Building a Probabilistic Model of Statement Replacements for Automatic Bug Fixing based on an Empirical Study of Real Java Bug Fixes

Mauricio Soto, Selva Samuel
School of Computer Science, Carnegie Mellon University, Pittsburgh, PA
mauriciosoto@cmu.edu, ssamuel@andrew.cmu.edu

ABSTRACT

With the rise of automated program repair in the last few years, there are a lot of questions that remain unanswered. We performed a study on 100 open source projects, in which we analyzed the 20 most recent bug fixing commits of this dataset, and found all the replacements made in these 62,470 commits from real life projects in Github.

We found very valuable information to guide automatic software repair, such as the most common and least common replacements made by human developers in order to build a successful patch; we analyzed the most and least common statement to replace others, and the most and least likely statement to get replaced by other. We also analyzed, once you have a statement that you want to replace (a faulty statement), what are the most likely statements to replace it for. These information will help automatic program repair tools to be more maintainable and user friendly, taking it one step closer to what humans developers do to repair their code. We performed an evaluation for our model, that shows via 10-fold cross validation that our model can predict replacements in a 78.52% of the times in the first 3 guesses. Finally we also performed a study to validate the accuracy of the regular expression being normally use to find bug fixing commits.

Keywords

Automatic error repair; Maintainability; Replacements

1. INTRODUCTION

Automatic bug repair is the branch of software engineering that deals with automated ways to repair errors in software without requiring human intervention [5, 6, 7, 9]. There have been several approaches taken towards improving the different methodologies to repair bugs in software automatically [5, 6, 10, 13]. One of the more successful approaches so far has been GenProg [6, 13], a tool which combines stochastic search methods like genetic programming with lightweight program analyses to find patches for real bugs in software. Based on evolutionary computation, GenProg applies four different kinds of possible changes to code in order to find a patch for a given bug. These changes are: (1) delete, (2) replace, (4) swap, and (4) append.

Our research aims to provide guidelines to improve one of these possible changes. The one which remains to be unstudied since it is probably the hardest of them: Replace. It will provide the data necessary for this tool and other approaches to have a guide on what is the way in which human programmers change their code when coding a fix for a bug. This way it will make it more likely for automatic error repair approaches to succeed in finding a patch for a particular error, and also, to provide automatic error repair software with heuristics to make the patches more human-like and therefore more readable and maintainable by human developers.

In this article we perform an empirical study that consists of mining 100 open source projects, and determining the frequency with which human programmers replace different statements to their source code in order to fix a bug.

In this paper, we study bug-fixing commits to Java programs, taken from several thousand human-made bug fixes from Github. We make observations to directly guide future research in automatic repair of Java programs, to increase the success rate of such techniques and the degree to which the patches are human-like and therefore more readable and maintainable by human developers. This is important, because patch quality is an important concern in this area [11].

Related work. Zhong and Su [14] ask some of the same questions we do, on 6 projects. We study a mutation operation that these researchers left out due to its complexity, which is the operator Replace, and we also look at more statement types than the previous work.. Similar to our study in Section 5, researchers have studied AST-level [8] and line-level changes [1] across bug fixing commits. Although it is worth notice that so far none of these approaches have studied in depth the Replace operator. An important study was done in this subject by Soto et al [12] in which they count the number of statements in the before-fix and after-fix versions of the code and try to infer a correlation between the two number of statements. This study avoids this assumption, and instead creates a direct comparison between the two abstract syntax trees to be able to infer more accurately when replacements are being made and create the model with this data.

2. DATASET AND CHARACTERISTICS

For the dataset of this study, we selected the top 100 Java projects in GitHub ranked according to the number of stars. In GitHub, when a user stars a project, it means that the user is:

- Creating a bookmark for easier access
- Showing appreciation to the repository maintainer for their work

Therefore, by selecting the projects with higher number of stars, we are including those Java projects which are

more liked and supported by the community of developers in GitHub, which may have a correlation with having a lot of followers, contributors and activity; and this diversity benefits our study since having a larger diversity of bug fixes, makes the model being built from this data more reliable and more diverse.

From each of these 100 projects, we picked the last 20 bug-fixing commits for two reasons. First, we had time restrictions, and making it with more than 20 commits became very time consuming. As a sample of this, for the first project, we tried getting all the bug fixing commits and cloning the before-fix and after-fix versions of each of the commits. We had to stop this process when the commits were taking 65GB space just for the first project. Therefore we considered that taking the last 20 commits would be a good balance between depth of each project and would also allow us to get the breadth we were looking by in analyzing 100 different projects.

