Using a probabilistic model to predict bug fixes

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

Claire Le Goues
Carnegie Mellon University
Pittsburgh PA
clegoues@cs.cmu.edu

***Abstract*—The abstract goes here.**

I. Introduction

Automatic program repair has attracted a lot of attention in the last several years due to its ability to tackle complex and expensive tasks such as being able to localize errors and fixing code with minimal interaction with human programmers. One of the most well known approaches of automatic program repair is the generation and validation approach. In this approach we start with source code that has one or several bugs on it; and we have a test suite which has passing test cases and failing test cases. The passing test cases confirm the expected behavior of the program, and the failing test cases expose the deviation of the expected behavior with the actual behavior.

These approaches have a wide variety of mutation operations, which are small changes that can be applied to the source code in order to modify its behavior. In order to pick which mutation operator will be used in a specific location, we first pick a fault location, which is a point in the code where we think the fault is located. This can be done in various ways and there is a whole set of possible approaches that can tell you with different levels of certainty what location to pick. This is not the step that we will be analyzing in this paper. We will be analyzing the next step: After we have a fault location, then it goes through a process in which it is analyzed which mutation operators can be applied to that specific location, or if the context of that location doesn't allow some mutation operators to be used in a particular location.

Once this process is finished, we have a set of possible mutation operators that can be applied to a location. At this point, it is chosen at random which mutation operator will be used from the set of mutation operators that can be applied at this location.

In order to pick the mutation operator, the probabilities of picking each of them are equally distributed; this means that from the set of mutations that can be applied to this location, all of them have the same probability of being chosen. Where, in reality, some mutation operators are much more common than others and the fact that we are choosing from them equally is making us often pick mutation operators that are less likely to actually fix the bug we are trying to fix.

In order to tackle this problem we have developed a probabilistic model which entails different probabilities for each of the mutation operators based on empirical data that describes how to human programmers fix their code.

With the probabilistic model we are able to chose from the set of possible mutation operators that can be applied to a particular location using the probabilities which describe how often do each of the mutation operators is able to fix bugs when used in real projects. If human programmers are more likely to use a particular mutation operator, then that mutation operator will have a higher probability of being picked than other mutation operators.

II. Background

One of the most well known approaches to automatic program repair is the generate and validate approach which is detailed in Figure 1. In this approach we start with source code that has one or various bugs to be fixed, and a test suite which contains passing test cases to assure the expected behavior of the program, and failing test cases which expose the behavior of the program that is expected but the output of the expected behavior does not match the output of the source code as is.

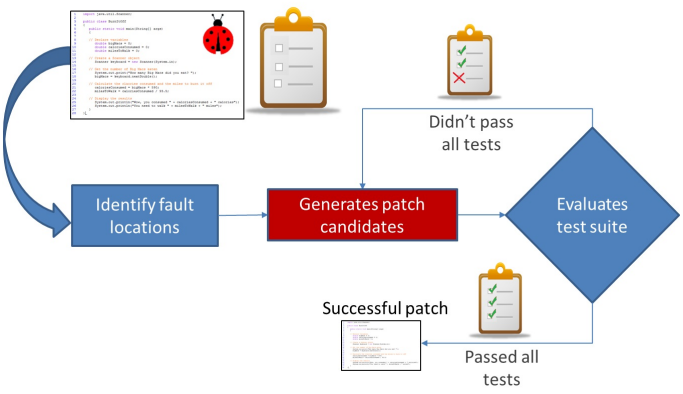

Fig. 1. Generate and validate approach

The first step of this process is to localize the error to a particular statement. There is an extensive list of possible approaches in literature in order to perform this tasks **List several fault loc papers**, but this paper will not focus on this particular step. Out main focus will be the next step: Generating candidate patches.

Once we know where the fault is located we proceed to create what we call "Candidate patches". Candidate patches are variations of the original program which may or may not be a patch for the bug(s) we are trying to fix. In order to create candidate patches we need to apply one or several mutation operators to the fault location.
There are several possible mutation operators that can be applied to a location in order to modify the behavior of a program. In order to analyze this we have taken mutation operators from two of the most successful approaches in program repair that use generate and validate methodology: Genprog **reference to Genprog paper** is a very well known and widely used tool that modifies statements in order to create candidate patches. PAR **PAR Reference** is a successful approach that bases their patch generation technique in creating a list of commonly used templates of the most used changes that human programmers perform in order to fix bugs in their source code. We have denominated these two techniques: General mutations **not sure if this is the best name for this** and Template based mutations.

A. General mutations

Claire Le Goues et al. have created a very successful approach which they denominate GenProg **reference to genprog**, which is a tool that modifies statements of source code in order to repair programs. A statement is the smallest standalone element that expresses some action to be carried out in a programming language. This tool is able to delete, append or replace statements in order to create candidate patches.
Once the tool has a set of candidate patches, it will run the test suite on each of those candidate patches. If it is able to find a candidate patch that can pass all the test cases in the test suite, that candidate patch will be considered a patch of the bug. If it is not able to pass all the test cases in the test suite, then it will go back to creating more candidate patches and will continue to do so until it reaches a stopping point, which may be a certain user defined number of generations or a user defined clock time limit.

