2017 IEEE/ACM 39th IEEE International Conference on Software Engineering Companion

Poster: An Empirical Study on Using Hints from Past Fixes

Hao Zhong
Department of Computer Science and Engineering
Shanghai Jiao Tong University, China
Email: zhonghao@sjtu.edu.cn

Na Meng
Department of Computer Science
Virginia Tech, USA
Email: nm8247@cs.vt.edu

***Abstract*—With the usage of version control systems, many bug fixes have accumulated over the years. Researchers have proposed various approaches that reuse past fixes to fix new bugs. However, some fundamental questions, such as how new bug fixes can be constructed from old fixes, have not been investigated. When an approach reuses past fixes to fix a new bug, the new bug fix should overlap with past fixes in terms of code structures and/or code names. Based on this intuition, we systematically design six overlap metrics, and conduct an empirical study on 5,735 bug fixes to investigate the usefulness of past fixes. For each bug fix, we create delta dependency graphs, and identify how bug fixes overlap with each other by detecting isomorphic subgraphs. Our results show Besides that above two major findings, we have additional ten findings, which can deepen the understanding on automatic program repair.**

I. Introduction

With the usage of version control systems, many bug fixes have accumulated over the years. Researchers have conducted various empirical studies to understand bug fixes. For example, Nguyen *et al.* find that bugs can be repetitive [7], indicating that it is feasible to fix new bugs using past fixes. Based on the observation, researchers have proposed approaches that reuse past fixes to repair new bugs. For example, Kim *et al.* extract fix patterns from thousands of bug fixes [2]. Martinez and Monperrus mine repair models from past bug fixes to guide the repair process [5]. Meng *et al.* infer program transformation from change examples, and then leverage the transformation to apply similar edits [6]. Long *et al.* train a model to locate and reuse related past bug fixes [4]. Although their approaches show promising results, we argue that neither existing empirical studies nore repair approaches are complete:

1. Existing empirical studies are based on simple or even manual analyses. In their empirical studies, researchers typically check out only buggy files and fixed files [10]. As such files are partial code, most empirical studies (*e.g.*, [5], [9]) were conducted with the support of PPA [1], the state-of-the-art analysis tool for partial code. Due to the challenges of analyzing partial code, PPA can build only abstract syntax trees for partial code, and is insufficient to support many complicated analyses. As a result, most studies (*e.g.*, [9]) do not fully explore their research questions, since PPA cannot support their desirable analyses. To conduct such analyses, some studies (*e.g.*, [2], [8]) are even based on manual analysis, which is error-proning and does not scale.

2. Some hypotheses are not fully explored. We notice that some hypotheses are not fully explored, before corresponding approaches are proposed. For example, Long *et al.* construct new fixes based on known past fixes [4]. Although their evaluation shows positive evidences that some bugs can thus be fixed, their underlying hypothesis is not fully explored by any empirical studies.

II. Methodology

Our study is based on the hypothesis: whether a fix can be constructed from past fixes depends on their similarity or overlap in terms of code names and/or structures. Previous studies (*e.g.*, [9]) collected many bug fixes, so it is feasible to ensure the representativeness of our study.

For each pair of modified source files, we build two system dependency graphs (g_l and g_r), and a delta graph:

Definition 1: A delta graph is defined as a triple, $\delta = \langle SG_l, SG_r, L\rangle$, where SG_l is a set of subgraphs of g_l; SG_r is a set of subgraphs of g_r; and $L \subseteq G_l \times G_r$ is a set of edges. A $\langle sg_{l1}, sg_{r1}\rangle$ edge denotes that sg_{l1} is modified to sg_{r1}.

For a set of past fixes $F = \{f_1, \ldots, f_n)\}$, a new bug fix f_b, and their delta graphs ($\Delta = \{\delta_1, \ldots, \delta_n\}$, and δ_b), we define the following overlap metrics:

1. Fully overlapped bug fixes (FI): A previous fix δ_i covers both the structure and name changes of δ_b, *i.e.*, $\delta_b \subseteq \delta_i$.

2. Partially overlapped bug fixes (PI): No previous fix can cover both the structure and name changes of δ_b, but the composition of some fixes cover both types of changes, *i.e.*, $\delta_b \subseteq \delta_m \cup \ldots \cup \delta_n$.

To understand the importance of structure changes, we define a function that transfers a delta graph to an abstract graph by replacing concrete methods and fields with standard representations. The transfer function $\mu(v)$ is shown below:

$$\mu(v) = \begin{cases} \text{invoke method}, & v \text{ invokes a method.} \\ \text{get field}, & v \text{ gets a field.} \\ \text{put field}, & v \text{ puts a field.} \\ v, & \text{otherwise.} \end{cases} \quad (1)$$

We present the resulting abstract graphs as $A = \{\alpha_1, \ldots, \alpha_n\}$, and α_b. We define the following two overlap metrics relevant to structure changes:

978-1-5386-1589-8/17 $31.00 © 2017 IEEE
DOI 10.1109/ICSE-C.2017.88

TABLE I: Overall result of learning from the same project

Project	Both				Structure				Code Name				Fix
	FI	%	PI	%	FS	%	PS	%	FN	%	PN	%	
aries	8	1.8%	10	2.3%	38	8.6%	144	32.6%	16	3.6%	37	8.4%	442
cassandra	68	2.8%	115	4.7%	383	15.6%	1,202	48.8%	126	5.1%	327	13.3%	2,463
derby	37	1.5%	44	1.8%	249	10.4%	865	36.2%	63	2.6%	169	7.1%	2,392
mahout	9	2.1%	14	3.2%	47	10.7%	155	35.4%	12	2.7%	29	6.6%	438
Total	122	2.1%	183	3.2%	717	12.5%	2,366	41.3%	217	3.8%	562	9.8%	5,735

3. Fully overlapped structure changes (FS): The structure changes of a previous fix α_i cover the structure changes of α_b, *i.e.*, $\alpha_b \subseteq \alpha_i$.

4. Partially overlapped structure changes (PS): α_b is composed of known structure changes, *i.e.*, $\alpha_b \subseteq \alpha_m \cup \ldots \cup \alpha_n$.

Bug fixes can involve code name changes. We define a function $\theta(\delta)$ to collect code name changes:

$$\theta(\delta) = \{(name_o, name_n)\} \quad (2)$$

where $name_o$ denotes an original code name, and $name_n$ denotes its modified new code name. For F and f_b, the extracted names changes are represented as $B = \{\beta_1, \ldots, \beta_n\}$ and β_b. The following two overlap metrics are defined relevant to name changes:

5. Fully overlapped name changes (FN): The name changes of a previous fix β_i cover the name changes of β_b, *i.e.*, $\beta_b \subseteq \beta_i$.

6. Partially overlapped name changes (PN): β_b is composable of known name changes, *i.e.*, $\beta_b \subseteq \beta_1 \cup \ldots \cup \beta_n$.

We implemented a tool, called GRAPA [10], that extracts system dependency graphs and delta graphs for bug fixes. For each pair of modified methods (m_l and m_r) in a bug fix, GRAPA builds two system dependency graphs, g_l and g_r. GRAPA compares g_l and g_r with the Hungarian algorithm [3]. Our evaluation results [10] show that GRAPA correctly builds delta graphs for more than 90% of fixes. More details on the tool are available at

http://cs.sjtu.edu.cn/~zhonghao/tr/grapa.pdf

III. Early Result and Future Work

Table I shows the overall result. Column "Project" lists names of projects. Columns "Both", "Structure", and "Code Name" list matched bugs with corresponding overlap metrics. Column "Fix" lists number of collected fixes. In total, Column "Both" shows that only several percents of bugs can be fixed, if an approach requires both structure changes and code name mappings. Column "Code Name" shows slightly better results, but Column "Structure" shows that more bugs can be fixed, if a repair approach requires only structure changes. Based on our results, it is likely to learn code structures from past fixes, but it is less likely to learn code mappings. As a result, only about 3% of bug fixes can be constructed from past fixes. In our future work, we plan to explore more open questions:

OP1: How creative is a bug fix?

Although many researchers admit the complexity of fixing bugs, some recent studies (e.g., [7], [9]) present contradicted evidences. This research question mainly concerns the explanation for the contradicted evidences. For each bug, we plan to investigate how many of its nodes and methods can be covered by past fixes. If a change never appears in past fixes, it shall be more difficult to be fixed and needs more creative activities.

OP2. What are the challenges when preparing the repository of past bug fixes?

For a bug under fixing, it needs to locate its related past fixes, before we can learn useful knowledge. This research question concerns how difficult it is to retrieve useful past fixes for a bug, which is reflected by the ratio from the useful past fixes to the total past fixes.

OP3. What is the potential to learn from other projects?

A project can have only limited past bug fixes, especially when the project is new. A natural way to handle this problem is to learn from other projects, but its effectiveness is largely unknown. To investigate this research question, we plan to explore to what degree can a new bug fix be constructed from past bug fixes from other projects.

IV. Acknowledgement

Hao Zhong is sponsored by the National High Technology Research and Development Program of China (863) No.2015AA015302, the National Nature Science Foundation of China No. 61572313, and the grant of Science and Technology Commission of Shanghai Municipality No. 15DZ1100305. Na Meng is sponsored by the NSF CCF No. 1565827.

References

[1] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. 23rd OOPSLA*, pages 313–328, 2008.

[2] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. 35th ICSE*, pages 802–811, 2013.

[3] H. W. Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[4] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *Proc. 43rd POPL*, pages 298–312, 2016.

[5] 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, 2013.

[6] N. Meng, M. Kim, and K. S. McKinley. LASE: locating and applying systematic edits by learning from examples. In *Proc. 35th ICSE*, pages 502–511, 2013.

[7] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen. Recurring bug fixes in object-oriented programs. In *Proc. 32nd ICSE*, pages 315–324, 2010.

[8] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai. Bug characteristics in open source software. *Empirical Software Engineering*, 19(6):1665–1705, 2014.

[9] H. Zhong and Z. Su. An empirical study on real bug fixes. In *Proc. 37th ICSE*, pages 913–923, 2015.

[10] H. Zhong and X. Wang. Boosting complete-code tools for partial code analysis. In *submitted*, 2017.