The problem of categorizing which commits are actually bug fixing commits or not is an open problem in the literature. This is usually performed by applying a regular expression to the commit message, and this way inferring if a commit is a bug fixing commit or not depending on the words the commit message has. We used as inspiration the regular expression used by Boa [2] to find bug fixing commits. We used a very similar regular expression to determine this, which is the following:

- "[Ff]ix(ed|es|ing)?(\s)*([Bb]ug|[Ii]ssue|[Pp]roblem)?(s)?"

To address the concern of knowing if this regular expression would get the commits that are actually bug fixing commits or not, we made the study in the following section.

2.1 Commit Validation

In order to build the probabilistic model for statement replacements, it is necessary to ensure that the commits selected using the regular expression specified above were truly bug-fixing commits. In particular, some of the commits whose commit messages matched the regular expression included changes to more than five Java source code files. But prior studies of bug fixes [14] have shown that programmers modify less than five files in order to fix a bug. So, it became necessary to check if these commits which involved modifications to five or more source code files were bug-fixing commits indeed.

We examined if commits that modified five or more Java files were bug-fixing commits or not by manually inspecting such commits. We selected the 50 most recent commits from the commit history of each of the 100 projects in our dataset. Some projects had less than 50 commits in total. We included all of the commits that were made to these projects. Among these commits, we then proceeded to select the ones that involved modification to five or more Java source files. We then manually inspected the commit messages of the selected commits to determine if the changes were being made in order to fix a bug. We included the commits which indicated that they were fixing issues such as a crash, NullPointerException or some other issue identified using a project-specific ID. We excluded the other commits. For example, if the commit message contains the words "prefix" or "suffix", it would match the regular expression that we have used, even though it is not fixing a bug. We excluded all such commits.

Project	Bug-fixing commits	Other commits
platform_frameworks_base	18	1
kotlin	12	2
elasticsearch	23	6
netty	9	4
libgdx	8	3
spring-boot	15	9
spring-framework	12	22
presto	17	7
RxJava	22	10
titan	12	5

Table 1: Distribution of bug-fixing commits in the 50 most recent commits for the top 10 projects with the highest number of commits

The 100 projects that we have used in our study have a total of 62,470 commits. We selected 1,353 of them for performing commit validation. These commits are the ones among the 50 most recent commits for each of the 100 projects in which five or more source code files have been changed. We discovered that 847 out of these 1,353 commits were actually bug-fixing commits. The remaining 506 commits were made for other reasons. Majority of the commits that we have included for building the probabilistic model are indeed bug-fixing commits. Table 1 shows the distribution of bug-fixing commits in the 50 most recent commits for the top 10 projects in terms of the total number of commits.

3. STATEMENT REPLACEMENTS

We build our probabilistic model at the granularity level of statements. So, for each pair of statement kinds, we count the number of times a statement of the first kind has been replaced with a statement of the second kind in the bug-fixing commits. In order to do this, we have used a tool named QACrashFix [4]. QACrashFix generates patches to bugs, particularly ones that result in crashes due to exceptions, by analyzing Q&A sites or programmer forums. It makes use of the fact that programmers working on projects using the same framework (e.g., Android, Hadoop, Spring, etc.) make similar kinds of mistakes related to the use of the framework. If a bug causes an exception, more often than not, programmers get the help of experts in fixing it by posting to Q&A sites. These posts usually include the exception message and stack trace as well. When another programmer encounters a similar exception, they can use QACrashFix which searches for posts made by other programmers on Q&A sites and constructs patches from the solutions posted on the page.

The construction of patches by QACrashFix involves four steps. First, a search query is constructed based on information about the exception. This search query is used to obtain a list of Q&A pages that contain possible fixes by performing a web search. The second step consists of extracting buggy and fixed code snippets from the Q&A pages in the search results. For each pair of buggy and fixed code snippets, QACrashFix generates an edit script that describes how the Abstract Syntax Tree (AST) of the buggy code may be transformed to that of the fixed code. In the third step, potential patches are constructed by customizing the edit

script to the code the programmer is searching a fix for. Finally, patches which fail to compile are eliminated.

We count statement replacements by generating an edit script that transforms the version of a source code file prior to a commit to the version of the same file after the commit. We use the edit script generation component of QACrashFix for this purpose. This component is based on another tool named GumTree [3]. The edit scripts generated by GumTree contain four types of operations: (1) add, (2) delete, (3) update, and (4) move. QACrashFix extends GumTree to add two more operations: (1) replace, and (2) copy. The replace operation contains information about AST nodes which must be replaced with other AST nodes. We only consider replace operations in which both the AST node being replaced and the replacement AST node correspond to Java statements.

4. BUILDING THE MODEL

In order to build the probabilistic model, we gather the data from the 100 projects. From each of these projects we took the 20 latest bug fixing commits, and each of these bug fixing commits consists of a variable number of files, which can vary between 1 to even a couple of hundreds. We ran the scripts to gather the data to build the model for 4 days straight on a laptop with an Intel i7 processor and 8GB RAM. We added all the instances of replacements that we found in our dataset and with these, we modeled the distribution of replacements performed by human programmers when patching bugs.