B. Template based mutations

Dongsun Kim et al. have created the approach which they denominate Pattern-based Automatic program Repair(PAR), in which they examined a large number of human created patches and have abstracted 16 different templates to be the most commonly used changes that human programmers do in order to fix their code: https://sites.google.com/site/autofixhkust/home/fix-templates The 16 templates which we consider in this category are the following:

1) Null Checker
2) Parameter Replacer
3) Method Replacer
4) Parameter Adder and Remover
5) Object Initializer
6) Sequence Exchanger
7) Range Checker
8) Collection Size Checker
9) Lower Bound Setter
10) Upper Bound Setter
11) Off-by-one Mutator
12) Class Cast Checker
13) Caster Mutator
14) Castee Mutator
15) Expression Changer
16) Expression Adder

III. Building the model

The first model we built was created based on the 200 Java projects in Github with the most stars. When users star a project they are creating a bookmark for easier access, and showing appreciation to the repository maintainer for their work. Projects with more stars are more popular among the developers of Github and therefore more likely to be reviewed by other developers and have higher standards of code quality than projects with less stars.
Once we have checked out the 200 Java projects with the most stars, we checkout the last 100 bug fixing commits per each project.
In order to tell apart a bug fixing commit from a non bug fixing commit we filter them by applying a regular expression to the commit message: **fix this text**

```
$[Ff]ix(ed|es|ing)?(\s)*([Bb]ug|[Ii]ssue|[Pp]r
```

The intuition behind this regular expression is to filter commits where their commit message includes phrases such as: "Fixed issue with variable x", "Fix for problem discussed in meeting", "Error fixed", "Fixing file bug", etc.
We also added two extra filters to our search: First we had commits that only modify java source code. This is with the intention of looking only into human fixing commits that fix java source code. Developers may also fix bugs in other sections of their projects, for example they may fix bugs in bash files, or xml code or any other kind of bug, and this wouldn't be relevant to our purpose of building a model that looks into the ways in which human developers fix their java code.
The last filter we applied to our search is to look only for the fixing commits that modify a maximum of 3 files per commit. This is for two reasons: first, we want to filter out big merges of code such as pull requests or initial commits where the commit is doing much more than just fixing a bug, which are the cases that we are interested in; and second, because these approaches usually work better when the fault is localized in a small number of locations.
For each of these bug fixing commits, we then checked the version of the code before the fix was performed and after the fix was performed. We refer to these versions as the "before-fix" version and the "after-fix" version.
We then need to know what changes are happening from the before-fix version to the after-fix version, and we need to be able to analyze how often these changes performed match the mutation operators we are analyzing in order to see what are the probabilities of each of them to happen.

In order to do this we used two different widely used tools for mapping changes between two versions of code: Gumtree and QACrashFix./todorefereces to these two papers
These tools create an AST representation of the program files for both the before-fix and after-fix representations of the programs for each of the files of the commit, and then use a set of heuristics to match the before-fix version with the after-fix version. Once this is done, the program's output is a set of changes needed to be performed to get from version before-fix to version after-fix.
We created two levels in the probabilistic model, the first one we will call the *Mutation operator probabilistic model*, and the second level we will call the *Replacements probabilistic model*.
The *Mutation operator probabilistic model* is a probabilistic model that describes the probabilities to choose between the several different mutation operators in a particular fault location.
The *Replacements probabilistic model* is the next level of the probabilistic model, which describes, if the "Replacement" mutation operator is picked, what are the probabilities of replacing one statement for another. We will explain this in detail below.

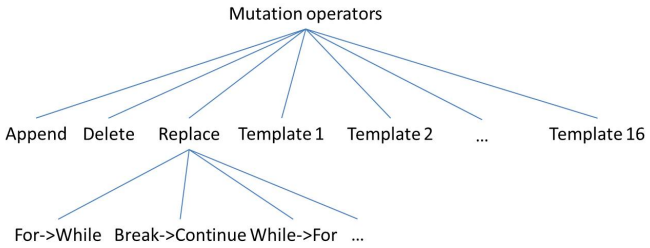

Fig. 2. 2 level probabilistic model

A. Mutation operator probabilistic model

In order to build the *Mutation operator probabilistic model* we created a program (available at **link to the github repo**) that goes through these changes and matches these changes with the mutation operators we described earlier.
We sum the instances of each of the appearances of the mutation operators in the lists of changes created out of the differences between the before-fix and after-fix versions of each of the files in the 100 last bug fixing commits per each of the 200 projects; and these instance sums are the data we take to create our probabilistic model.

B. Replacements probabilistic model

In order to build the *Replacements probabilistic model* we created a program similar to the one mentioned before (available at **link to the github repo**) that goes through these changes and when it encounters a replacement mutation operator, then it looks at what kind of statement is being replaced and what kind of statement is replacing the other. We call these statements "replacee" and "replacer" accordingly.
We analyzed 22 different kinds of statement and the probability in which each of them replaces another. For example, what is the probability that a For loop would replace a While loop, or the probability that a Break statement would replace a Continue statement, etc.
Since there are 22 statement types that we analyzed and each of these 22 may replace each of these 22 statement types, we then have 484 combinations of replacement combinations.
It is also worth noticing that these probabilities are not reciprocal, meaning that the probability of a For loop replacing a While loop is different from the probability of a While loop replacing a For loop, and the same applies to all the different statement types.

IV. Sanity Check

Section text here.

A. 10 fold cross validation

Subsection text here.

B. Small Example

Subsection text here.

V. Evaluation

Section text here.

VI. Upgrading the model

A. Single line bugs

Subsection text here.

B. Multi line bugs

Subsection text here.

VII. Discussion

Section text here.

VIII. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

Fix bibliography

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. *ICSM*, pages 230–239, 2013.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The plastic surgery hypothesis. In *FSE*, pages 306–317, 2014.

[3] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with smt. In *CSTVA*, 2014.

[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 *ICSE*, pages 422–431, 2013.

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