Toward Improving Graftability on Automated Program Repair

Soichi Sumi, Yoshiki Higo, Keisuke Hotta, and Shinji Kusumoto
Graduate School of Information Science and Technology, Osaka University, Japan
Email: {s-sumi,higo,k-hotta,kusumoto}@ist.osaka-u.ac.jp

***Abstract*—In software evolution, many bugs occur and developers spend a long time to fix them. Program debugging is a costly and difficult task. Automated program repair is a promising way to reduce costs on program debugging dramatically. Several repair techniques reusing existing code lines have been proposed in the past. They reuse code lines already existing in the source code to generate variant source code of a given source code (if an inserted code line to fix a given bug is identical to any of the code lines in existing source code, we call the code line *graftable*). However, there are many bugs that such techniques cannot automatically repair. One of the reasons is that many bugs require code lines not existing in the source code of the software. In order to mitigate this issue, we are conducting our research with two ideas. The first idea is using a large dataset of source code to reuse code lines. The second idea is reusing only structures of code lines. Vocabularies are obtained from faulty code regions. In this paper, we report the feasibilities of the two ideas. More concretely, we found that the first and second ideas improved *graftability* of code lines to 43–59% and 56–64% from 34–54%, respectively. If we combine both the ideas, *graftability* was improved to 64–69%. In cases where we used the second idea, 24–49% variables used in reused code lines were able to be retrieved from the surrounding code of given faulty code regions.**

***Index Terms*—Automated program repair, Source code analysis, Code reuse**

I. Introduction

Program debugging lies in the main activities of software evolution. Developers generally use a suite of test cases to check whether the software system works as required. Program debugging is a difficult and time-consuming task. Some researchers have reported that debugging consumes more than half of all the costs in software development [1].

Automating program debugging is a promising way to reduce its cost. Debugging consists of two phases: the first phase is locating the cause of a given bug; the second phase is changing code to remove the bug. Many research studies have been dedicated to automating fault localization [6], [16]. On the other hand, automating code changes to remove bugs (so-called *automated program repair*) has been attracting much attention recently. Automated program repair techniques generate a variant program that passes all the given test cases. Generating a variant program is a sequence of operations such as inserting, deleting, and replacing a code line on a given program. If a variant program passes all the given test cases, it is regarded as a repaired version of the given program. If the variant does not, another variant program is generated.

A variety of techniques have been proposed to realize automated program repair, and ones using genetic programming have shown their capabilities [9], [13]. Such techniques generate variant programs with insertion, deletion, and replacement operations. In cases of deletion, a faulty code line identified by a fault localization technique is deleted, which is not a bothering operation. On the other hand, in cases of insertion and replacement, a code line to be added is randomly selected from the source code of the given program. There are two issues on randomly selecting code lines from existing source code. The first issue is taking a long time to generate a repaired version of a given program. If a program includes a huge number of code lines in its source code, it is difficult to realize efficiently generating a repaired version with random selections. The second issue is low *graftability*. Herein, **graftable** means bugs can be removed by reusing code lines existing in the source code. Le Goues et al. reported that they were able to fix 52% (=55/105) of given bugs with their tool GenProg [9]. Barr et al. also reported 42% of commits can be organized with code lines included in existing source code [2]. In other words, almost half of bugs cannot be fixed by reusing existing code lines as they are in cases where fixing these bugs requires some code lines to be inserted or replaced.

In this research, we are tackling the second issue. The aim of this research is constructing a technique to remove more bugs. The key ideas of our technique are as follows.

- The first idea is using a large set of source code to select code lines for insertion and replacement operations.
- The second idea is reusing only the structure of code lines. Vocabularies are retrieved from surrounding code of a given faulty code line.

Existing techniques use code lines in the same software as they are, which should be a reason that the techniques have low ratios of *graftable* bugs. In this research, we enlarge targets of code line selection and reuse only the structure of code lines.

We have conducted an experiment to evaluate the two ideas to investigate whether every code line in bug-fix commits was *graftable* or not. In other words, we checked whether every code line in bug-fix commits is obtainable from existing source code or not. Its results include the followings.