According to The Eclipse Java development tools (JDT) [1], the java language contains 22 different kinds of statements. We based our model and results on these 22 statement types. Since the names of the statement types are very long, we created abbreviations for all of them so that the table of the model is clearer. The abbreviations for the statement types are the following:

```
As: AssertStatement
Bl: Block
Br: BreakStatement
CI: ConstructorInvocation
Co: ContinueStatement
Do: DoStatement
Em: EmptyStatement
EF: EnhancedForStatement
Ex: ExpressionStatement
Fo: ForStatement
If: IfStatement
La: LabeledStatement
Re: ReturnStatement
SC: SuperConstructorInvocation
Ca: SwitchCase
Sw: SwitchStatement
Sy: SynchronizedStatement
Th: ThrowStatement
Tr: TryStatement
TD: TypeDeclarationStatement
VD: VariableDeclarationStatement
Wh: WhileStatement
```

[1]http://www.eclipse.org/jdt/

5. EVALUATION

We have evaluated our model by determining how likely it is that the model can predict the behavior of real bug fixes that are performed by humans. We have done this by using 10-fold cross validation. By using this technique, we randomly partitioned the 100 projects that we have used into 10 different groups (called folds). Each fold is of the same size and contains 10 projects. We then divided the 10 folds into two sets, namely the training set and the test set. The test set consists of one fold. The remaining nine folds constitute the training set. The model is built using the folds in the training set. This model is validated against the test set by checking how successfully it predicts the data in the test set. This process is repeated 10 times. In each iteration, a different fold is selected for the test set.

In order to measure the prediction performance of the model built from the training set, we select a pair of replacee-replacer statements from the test set. The replacee statement is the statement being replaced and the replacer statement is the one with which the replacee statement is replaced. From the model, we get the top 3 statements which are most likely to be used in place of the replacee statement while a programmer is fixing a bug. If the replacer statement is included in these 3 statements then we consider the model to have made a successful prediction about the statement the programmer has used in place of the replacee statement. Otherwise, the prediction made by the model is considered to be unsuccessful. This process is repeated for each pair of replacee-replacer statements in the test set. We counted the number of times the model was able to make a successful prediction as well as the number of times in which it failed.

For example, suppose that the 10 projects in fold 1 are to be used as the test set in the first run. So in this case, folds 2 to 10 (which are the remaining 9 folds) form the training set and would be used to build the probabilistic model.

As an example, consider the row "Re" (short for "Return") in the model shown in Table 2. As you may notice, the three highest probabilities of statements that can replace a Return statement are:

- Expression Statement (43%)
- If Statement (22%)
- Variable Declaration Statement (19%)

These would be the top three guesses of the model when trying to predict the testing dataset. So if, for example, the testing dataset would be

- Expression Statement: 4
- If Statement: 2
- Variable declaration Statement: 2
- Do Statement: 1
- Continue Statement: 1

In this example, we would get 8 instances of replacements correctly guessed by the model in the first 3 guesses, and 2 instances of replacements incorrectly guessed by the model in the first 3 guesses. Which means that the prediction of this model with this testing dataset had a success rate of 80%.

	As	Bl	Br	CI	Co	Do	Em	EF	Ex	Fo	If	La	Re	SC	Ca	Sw	Sy	Th	Tr	TD	VD	Wh
As	0	0	0	0	0	0	0	0	25	0	25	0	25	0	0	0	0	0	0	0	25	0
Bl	0	0	0	1	0	0	0	0	33	0	15	0	31	0	0	0	0	0	0	0	19	0
Br	0	0	0	0	0	0	0	0	48	0	12	0	0	0	40	0	0	0	0	0	0	0
CI	0	10	0	0	0	0	0	0	0	0	0	0	10	80	0	0	0	0	0	0	0	0
Co	0	0	0	0	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0
Do	0	0	0	0	0	0	0	0	100	0	0	0	0	0	0	0	0	0	0	0	0	0
Em	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
EF	0	12	0	0	0	0	0	0	54	4	12	0	0	0	0	0	0	0	12	0	8	0
Ex	0	26	3	0	0	0	0	1	0	0	13	0	3	0	0	0	0	1	3	0	50	0
Fo	0	8	0	0	0	0	0	17	0	0	0	0	75	0	0	0	0	0	0	0	0	0
If	0	19	2	0	0	0	0	1	52	0	0	0	9	0	0	0	0	0	3	0	14	0
La	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Re	0	2	0	2	5	0	0	0	43	0	22	0	0	2	2	0	0	3	2	0	19	0
SC	0	8	0	54	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	15	0
Ca	0	0	30	0	0	0	0	0	20	0	10	0	30	0	0	0	0	0	0	0	10	0
Sw	0	33	0	0	0	0	0	0	0	0	67	0	0	0	0	0	0	0	0	0	0	0
Sy	0	0	0	0	0	0	0	0	10	0	20	0	0	0	0	0	0	0	70	0	0	0
Th	0	0	0	0	0	0	0	0	71	0	0	0	29	0	0	0	0	0	0	0	0	0
Tr	0	0	0	0	0	0	0	0	26	0	9	0	4	0	0	0	30	0	0	0	30	0
TD	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
VD	0	6	0	0	0	0	0	1	81	1	6	0	2	0	0	0	0	0	2	0	0	0
Wh	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Table 2: Likelihood of replacing a statement type (row) by a statement of another type (column), for Java. [Numbers in the table are percentages.]

