A deeper look into bug fixes: patterns, replacements, deletions, and additions

Mauricio Soto
Carnegie Mellon University
Pittsburgh, PA
mauriciosoto@cmu.edu

Chu Pan Wong Carnegie
Mellon University
Pittsburgh, PA
email@domain.com

Ferdinand Lastname
Carnegie Mellon University
Pittsburgh, PA
email@domain.com

Claire Le Goues
Carnegie Mellon University
Pittsburgh, PA
clegoues@cs.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 the Github dump of September 2015 in which we analyzed all the fixing revisions of this dataset, and found all the replacements made in these 46,301,429 files 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.

Keywords

Automatic error repair; Maintainability; Human-like patches

1. INTRODUCTION

Automatic bug repair is the branch of computer science that deals with automated ways to repair errors in software. There have been several approaches taken towards improving the different methodologies to repair bugs in software automatically [4][7][6] [3], one of the most accepted approaches so far has been GenProg [7][6], which is an evolutionary program repair tool and 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 three out of four of these possible changes: Delete, Replace and Append. 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 study the frequency with which human programmers replace, delete and append different statements to their source code in order to fix a bug.

2. METHODOLOGY

Our methodology is applied to the latest official data dump of Github as of September 2015 provided by [1].

2.1 Data Collection

We used the Boa platform [1] to be able to query the Github repository looking for patterns that humans use to patch errors in the their code. Our main goal is to provide automatic error fixing approaches with the data necessary to be able to replace, delete and append statements in the patches being created by them.

Our Boa queries are divided in two types:

2.1.1 Replace Query

The replace query is a query in which first we set the project that we are going to use, which is the Github dump of September 2015 provided by [1].

We then create a visitor to traverse all the nodes of this repository. First thing the visitor does is to ask weather the node is a Revision, and if it is, then it asks if it is a fixing revision. Therefore we are analyzing only files that were already committed and are now being reviewed to fix an error. We use Boa's infrastructure to do this by visiting the node they label as "Revision" and then calling the function defined by Boa "isfixingrevision()".

A replacement needs two statements, one that is deleted and another one that is inserted. We then go through all the

Table 1: Amount of appearances per replace kind

	Assert	Break	Continue	Do	For	If	Label	Return	Switch	Switch	Synchronized	Throw	Try	TypeDecl	While
Assert	-	340	232	27	517	1764	14	1260	211	191	82	1004	352	0	206
Break	455	-	1254	176	3757	9096	90	10538	771	455	672	3123	3660	18	2278
Continue	229	1384	-	221	1909	3094	85	3486	772	635	233	1227	1493	30	1086
Do	32	205	171	-	474	648	12	543	106	92	44	227	325	4	558
For	505	3366	1516	316	-	14547	56	14924	2791	1852	1044	5642	5796	41	6582
If	1402	8820	2430	476	13703	-	247	30489	8639	5710	2472	8960	13714	84	5622
Label	19	44	46	6	56	255	-	270	44	22	5	80	53	2	33
Return	1219	8603	3009	531	12620	28536	164	-	6088	3657	2021	15346	11486	71	4684
Case	298	839	506	125	3002	8039	33	6261	-	126	649	2206	2784	24	1864
Switch	281	533	379	56	1997	5175	22	4090	126	-	212	1571	1311	7	878
Synchronized	79	606	289	66	1122	3070	158	2853	562	256	-	1303	1274	8	466
Throw	821	2844	1195	197	5993	10664	113	16688	2002	1446	936	-	4443	37	1934
Try	440	3976	1254	214	6414	16294	45	14119	3178	1509	1243	4795	-	46	2218
TypeDecl	2	35	15	1	49	83	1	137	21	9	4	35	46	-	11
While	301	2295	1016	937	8257	6587	41	6280	1866	766	606	2028	2677	27	-

files that have been changed in this data set asking for two conditions: it must be a fixing revision and there must have been a previous version of this file before. If these conditions are met then we initialize two counters: one for the first statement we are analyzing (the one that was deleted), and another one for the second statement we are analyzing (the one that was inserted).