- In cases where we use the first idea, the ratio of *graftable* code lines was increased from 37–54% to 43–59%.
- If we use the second idea, the ratio of *graftable* code lines was increased to 56–64%.
- If we use the both ideas, *graftability* of code lines was dynamically increased to 64–69%.
- In case where the second idea is adopted, we need to collect variable name information in one way or another. In the experiment, we confirmed that 24–49% variable names were able to be retrieved from back-and-forth 5 lines of faulty code lines and 36–78% variable names were obtainable if we consider whole the same file.

II. Related Work

Some techniques have been proposed for automated program repair. One of the most cynosure techniques is using genetic programming to generate a repaired version of a given program [9], [13]. GenProg takes a faulty program and a suite of test cases as its inputs and generates a repaired version of the program with genetic programming [9]. In GenProg, variant programs are generated with three operations, *mutation*, *crossover*, and *selection*.

Mutation: this operation makes a variant program from a given program by applying a small change, which is inserting, deleting, or replacing a code line.

Crossover: this operation makes two variant programs from given two programs by randomly exchanging pieces of the representations of the two parents.

Selection: this operation selects some variant programs made with *mutation* and *crossover* operations. The number of passed test cases is a standard to select variant programs. Selected variant programs are leveraged to make next-generation variant programs.

All *n*-th generation variant programs made by *mutation* and *crossover* are tested with given test cases. If a variant program passes all the test cases, it is regarded as a repaired version of a given faulty program. If none of variant programs pass all the test cases, some of the variant programs are selected as targets of *mutation* and *crossover* operations to make *(n+1)*-th generation variant programs.

GenProg runs all the test cases to test every variant program, so that runtime of test cases occupies a large portion of GenProg's total runtime. Unlike GenProg, RSRepair generates only a single variant program and running test case for it [13]. If a variant program does not pass any of the test cases, RSRepair does not run remaining test cases. This strategy has both advantages and disadvantages. An advantage is that RSRepair's runtime is shorter than GenProg in cases where the both tools can generate repaired programs. A disadvantage is that RSRepair can generate repaired programs only if a given faulty program contains only a single bug. On the other hand, GenProg is able to generate repaired programs even if there are multiple bugs in a given program.

Both GenProg and RSRepair perform *mutation* operations to generate variant programs. The *mutation* operation includes the following three manipulations and one of them is randomly applied to a given program for each time.

Deletion: this manipulation deletes a faulty code line of a given program.

Insertion: this manipulation selects a code line from a given program and inserts it to the next line of a faulty code line of the program.

Replacement: this manipulation is a combination of *deletion* and *insertion*. A faulty code line of a given program is deleted and then a code line selected from the given program is inserted to the deleted position.

In *insertion* and *replacement* manipulations, code lines already existing in a given program are reused to generate variant programs. Barr et al. conducted an experiment to investigate the capability of such reuse-based approaches on 12 open source software [2]. Their investigation results showed that 42% of commits were organized with existing code lines.

SemFix is also a repair tool and it is based on programming semantic theory [11]. The tool generates a repaired version of a given program by using logical expressions of properties that the given program must satisfy. SemFix is able to generate repaired versions for given faulty programs where GenProg and RSRepair are not able to. SemFix solves logical expressions with SMT solver [3]. SMT problems are NP-complete, so that it is difficult to solve complex logical expressions. DirectFix is an improvement of SemFix [15]. DirectFix has been designed to generate simpler patches for given bugs.

PAR is a tool for automated program repair and it uses human-written patches to remove given bugs [7]. Literature [7] reported that PAR was able to generate repaired versions for given faulty programs whether GenProg was not. However, Monperrus critically discusses PAR's experimental design [10].

III. Research Motivation

Le Goues et al. reported that GenProg was able to fix 55 out of 105 bugs [9]. One reason why there are bugs that GenProg cannot fix is that bugs occasionally require code lines that do not exist in the source code of the software to fix them. Barr et al. investigated how *graftable* commits are, i.e., how much commits can be reconstituted from existing code. As a result, they found that 42% of commits are *graftable*.

The authors consider that if commits are more *graftable*, more bugs will be able to be fixed with automated program repair techniques. One way to increase *graftability* is using a large set of source code to reuse code lines. A large set of source code, which consists of many software projects, should include more various code lines than its own software. Thus, *graftabiliy* should get increased by using a large dataset. Consequently, we investigate the following research question.

RQ1: how much does *graftability* of code lines get increased by using a large set of source code?