Test Fold No.	Instances correctly predicted	Instances incorrectly predicted	Success rate
1	72	11	86.75%
2	82	43	65.60%
3	64	23	73.56%
4	752	423	64.00%
5	83	42	66.40%
6	81	18	81.82%
7	713	6	99.17%
8	186	25	88.15%
9	37	9	80.43%
10	23	6	79.31%

Table 3: 10-fold cross validation of the probabilistic model

Following this approach, we performed a 10 fold cross validation in our model and the result of each of the runs taking a different fold as testing data is detailed in Table 3.

The mean of the 10 different success rates per each of the 10 folds gives us a mean success rate of 78.52% of success in predictability of the model being able to guess the correct statement by which each statement is being replaced. The authors of this paper think that if we analyze the how likely the model is to predict the 10 different folds using the first 5 guesses of the model instead of the first 3 guesses, then the success rate of the model would have been closer to a 95%. But this couldn't be done due to time restrictions, and it is advised to be done as future work.

6. CONCLUSION AND FUTURE WORK

Our work is publicly available to be downloaded, extended and reproduced by anyone.[2] As some possible extensions of our work, the authors of this paper think that it would be valuable to create a model with a larger sample of data, both in depth and breadth, but the authors of this paper consider that in this case it is more valuable to give a major importance to breadth, since the model benefits more from diversity of different programmers and different projects, and not as much from a repeated pattern of bug fixing in a particular set of projects. This will also help to the fact that the model will be less likely to overfit the training data. As an extension of section 2.1, we would like to dig deeper into why is the number of actual bug fixing commits so low, and what can be done to mitigate this. Since this error is causing a lot of noise to be introduced to every study that deals with similar regular expressions and who are trying to caught bug fixing commits, and they are getting almost a 40% of their population to be not actual bug fixing commits. As a next step for this study, it would be interesting to insert this model into GenProg's code and compare its performance with previous runs and also to different approaches to see how well it works with our approach.

7. ACKNOWLEDGMENTS

8. REFERENCES

[1] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. In *Proceedings of the 2013 IEEE International Conference on Software Maintenance*, ICSM '13, pages 230–239, Washington, DC, USA, 2013. IEEE Computer Society.

[2] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-large-scale Software Repositories. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 422–431, Piscataway, NJ, USA, 2013. IEEE Press.

[3] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and Accurate Source Code Differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 313–324, New York, NY, USA, 2014. ACM.

[2]https://github.com/mausotog/ReplacementsEmpiricalStudy

[4] Q. Gao, H. Zhang, J. Wang, Y. Xiong, L. Zhang, and H. Mei. Fixing Recurring Crash Bugs via Analyzing Q&A Sites. In *Proceedings of the 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ASE '15, pages 307–318, Washington, DC, USA, 2015. IEEE Computer Society.

[5] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 802–811, Piscataway, NJ, USA, 2013. IEEE Press.

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.*, 38(1):54–72, Jan. 2012.

[7] F. Long and M. Rinard. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 298–312, New York, NY, USA, 2016. ACM.

[8] M. Martinez and M. Monperrus. Mining Software Repair Models for Reasoning on the Search Space of Automated Program Fixing. *Empirical Softw. Engg.*, 20(1):176–205, Feb. 2015.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 448–458, Piscataway, NJ, USA, 2015. IEEE Press.

[10] K. Pan, S. Kim, and E. J. Whitehead, Jr. Toward an Understanding of Bug Fix Patterns. *Empirical Softw. Engg.*, 14(3):286–315, June 2009.

[11] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 24–36, New York, NY, USA, 2015. ACM.

[12] M. Soto et al. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. *Mining Software Repositories Challenge Track*, 2016.

[13] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically Finding Patches Using Genetic Programming. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 364–374, Washington, DC, USA, 2009. IEEE Computer Society.

[14] H. Zhong and Z. Su. An Empirical Study on Real Bug Fixes. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 913–923, Piscataway, NJ, USA, 2015. IEEE Press.