fixes a bug, we checked if its commit message contains at least one of the keywords: 'error, 'bug, 'fix, 'issue, 'mistake, 'incorrect, 'fault, 'defect, 'flaw, and 'type. This process for estimating bug-fixing commits was previously used by [11] and was shown to achieve 96% accuracy on a set of 300 manually verified commits. It also achieved 97.6% [12] on a set of 384 manually verified commits. We sampled 100 random commits containing SStuBsfrom the small version of the dataset and found it to achieve 94% accuracy. The above process produced a total of 115605 and 883982 commits that were classified as bug-fixing for the small and large dataset versions respectively.

[3]https://github.com/mast-group/mineSStuBs

C. Selecting Bug Fix Commits of Single Statement Changes

We have opted to restrict the dataset to small bug fixes that do not require much code modification to fix. Additionally, we are interested in bugs that are not just syntactic errors but cases where the code compiles both before and after the bug was located and repaired. As we are interested in simple bugs that involve only a single statement, we filter out any commits that either add or delete a Java file. We also filter out commits which make a multiple-statement change at any single position in the Java file. We do *not* filter out commits that make single-line modifications at more than one position in the same file. Similarly to the diff algorithm, we consider a modification as deleting the old lines/statements and then adding the new ones. To estimate whether a modification spans across multiple statements we calculate the diff for each modified Java file, and for each modified chunk, we count how many statements were modified. In the case of blocks each statement in the block's body is counted as different statement. For `if` and `while` statements, we count the condition as a separate statement for this purpose. This method allows to us include fixes to single simple statements that span across multiple lines (e.g., due to stylistic reasons) as a simple fix, unlike a line-based approach. Any commits that modify multiple statements in any single position returned by the diff are dropped while we still maintain commits for which a file's diff contains multiple positions with single statement modifications. In the first case it is not trivial to align the deleted and added statements while it is in the latter. For example, one or more of the deleted statements may have been replaced by multiple of the added ones while simultaneously one or more of the deleted statements may have simply been deleted. We note that our tool ignores any changes to comments, blank lines as well as any formatting changes. Our methodology allows cases where for an example the same expression containing a bug appeared multiple times in the file. This filtering produces almost 13000 and 86769 commits for the two dataset versions. Lastly, the employed methodology works in a similar way to the popular SZZ algorithm [13] and its extensions [14], [15] that have extensively been used to spot fix inducing changes.

D. Creating Abstract Syntax Trees

Each file in the commit that contains one or more bugs is parsed, yielding an abstract syntax tree (AST) of the file before the repair. Then, for each repaired line in the file we extract the AST after applying the repair only on that line and leaving the rest of the lines as is. Each extracted pair of ASTs (original and single fix) only differ on the node(s) for the modified line. By performing a simultaneous depth-first traversal on the two ASTs we locate the first node on which the two ASTs differ.

E. Filtering out Clear Refactorings

Although we filter for bug-fixing changes in Step B, there might still exist changes in the data that do not fix a bug or that do not even produce any behavioural changes. This could happen because the commit-message filter had a false positive, or because the change is tangled [16], and contains a bug-fixing modification along with unrelated ones to other files. To reduce the number of non-fixing changes in the dataset, we observe that there is a class of refactorings that can produce small changes, namely renamings. We filter the files in our dataset to remove such changes. Our method spots variable, function, or class renaming as well as any uses of them across other modified files in the commit. Any refactored lines are excluded from the dataset.

F. SStuB Patterns

We next describe the 16 SStuB patterns. We opted to choose patterns that appear often. Many of these have been used in pattern-based repair and mutation tools [2]–[4], [7]. Here we provide a brief description of each pattern but do not include examples due to page limitations. However, examples for each pattern are provided in the README of the GitHub repository for the mining tool.

