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

A considerable amount of research has been done in the area of automatic bug repair [7, 8, 9, 11, 12, 15] with the goal of developing automated techniques to repair errors in software without requiring human intervention. One of the more successful approaches so far has been GenProg [8, 15], 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 three different kinds of possible changes to code in order to find a patch for a given bug. These changes are: (1) delete, (2) replace, and (3) append. However, GenProg is limited by the fact that it is possible to generate candidate patches which fail to compile because it applies the mutation operators (delete, replace and append) in a random manner.

In addition, a considerable proportion of these works target programs written in C. Indeed, researchers targeting Java often start from or compare against Java-based implementations of techniques originally implemented for C [3, 7]. This is an oversight because Java and C are very different languages.

One way of addressing the limitations discussed above is to make automatic bug repair tools mimic the bug-fixing behavior of human programmers. This can be done by building a probabilistic model of the actions taken by programmers when they fix bugs.

In this paper, we address the challenge of generating effective patches for bugs in Java programs whose fixes require statement replacements. We build a probabilistic model based on the statement replacement actions of human programmers in fixing bugs in Java programs. When a statement has to be replaced with another to fix a bug, the probabilistic model specifies the most likely candidates for the replacement operation. To build the probabilistic model, we conducted an an empirical study of 100 open source Java projects in GitHub and determined the frequency with which human programmers replace different statements in their source code in order to fix a bug. Our model can 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 [13].

Related work. Zhong and Su [16] have studied the nature of bug fixes with a view toward improving existing program repair techniques and to understand their limitations. However, their study was limited to bug fixes from six Java projects. Our study is limited to replacements, but we study a significantly larger number of projects and we look at more statement types. Similar to our study, researchers have studied AST-level [10] and line-level changes [1] across bug fixing commits. Although the granularity differs, our study is also novel with respect to the number of projects studied. Barr et al. [2] studied changes to Java programs to understand the "Plastic Surgery Hypothesis" underlying certain types of program repair, an orthogonal concern. Kim et al. [7] manually analyze changes to Java programs to inform automated repair, and show that doing so results in higher-quality repairs. Their results motivate studies of human repairs, as it may result in better patches. Long and Rinard learn probabilistic models from bug fixes to C code [9]. Their model incorporates all possible operations that may be performed in order to fix a bug. Our model, on the other hand, describes only the replace operations that programmers may perform for fixing a bug. Moreover, it does not inform a new

technique. Rather, it serves as a starting point for research on the repair of Java bugs. Soto et. al. [14] use a heuristic to count statement replacements because of limitations of the Boa infrastructure [4] in determining differences between file versions. We are able to count statement replacements in a more direct manner.

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. This includes projects like the Spring framework with which a lot of good developers are associated. As a result, building the probabilistic model based on a study of these projects would provide the benefit of making the model more reliable.

We restricted ourselves to studying only 20 recent commits because of time and storage limitations. Analyzing just 20 commits per project took a total of four days. Moreover, storing the before-fix and after-fix versions of each file in every commit for one project with about a 1000 commits required approximately 65GB of disk space. For these reasons, we have deferred a study of the full commit history of each of the projects as future work.

Determining whether a commit is a bug-fixing commit or not is usually done by applying a regular expression to the commit message. This method relies on the presence of certain words (that indicate that this commit might be related to a bug fix) and their order in the the commit message. We have used a regular expression that is similar to the one used by Boa [4] in order to find bug-fixing commits. The regular expression that we have used is as follows:

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

In order to ensure that the commits whose messages matched the above regular expression are actually bug-fixing commits, we performed 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 [16] 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 [6]. 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 [5]. 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 a probabilistic model of statement replacements in bug fixes, we analyzed the 20 most recent bug-fixing commits in each of the 100 projects that we have selected for this study. The script for building the model by analyzing the commits took 4 days to run to completion on a laptop with an Intel Core i7 processor and 8GB RAM.