However, a concern is that variables in code lines of a large dataset should be an obstacle to reuse its code lines. Generally, in automated program repair, code lines are reused as they are. Consequently, code lines must be fit to the context of a given faulty code region from the viewpoint of structure and vocabulary. A large corpus of code provides a larger variety of structure of code lines compared to using a single project, which should improve *graftability*. On the other hand, it also provides a larger variety of vocabulary, which might adversely affect *graftability*. In other words, even if code lines structurally fit faulty code regions, it depends on vocabulary used in the code lines whether they are reusable or not.

We conducted a small experiment to reveal how much variables used in a software project are used in different projects. In this experiment, we leveraged two sets of software projects, which were used in our previous research [5]. The first dataset consists of 83 software projects that have been developed in *Apache Software Foundation*. The second one

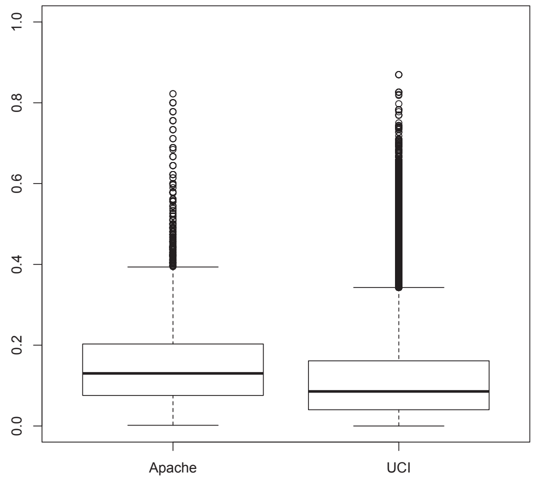


Fig. 1. Distributions of $v(p_1, p_2)$ for the two datasets

consists of 500 software projects extracted from *UCI source code dataset* [12]. All the projects included in both the datasets are written in Java language. Please refer to literature [5] to see how those datasets were constructed because there is no space in this paper to describe that.

To illustrate the issue of vocabulary, we calculated the following value on two-project permutation in each dataset.

$$v(p_1, p_2) = \frac{|V(p_1) \bigcap V(p_2)|}{|V(p_1)|} \quad (1)$$

where

- p_1, p_2: projects included in a given dataset. p_1 must be a different project from p_2.
- $V(p)$: a set of variable names appearing in project p. Only variable names are included in this set. Method names, class names and other identifiers are not included.
- $|V|$: the number of elements in a given set V.

$v(p_1, p_2)$ means how much p_1's variable names are also used in a different project p_2.

Figure 1 shows distributions of $v(p_1, p_2)$ for the two datasets. We can see that most of the values are low. The upper quartiles for *Apache* and *UCI* datasets are 20.2% and 16.1%, respectively. Those results show that most of variables used in a software project are not used in different projects. Thus, vocabulary differences can be an obstacle that code lines in a software project is reused in another project. Nevertheless, it should be possible that more code lines are able to be reused if variables are normalized. Consequently, we investigate the following research question.

RQ2: how much does *graftability* of code lines get increased by normalizing variable names?

If we normalize variables in code lines, we need to retrieve variable names from other information. A simple way to obtain variable names is checking surrounding code of the given faulty code line. Consequently, we also investigate the following research question.

RQ3: how much do variable names in inserted code lines appear in the surrounding code of faulty code lines?

IV. Experiment

We conducted an experiment to answer the three RQs.

A. Data Collection

We used five software projects, which are shown in Table I. For those projects, we investigated *graftability* of code lines included in bug-fix commits. The bug-fix commits were identified by using JIRA. More concretely, we firstly collected JIRA's issues satisfying all the following conditions:

- their status is *Closed*,
- their issue type is *Bug*, and
- their resolution is *Fixed*.

Then we identified commits including the commit IDs of the collected commits. Besides, in Hadoop-common, we removed a commit 9a9fcf8 from our targets because the commit had added 249K lines of code.

As a large set of source code, we used *UCI source code dataset* [12]. It includes more than 13,000 Java projects or more than 300 million lines of code. Its size is much larger than the source code of the target software projects.

B. Data Analysis

This experiment includes three sub-experiments.

EXP1: investigating whether each code line added by the target commits is *graftable* or not. The *graftability* of code lines was checked with the following two conditions.