- *Change Identifier Used* Checks whether an identifier appearing in some expression in the statement was replaced with an other one. Developers can easily by accident utilize a different identifier than the intended one that has the same type. Copy pasting code is a potential source of such errors. identifier with similar names may further contribute to the occurrence of such errors.
- *Change Numeric Literal* Checks whether a numeric literal was replaced with another one. It is easy for developers to mix two numeric values in their program.
- *Change Boolean Literal* Checks whether a Boolean literal was replaced. True is replaced with False and vice-versa. In many cases developers use the opposite Boolean value than the intended one.
- *Change Modifier* Checks whether a variable, function, or class was declared with the wrong modifiers. For example a developer can forget to declare one of the modifiers.
- *Wrong Function Name* Checks whether the wrong function was called. Functions with similar names and the same signature are usual pitfall for developers.
- *Same Function More Args* Checks whether an overloaded version of the function with more arguments was called. Multiple versions of the same function with different numbers of arguments can often be confused by developers.
- *Same Function Less Args* Checks whether an overloaded version of the function with less arguments was called. For instance, a developer can forget to specify one of the arguments and not realize it if the code still compiles.
- *Same Function Change Caller* Checks whether in a function call expression the caller object for it was replaced with another one. When there are multiple variables with the same type a developer can accidentally perform an operation. Copy pasting code is a potential source of such errors. Variables with similar names can also further contribute to the occurrence of such errors.
- *Same Function Swap Args* Checks whether a function was called with two of its arguments swapped. When multiple arguments of a function are of the same type, if developers do not accurately remember what each argument represents then they can easily swap two such arguments without realizing it. This pattern was also used in DeepBugs [4].
- *Change Binary Operator* Checks whether a binary operand was accidentally replaced with another one of the same type. For example, developers very often mix comparison operators in expressions. A similar pattern was also used in DeepBugs [4].
- *Change Unary Operator* Checks whether a unary operand was accidentally replaced with another one of the same type. For example, developers very often may forget the ! operator in a boolean expression.
- *Change Operand* Checks whether one of the operands in a binary operation was wrong. This pattern was also used in DeepBugs [4].
- *More Specific If* Checks whether an extra condition (&& operand) was added in an `if` statement's condition.
- *Less Specific If* Checks whether an extra condition which either this or the original one needs to hold (|| operand) was added in an `if` statement's condition.
- *Missing Throws Exception* Checks whether the fix added a `throws` clause in a function declaration.
- *Delete Throws Exception* Checks whether the fix deleted an `throws` clause in a function declaration.

G. SStuB Pattern Matching

Finally, each pair of ASTs is checked to detect whether one of the SStuB patterns is satisfied. Each pattern can be expressed as a mutation operation on the original AST that produces the new one. All instances are added to the single-statement dataset, while only those that match SStuB patterns are saved in the SStuBs dataset.

III. ManySStuBs4J Dataset Statistics

The ManySStuBs4J dataset consists of 9639 and 64026 instances of single statement bugs mined from almost 13000 and 87000 bug-fix commits with only single-statement changes respectively for each version. Consequently, on average almost 2 single statement bugs and 0.75 SStuBs were mined per valid commit. The data is saved in JSON files and detailed information is available in the GitHub repository. Each SStuB instance is also annotated with the SStuB pattern satisfied, the project's name, the Java file's name, and the line at which the bug starts. In some cases a statement might fit more than one patterns. In those cases it is counted as separate instances. However, in most cases the patterns are distinct. The statistics for each of the 16 SStuB patterns of the ManySStuBs4J dataset are shown in Table I. Patterns that are similar are grouped together (e.g. patterns that concern functions) and sorted in descending frequency order. The three most common SStuB patterns are *Change Identifier Used*, *Wrong Function Name*, and *Change Numeric Literal*.

We note that the mined bugs have not been annotated by severity and we expect that to vary. Some of the bugs appear in test code. Although bugs in test code will not reach a final product, they can have significant effect on it as they can potentially mask important bugs in it. Such bugs might also be quite tedious to locate as it is very rare to test a test suite and even if we follow that logic we would have to endlessly create tests for the tests. Lastly, as it was recently shown [17] unit tested code does not appear to be associated with fewer failures while increased coverage is associated with more failures.

IV. Research Questions

Although the focus of this paper is the dataset, we perform a simple analysis to support our design decision to focus our new dataset on SStuBs. In order to explore whether the SStuB templates in this work are useful targets for program repair techniques, we asked two research questions.

RQ1. How common are SStuBs in real code?

We measured for each SStuB type the percentage of single statement modifications that are not clear refactorings and fit the pattern. These are visualized in Table I. For each project P we also estimated the following two densities for the mined SStuBs: (a) the number of SStuBs in project P / total lines in P at the final snapshot and (b) the number of SStuBs in project P / total lines added and deleted in P by the final snapshot. Thus, estimating the frequency per line of code modifications in the project's history. That is counting any line that was added or deleted to the project from the start to its latest version. A line modification is counted twice (once as a deletion and once as an addition). Once for deleting the old and once for adding the new line. Comments and empty lines were excluded from these estimations. We found that in the smaller version of the dataset SStuBs appear with densities of about 2540 and 30000 lines of code (LOC) respectively.