At this point we visit each statement in the previous state of that changed file, before the change was applied and we count the amount of appearances of the first statement we are analyzing (the one that was deleted), and then we count the amount of appearances of the second statement we are analyzing (the one that was inserted). We save these values, and do the same for the latest version of the file and save these values as well.

We then compare these results to see if either the first and second statements being analyzed increased or decreased. Depending on these we have two conditionals:

If the amount of occurrences of the first statement decreased and the amount of occurrences of the second statement increased on the same file, then we say that the first statement was replaced by the second statement in that file. Likewise, if the amount of occurrences of the first statement increased and the amount of occurrences of the second statement decreased on the same file, we say that the second statement was replaced by the first statement in that file. We count the amount of files in which this happens for each of these two cases.

We need to test this with all the possible combinations of two different statement types since we want to know what is the probability of an specific statement to be replaced by another specific statement. For example, we want to know how often does a For loop gets replaced by a While loop in a successful human written patch, or how often does a Break statement gets replaced by a Continue statement in a successful human written patch.

Because of this we have created Table 1 which records the amount of replaces happening from one kind of statement to another. It is a matrix of fifteen by fifteen slots that describes which statements are being replaced by which other. The numbers in the slots represent the amount of files in

which that replacement took place. In the first column we have the list of statements that are being replaced, and on the top row we have the list of statements for which that statement was replaced for. For example, if we take the intersection of row 5 (DoStatement row) and column 2 (AsserStatement column), we have a 32, which means that from the 46,301,429 files categorized by Boa as fixing revisions in Github in the month of September 2015, only 32 of them replaced a DoStatement for an AssertStatement.

It is important to notice that this analysis doesn't consider replacing an statement kind for a different version of the statement kind, since it is counting the amount of appearances of each statement kind. So, if for example, the developer modifies the condition inside of an if statement, the amount of if statements in the file is going to remain the same

2.2 Data Analysis

Our queries were ran on the dump of Github of September 2015. In these real life projects we found 46,301,429 files being fixed. In these files, there is a total of 517,281 statement kinds being replaced, and 511,198 replacing these statements kinds.

We analyzed each of these files looking for statement kinds being either replaced by other kinds, deleted or appended to the file.

We found that the most common Replacer statement kind, meaning the statement kind that most commonly replaces others is the **If Statement** with 101,366 appearances. The least common Replacer, meaning the statement kind which is less likely to replace other statement kinds is the **Type Declaration Statement** with 447 appearances.

The most common Replacee, which is the most likely statement kind to be replaced by another statement kind is the **Return Statement** with 111,938 appearances. The least common Replacee, which is the least common statement to be replaced by another statement kind is the **Type Declaration Statement** with 399 appearances.

2.2.1 Likeliness of replacement

This study is directly applicable to automatic software

repair. Most common tools for automatic software repair [4][7][6] first find a faulty statement using several different fault localization techniques [5], and once they have that faulty statement, they apply different kinds of edits to it, usually at random. Which makes it equally likely to replace an statement for another without looking at the kinds of statements being replaced.

With this study, we are able to make a more accurate guess, when having a known faulty statement, to which statement it is more likely to replace it for. By performing simple math with the data of Table 1, we can find the most likely replacements that will end up being a successful patch.

For example, if we have a faulty statement that happens to be a For Statement. Then we can look up in Table 1 the For Statement Column (6th column). If we add all the values in that column, we get 59,870 changes in which a For Statement was replaced by some other kind of statement. Then we calculate the percentage of each of the values in the column to see the likeliness of each of the statements to replace it for, and in this example we see that the top 3 most likely statements to replace a For Statement are an If statement with a 22.88% likeliness, followed by a return statement with a 21.08%, and third is a While statement with a 13.73% of likeliness to replace a For Statement.