PARENT: a given code line added by a target commit is identical to any of the code lines included in the revision just before the commit.

LARGESET: a given code line added by a target commit is identical to any of the code lines included in the revision just before the commit or the *UCI dataset*.

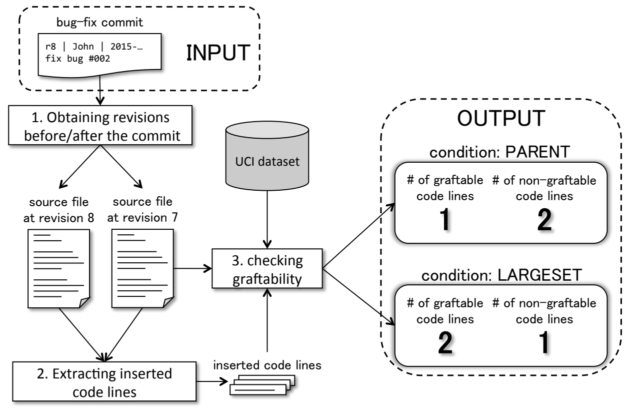
EXP2: investigating whether each code line added by the target commits is *graftable* or not in case that every variable name is normalized. The difference between EXP1 and EXP2 is that only EXP2 checked *graftability* of code lines on normalized source code.

EXP3: investigating whether each variable name included in each code line added by the target commits appears in the surrounding code region of a given faulty code line.

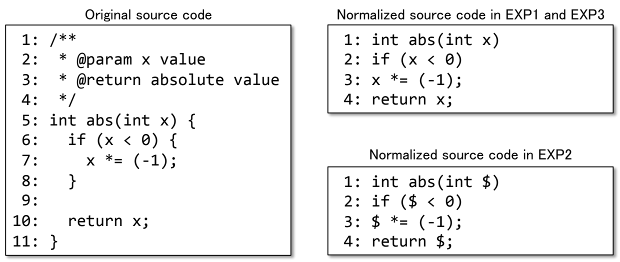
Figure 2 shows an overview of EXP1 and EXP2. The investigations consist of the following steps.

TABLE I
Target software

Software	Start commit ID	End commit ID	# bug-fix commits
Camel	babdb30 (4/Jun/2007)	390e6c2 (26/Mar/2015)	1,625
Felix	826b7fc (27/Mar/2006)	50d4715 (15/Apr/2015)	1,472
Hadoop-common	84541f6 (23/Jun/2009)	0a9f2c5 (21/Aug/2014)	922
Hive	011a680 (13/Nov/2008)	7b49691 (17/Apr/2015)	2,281
OpenJPA	eb96e89 (1/Aug/2006)	bdaf141 (16/Apr/2015)	1,348


bug−fix commit
r8 | John | 2015-…
fix bug #002
INPUT
1. Obtaining revisions before/after the commit
source file at revision 8
source file at revision 7
UCI dataset
3. checking graftability
2. Extracting inserted code lines
inserted code lines
OUTPUT
condition: PARENT
# of graftable code lines
# of non−graftable code lines
1
2
condition: LARGESET
# of graftable code lines
# of non−graftable code lines
2
1

(a) Operation flow


Original source code
Normalized source code in EXP1 and EXP3
Normalized source code in EXP2

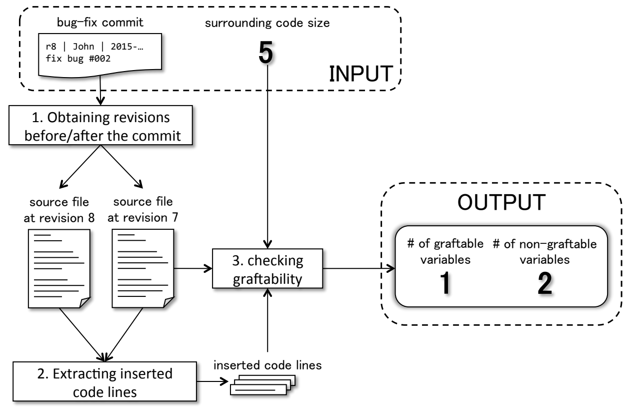
```
/**
 * @param x value
 * @return absolute value
 */
int abs(int x) {
  if (x < 0) {
    x *= (-1);
  }

  return x;
}
```

```
int abs(int x)
if (x < 0)
x *= (-1);
return x;
```