We also estimated the same densities for the larger dataset version. We found that such bugs appear with a frequency of about 1600 and 20000 LOC respectively. As a threat to validity, we acknowledge that the number of LOC in the final snapshot may not be the most informative denominator for a measure of bug density, but developing better measures seems a thorny issue that is best left to future work.

RQ2. Can SStuBs be spotted by existing tools such as static analyzers?

We measure the proportion of bugs in our dataset that can be identified by the popular static analysis tool SpotBugs.[4] If SpotBugs reports any bug for the line containing the SStuB then we consider that SpotBugs successfully detected it. We find that SpotBugs could only locate about 12% of SStuBs while also reporting more than 200 million possible bugs when configured to report all warnings, even those with low confidence. This means that a developer would have to look through hundreds of thousands of warnings produced by SpotBugs to locate a single SStuB. This highlights the necessity for tools that are specifically built to detect SStuBs. The scripts used to run and evaluate SpotBugs are also available in our repository.

[4]https://spotbugs.github.io/

TABLE I
Statistics for each SStuB pattern.

Pattern Name	*SStuBs*	*Ratio*	*SStuBs L*	*Ratio L*
Change Identifier Used	3290	13.48%	22773	14.81%
Change Numeric Literal	1178	4.82%	5447	3.54%
Change Boolean Literal	166	0.68%	1841	1.20%
Change Modifier	1028	4.21%	5010	3.26%
Wrong Function Name	1491	6.11%	10179	6.62%
Same Function More Args	807	3.31%	5100	3.32%
Same Function Less Args	185	0.76%	1588	1.03%
Same Function Wrong Caller	196	0.80%	1504	1.00%
Same Function Swap Args	131	0.54%	612	0.40%
Change Binary Operator	327	1.34%	2241	1.46%
Change Unary Operator	174	0.71%	1016	0.66%
Change Operand	127	0.52%	807	0.52%
Less Specific If	220	0.90%	2813	1.83%
More Specific If	203	0.83%	2381	1.55%
Missing Throws Exception	69	0.28%	206	0.13%
Delete Throws Exception	47	0.19%	508	0.33%
TOTAL	7824	32.05%	51537	33.52%

V. Related Work

Several previous data sets of real-world bugs have been curated. Defects4J [18] is a popular dataset consisting 395 Java bugs. Each bug is fixed in a single commit but the fix may modify multiple source code lines. The ManyBugs dataset [19] contains 185 C bugs, a subset of which were used by the GenProg [2], Prophet [3] and SPR [7] papers. Bugs.jar [20] is comprised of 1,158 Java bugs and their patches. These datasets have the disadvantage of being relatively small. More recently, a few larger-scale data sets of small bugs have been created. The combined datasets are the CodRep dataset [21] and the Bugs2Fix dataset [22] resulting in 40289 one-line bugs. These datasets are combined into a single dataset of one line bugs by [8]. Our datasets are of similar size consisting of 24412 and 153751 single-statement bugs. In contrast, our dataset focus on estimating the frequency of SStuB templates, motivated by recent program repair tools and also operates on the statement level, which prevents falsely excluding instances due to formatting or stylistic reasons. Also, the projects from which the small version of our dataset was generated can easily be built using Maven, allowing it to be used to evaluate methods that require a test suite. Lastly, unlike previous datasets, we take additional steps to filter out refactorings, although we acknowledge that such instances might be rare. In our case however, we were able to filter out almost 5000 and 35000 refactored statements for the two dataset versions.

VI. Conclusions

We introduce a new, large-scale dataset of real-world SStuBs, simple one-statement bugs, in Java for the evaluation of program repair techniques. The distinguishing feature of our dataset is that where possible, the SStuBs are categorized into one of 16 bug templates, which are inspired by those considered in state-of-the-art program repair methods. These types of bugs often result in code that compiles, which means that they are particularly interesting for automated repair. We find that SStuBs occur relatively often — one per 1600 LOC in the projects we study — making them potentially a promising evaluation dataset for repair techniques that could be used to estimate their actual recall.

