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 couple of years, there are a lot of questions to 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 computer science that deals with automated ways to repair errors in software(e.g., [5, 6, 7, 9]). There have been several approaches taken towards improving the different methodologies to repair bugs in software automatically [1, 3, 4, 12], one of the most accepted approaches so far has been GenProg [3, 12], a tool which combines stochastic search methods like genetic programming with lightweight program analyses to find patches for real bugs in existing software. This evolutionary program repair tool applies four different kinds of possible changes to code in order to find a patch for a given bug. These changes are the following: Delete, Replace, Swap and 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 [10].

Related work. Zhong and Su [13] 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 [2] 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 [11] 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 project we picked the 100 Java projects in Github with the most 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 picking the projects with more stars we are taking the Java projects more liked and supported by the

community and developers of 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 65 gigs of 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 [robert] 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.

aaaaa aaaaa

Project	Bug fixing commits	Other commits
react-native	18	6
elasticsearch	23	6
RxJava	22	10
okhttp	11	19
picasso	16	9
MPAndroidChart	15	1
android-async-http	11	6
guava	14	11
spring-framework	12	22
libgdx	8	3

Table 1:

aaaaa aaaaa aaaaa aaaaa aaaaa aaaaa aaaaa aaaaa aaaaa

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.

3. REPLACEMENTS AND QACRASHFIX

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 8 gigs of 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
```

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

```
Sy: SynchronizedStatement
Th: ThrowStatement
Tr: TryStatement
TD: TypeDeclarationStatement
VD: VariableDeclarationStatement
Wh: WhileStatement
```

5. EVALUATION

For the evaluation section we wanted to know how likely is it that our model can predict the behavior of real bug fixes performed by humans. Therefore we decided to evaluate our model using 10-fold cross validation. By using this technique we randomly divided our 100 projects into 10 different groups (called folds), and then we make 10 runs, each run picking one of the folds as the testing data, and the rest of them being the training data.

For example, in the first run, we take the 10 projects in fold 1 and we are going to use those as the testing data. Meanwhile, we will build a pseudo probabilistic model from the remaining 9 folds, and then we will test how likely is it that the pseudo probabilistic model is capable of predicting the behavior of the testing data. We do this 10 times, each of them taking a different fold as the testing set.

Our interpretation of "predicting" for this study is the following: The model "predicts" an instance of a replacement in the testing dataset when given the statement to replace (replacee) it is able to guess with which statement it was replaced by (replacer) correctly within the first three guesses (the three highest percentages of probabilistic replacement for that statement type in the model).

As an example, consider the row "Re" (short for "Return") in the model shown in Table 1. 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%.

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 2.

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. DISCUSSION 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] *Automatic patch generation learned from human-written patches*, 2013.

[2] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and M. Di Penta. LHDiff: A Language-Independent Hybrid Approach for Tracking Source Code Lines. *ICSM*, pages 230–239, 2013.

[3] L. G. Claire, N. ThanhVu, F. Stephanie, and W. Westley. Genprog: A generic method for automated software repair. *IEEE Trans. Software Engineering*, 2012.

[4] K. P. et al. Towards an understanding of bug fix patterns. *Empirical Software Engineering*, 2009.

[5] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *ICSE*, pages 802–811, 2013.

[6] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Transactions on Software Engineering*, 38(1):54–72, 2012.

[7] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, 2016.

[8] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of

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

	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.

Testing in Fold	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

automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.

[9] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for simple program repairs. In *ICSE*, 2015.

[10] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, pages 24–36, 2015.

[11] M. Soto et al. A deeper look into bug fixes: Patterns, replacements, deletions, and additions. *Mining Software Repositories Challenge Track*, 2016.

[12] W. Westley, N. ThanVu, L. G. Claire, and F. Stephanie. Automatically finding patches using genetic programming. *International Conference on Software Engineering (ICSE)*, 2009.

[13] H. Zhong and Z. Su. An empirical study on real bug fixes. In *ICSE*, pages 913–923, 2015.