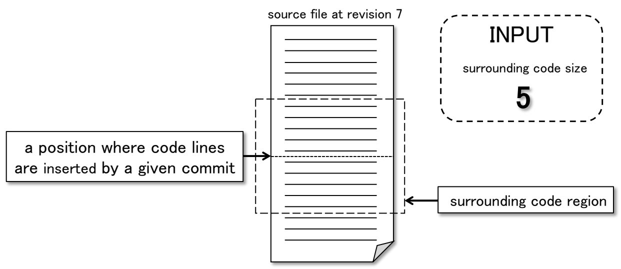
```
int abs(int $)
if ($ < 0)
$ *= (-1);
return $;
```

(b) Normalization example

Fig. 2. Overview of investigating *graftability* of code lines


bug−fix commit
r8 | John | 2015-…
fix bug #002
surrounding code size
5
INPUT
1. Obtaining revisions before/after the commit
source file at revision 8
source file at revision 7
3. checking graftability
2. Extracting inserted code lines
inserted code lines
OUTPUT
# of graftable variables
# of non−graftable variables
1
2

(a) Operation flow


source file at revision 7
INPUT
surrounding code size
5
a position where code lines are inserted by a given commit
surrounding code region

(b) Surrounding code line

Fig. 3. Overview of investigating *graftability* of variables

STEP1: obtaining revisions before/after a given commit.
STEP2: extracting inserted code lines.
STEP3: checking *graftability* for each code line.

In STEP1, by specifying the ID of a given commit, we can easily obtain its before and after revisions. All the source files of the revisions are retrieved because they are used in STEP3. Obtained source files are normalized with the following rules.

- Open and close brackets ("{" and "}") are removed.
- Comments, blank lines, and indents are removed.
- variable names are normalized (only in EXP2).

Figure 2(b) shows a normalization example. In EXP2, all the variable names were replaced with the same special token. This normalization was performed only on variable names. Other identifiers such as method names were not replaced.

STEP2 is a procedure to identify inserted code lines in the commit. The identification is operated with UNIX `diff` tool. Identified inserted code lines are regarded as code lines necessary for repaired program.

In STEP3, each of the inserted code lines is checked whether it is the same as any of the code lines in the previous revision of the source code or in *UCI dataset*.

Figure 3 shows an overview of EXP3. Its STEP1 and STEP2 are exactly the same as EXP1. In STEP3, each variable in the inserted code lines is checked whether it appears in the surrounding code region (see Figure 3(b)).

C. Result and Discussion

Table II shows the *graftability* of code lines. The numbers in the parentheses are the improvement from the original *graftability*. This table leads us to the following answers.

Our answer to the RQ1: by using *UCI dataset*, *graftability* of code lines rose to 43–59% from 37–54%.

Our answer to the RQ2: by normalizing variable names, *graftability* of code lines rose to 56–64%.

We confirmed that there are significant differences between original *graftability* and PARENT/LARGESET ones with Wilcoxon test and the siginificant levels are large with Cliff's delta where *p-value* is 0.05. In case where we combined the two strategies, *graftability* of code lines jumped to 64–69%, which were increased by 16–27% from the original *graftability*.

Table III shows the *graftability* of variables. This table lead us to the following answer.

Our answer to the RQ3: in cases of taking variables from a surrounding code region (back-and-forth five

TABLE II
Graftability OF CODE LINES

Software	w/o variable normalization		w/ variable normalization	
	PARENT	LARGESET	PARENT	LARGESET
Camel	53.7%	58.8%	63.9%	69.3%
	–	(+5.1%)	(+10.2%)	(+15.5%)
Felix	43.4%	51.0%	59.9%	68.8%
	–	(+7.6%)	(+16.5%)	(+25.4%)
Hadoop	43.7%	50.0%	59.3%	66.2%
	–	(+6.3%)	(+15.6%)	(+22.5%)
Hive	37.3%	42.9%	56.3%	64.1%
	–	(+5.6%)	(+19.0%)	(+26.8%)
OpenJPA	42.7%	49.2%	57.6%	65.0%
	–	(+6.5%)	(+14.9%)	(+22.3%)

lines), *graftability* of variables was 24–49%. In cases of whole the same file, it rose to 36–78%.