References

[1] M. A. F. Müllerburg, “The role of debugging within software engineering environments,” *SIGPLAN Not.*, vol. 18, no. 8, pp. 81–90, Mar. 1983. [Online]. Available: http://doi.acm.org/10.1145/1006142.1006165

[2] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, “Genprog: A generic method for automatic software repair,” *IEEE Trans. Software Eng.*, vol. 38, no. 1, pp. 54–72, 2012.

[3] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” *SIGPLAN Not.*, vol. 51, no. 1, pp. 298–312, Jan. 2016. [Online]. Available: http://doi.acm.org/10.1145/2914770.2837617

[4] M. Pradel and K. Sen, “Deepbugs: A learning approach to name-based bug detection,” *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 147:1–147:25, Oct. 2018. [Online]. Available: http://doi.acm.org/10.1145/3276517

[5] M. Monperrus, “Automatic software repair: A bibliography,” *ACM Comput. Surv.*, vol. 51, no. 1, pp. 17:1–17:24, Jan. 2018. [Online]. Available: http://doi.acm.org/10.1145/3105906

[6] C. Sadowski, J. van Gogh, C. Jaspan, E. Soederberg, and C. Winter, “Tricorder: Building a program analysis ecosystem,” in *International Conference on Software Engineering (ICSE)*, 2015.

[7] F. Long and M. Rinard, “Staged program repair with condition synthesis,” in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: ACM, 2015, pp. 166–178. [Online]. Available: http://doi.acm.org/10.1145/2786805.2786811

[8] Z. Chen, S. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus, “Sequencer: Sequence-to-sequence learning for end-to-end program repair,” *CoRR*, vol. abs/1901.01808, 2019. [Online]. Available: http://arxiv.org/abs/1901.01808

[9] F. P. Miller, A. F. Vandome, and J. McBrewster, *Apache Maven*. Alpha Press, 2010.

[10] G. Gousios, “The ghtorrent dataset and tool suite,” in *Proceedings of the 10th Working Conference on Mining Software Repositories*, ser. MSR ’13. Piscataway, NJ, USA: IEEE Press, 2013, pp. 233–236. [Online]. Available: http://dl.acm.org/citation.cfm?id=2487085.2487132

[11] B. Ray, V. Hellendoorn, Z. Tu, C. Nguyen, S. Godhane, A. Bacchelli, and P. Devanbu, “On the” naturalness” of buggy code,” ser. ICSE ’16. ACM, 2016.

[12] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “An empirical investigation into learning bug-fixing patches in the wild via neural machine translation,” in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: ACM, 2018, pp. 832–837. [Online]. Available: http://doi.acm.org/10.1145/3238147.3240732

[13] J. Sliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?” in *International Workshop on Mining Software Repositories*. ACM, 2005.

[14] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, “Automatic identification of bug-introducing changes,” in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE’06)*, Sep. 2006, pp. 81–90.

[15] C. Williams and J. Spacco, “SZZ revisited: Verifying when changes induce fixes,” in *Proceedings of the 2008 Workshop on Defects in Large Software Systems*, ser. DEFECTS ’08. New York, NY, USA: ACM, 2008, pp. 32–36. [Online]. Available: http://doi.acm.org/10.1145/1390817.1390826

[16] K. Herzig and A. Zeller, “The impact of tangled code changes,” in *Working Conference on Mining Software Repositories*. IEEE Press, 2013, pp. 121–130.

[17] E. Chioteli, I. Batas, and D. Spinellis, “Does unit-tested code crash? a case study of eclipse,” *arXiv preprint arXiv:1903.04055*, 2019.

[18] R. Just, D. Jalali, and M. D. Ernst, “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*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

[19] C. L. Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, “The manybugs and introclass benchmarks for automated repair of c programs,” *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, Dec 2015.

[20] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, “Bugs.jar: A large-scale, diverse dataset of real-world java bugs,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR ’18. New York, NY, USA: ACM, 2018, pp. 10–13. [Online]. Available: http://doi.acm.org/10.1145/3196398.3196473

[21] Z. Chen and M. Monperrus, “The codrep machine learning on source code competition,” *CoRR*, vol. abs/1807.03200, 2018. [Online]. Available: http://arxiv.org/abs/1807.03200

[22] M. Tufano, C. Watson, G. Bavota, M. D. Penta, M. White, and D. Poshyvanyk, “An empirical study on learning bug-fixing patches in the wild via neural machine translation,” *CoRR*, vol. abs/1812.08693, 2018. [Online]. Available: http://arxiv.org/abs/1812.08693