As mentioned in section 3, we used QACrashFix to determine which statements have been replaced with which other statements. Since QACrashFix uses the Eclipse Java Development Tools (JDT)[1] to parse code snippets, we have used the statement types used in the Abstract Syntax Tree (AST) generated by JDT as the statement types in our model. The JDT classifies statements in the Java language into 22 categories. We based our model and results on these 22 statement types. Since the names of the statement types are long, we have abbreviated them for easy reference. The statement types and their respective abbreviations are as follows:

```
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/

We built the probabilistic model by determining the instances in which statements of a particular type (replacee type) have been replaced to fix a bug. Among those instances, we determined the type of the statement being used as the replacement statement (replacer type). We then counted the number of times a statement of the replacer type has been used to replace a statement of the replacee type. By following this process by considering each of the 22 statement types described above as the replacee type, we built the model that is shown in Table 2. The statement types along the rows of the table are replacee types and the statement types along the columns are replacer types. The values shown in the table are percentages. For example, the first row of the table indicates that in all cases where an assert statement was replaced to fix a bug, it was replaced by an expression statement 25% of the time, by an "if" statement 25% of the time, by a return statement 25% of the time and by a variable declaration statement 25% of the time.

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

	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.]

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. Let us suppose that we obtained the model shown in Table 2. Suppose now that in the test set an expression statement (`Ex`) is replaced with an "if" statement (`If`) for fixing a bug. From the model, the three most likely statements that can replace an expression statement are:

- Variable Declaration Statement (`VD`): 50%
- Block Statement (`Bl`): 26%
- If Statement (`If`): 13%

Since "if" statements are one of the top three statement types that can replace an expression statement, our model has made a successful prediction in this case. On the other hand, if an expression statement was being replaced with, say, a try statement, then in this case, the model would have failed to correctly predict programmer behavior in fixing the bug.

Following this approach, we performed a 10-fold cross validation of our model. The result of each of the iterations taking a different fold as test set is shown in Table 3. The mean of the 10 different success rates in each of the 10 folds gives us a mean success rate of 78.52% for the model's ability to predict which statement is likely to be used by a programmer to replace a particular statement with when they are fixing a bug. We suspect that the success rate of the model would have been higher if we had used the top five statements instead of three to make the prediction. But we were not able to do this because of time limitations and plan to do this as future work.

6. CONCLUSION AND FUTURE WORK

In this work, we have begun to address two limitations of current automated program repair tools that are based on the generate-and-validate approach: (1) These tools generate a lot of patches that fail to compile. (2) Most of these tools are targeted to fixing bugs in C programs. In particular, our work aims to improve automated bug repair approaches for Java. For this purpose, we have built a probabilistic model that mimics the statement replacement actions of human programmers when they fix bugs in Java programs. We have evaluated our model using a 10-fold cross-validation approach. Our work is publicly available for download[2].

There are a number of ways in which this work can be extended. First, a larger number of projects and a larger number of commits for each project needs to be considered for building the probabilistic model. As stated earlier, we restricted the number of projects and commits in this study because of time and storage limitations. Analyzing a larger number of projects and commits would yield a better model.

Second, the effect of using a higher threshold for the number of prediction statements on the prediction performance of the model needs to be evaluated further. As mentioned in section 5, we have used the 3 most likely statements according to our model when we need to predict the type of the statement that should be used to replace a given statement in order to fix a bug. We suspect that using more number of statements would lead to better prediction.

Third, it would be interesting to incorporate this model into various generate-and-validate program repair tools, par-

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

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

ticularly GenProg in order to study the effectiveness of the model in generating patches.

7. 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] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 306–317, New York, NY, USA, 2014. ACM.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic Repair of Buggy if Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, CSTVA 2014, pages 30–39, New York, NY, USA, 2014. ACM.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo. A Deeper Look into Bug Fixes: Patterns, Replacements, Deletions, and Additions. *Mining Software Repositories Challenge Track*, 2016.

[15] 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.

[16] 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.