### 2.2.2 Most common replacements

Our results show that the most common replacement is when developers replace a Return statement for an If statement. This showed up in 30,489 files in the code database. The second most common replacement is an If statement being replaced by a Return statement, which had 28,536 appearances in the code database. Third to that, is a Return statement being replaced by a Throw statement.

Most common replacements
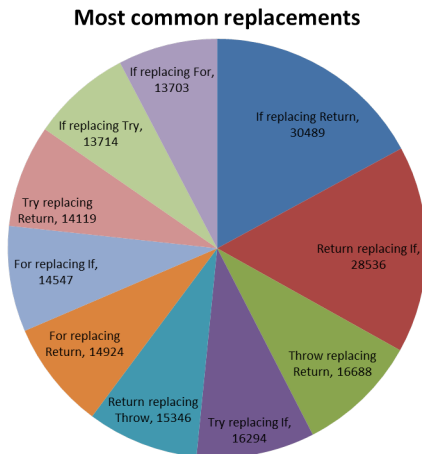
2.2.3 Least common replacements

On the opposite side of the data, we can see that the least common replacement was an Assert statement replacing a Type Declaration statement. This was the only replacement to have zero appearances in our search through the 46,301,429 files being fixed. Close to it, as the second least common replacement was replacing a Do statement for a Type Declaration statement, with one appearance in our code database, and on third place, the replacement of a Label being replaced by a Type Declaration Statement had one

appearance in our code database.

Least common replacements
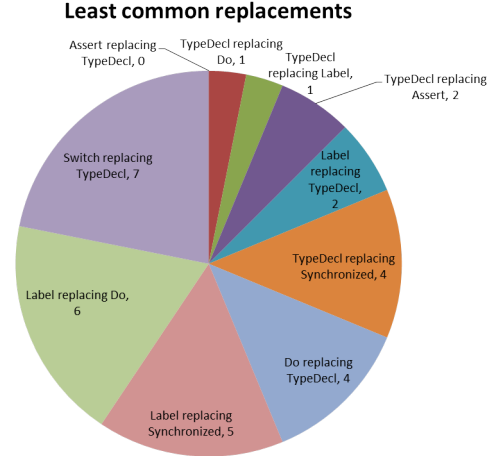
3. RELATED WORK

A similar study was performed by Dongsun Kim et al. [4] in which they look for the most common ways in which programmers patch bugs in software. The researchers developed a variation of the tool Genprog [7][6] with several different templates resembling patters programmers use to patch bugs.

A good complement for this paper is [2], where they search additions and deletions in a smaller data set. In this paper, we focus on the replacements, which have been left out of the analysis of [2].

4. CONCLUSIONS

The findings of our study provide a set of useful guidelines for automatic program repair tools with which their search for a patch can be now improved by having the knowledge of how likely it is to replace a faulty statement by the different kinds of statements available.

5. REFERENCES

[1] Robert Dyer et al., *Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories*, 35th International Conference on Software Engineering; ICSE 2013, San Francisco CA, USA

[2] Hao Zhong, Zhendong Su *An Empirical Study on Real Bug Fixes*, In Proceedings of ICSE 2015, Firenze, Italy, May 16-24, 2015.

[3] Kai Pan et al. *Towards an understanding of bug fix patterns*, Journal Empirical Software Engineering Volume 14 Issue 3, June 2009 Pages 286 - 315

[4] Dongsun Kim et al. *Automatic patch generation learned from human-written patches*, Proceeding ICSE '13 Proceedings of the 2013 International Conference on Software Engineering

[5] Zachary P.Fry and Westley Weimer *A human study of fault localization accuracy*, ICSM '10 Proceedings of the 2010 IEEE International Conference on Software Maintenance

[6] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, Westley Weimer *GenProg: A Generic Method for*

Automated Software Repair. IEEE Trans. Software Engineering 38(1): 54-72 (January/February 2012)

[7] Westley Weimer, ThanVu Nguyen, Claire Le Goues, Stephanie Forrest *Automatically Finding Patches Using Genetic Programming.* International Conference on Software Engineering (ICSE) 2009: 364-374