Only OpenJPA has much lower *graftability* of variables than the other projects. We investigated dozens of OpenJPA's target commits and found that many of them include tangled changes. Variables that were not able to be retrieved were inserted code lines not for bug-fix but for other purpose such as adding new functions. If we would have been able to use only the code lines that were inserted to the source code for fixing bugs, *graftability* of variables would have been much better.

D. Threats to Validity

Here, we describe threats to validity in the experiment.

Bug-fix commit: we collected commits related to bugs managed in JIRA. However, the code lines added by the commits are not always related to fix bugs. As in the case for OpenJPA, commits occasionally include tangled changes [4], [8]. If we would have been able to use code lines that contributed to fix bugs, *graftability* of code lines and variables would have been better.

Target Software: we conducted the experiment on only five projects, which are written in Java and use JIRA to manage their issues. We are going to conduct more experiments on other programming languages because different programming language have different tendencies on bug occurrences [14].

V. Conclusion

In this paper, we reported results of a preliminary experiment on our research to improve automated program repair. Our research aim is fixing more bugs by reusing code lines already existing in the source code. We call to what extent we can reuse existing code lines to fix bugs *graftability*. We have two ideas to improve *graftability*: the first one is reusing code lines from a large set of source code, not only the source code of the software; the second one is reusing only the structure of code lines, variables are retrieved from the surrounding code of the faulty code line. The results shows that:

- in cases that we use *UCI source code dataset*, which includes more than 300 million lines of code, code line *graftability* rose to 43–59% from 37–54%;
- in cases that we reuse only the structure of code lines (variable names are normalized in advance), *graftability* rose to 56-64%;
- in cases that we adopt both the two ideas, *graftability* jumped to 64–69%.

TABLE III
Variable graftability

Software	Threshold of surrounding code region			
	5	10	20	same file
Camel	49.0%	55.5%	60.6%	67.8%
Felix	46.4%	54.1%	60.5%	72.6%
Hadoop	44.0%	51.1%	58.2%	70.8%
Hive	34.5%	41.0%	47.7%	77.8%
OpenJPA	23.6%	27.4%	30.2%	35.5%

We also investigated how many variable names were able to be retrieved in cases that normalized code lines are reused:

- in cases that variable names were searched in back-and-forth five lines, 24–49% variables were obtainable;
- in cases that we search variable names in whole the same file, variable name *graftability* rose to 36–78%.

Those results show both our ideas are promising to promote reusing existing code lines for automated program repair. In the future, we are going to conduct more experiments to investigate how much the size of dataset affect *graftability* and whether using domain-specific data is advantageous or not. Besides, we are going to implement those functions on existing repair tools such as GenProg. However, we have some issues to use our ideas. The biggest issue is how we get code lines contributing to fix a given bug from a large dataset. If we randomly select code lines like existing techniques, a very long time should be required to finish fixing bugs. We need to develop a way to efficiently retrieve code lines that can fix given bugs. Currently, we are going to use code similarity to retrieve code lines from a large dataset.

Acknowledgments

This work was supported by JSPS KAKENHI Grant Numbers 25220003 and 24680002.

References

[1] J. Baker. The gpl-violations.org project. http://goo.gl/SYC1h7, Feb. 2012. Last accessed 20 April 2015.

[2] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro. The Plastic Surgery Hypothesis. In *FSE'14*, pages 306–317.

[3] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08/ETAPS'08*, pages 337–340.

[4] K. Herzig and A. Zeller. The impact of tangled code changes. In *MSR'13*, pages 121–130.

[5] Y. Higo and S. Kusumoto. How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods. In *FSE'14*, pages 294–305.

[6] J. A. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *ASE'05*, pages 273–282.

[7] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *ICSE'13*, pages 802–811.

[8] H. Kirinuki, Y. Higo, K. Hotta, and S. Kusumoto. Hey! are you committing tangled changes? In *ICPC'14*, pages 262–265.

[9] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In *ICSE'12*, pages 3–13.

[10] M. Monperrus. A Critical Review of "Automatic Patch Generation Learned from Human-written Patches": Essay on the Problem Statement and the Evaluation of Automatic Software Repair. In *ICSE'14*, pages 234–242.

[11] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. C handra. SemFix: Program Repair via Semantic Analysis. In *ICSE'13*, pages 772–781.

[12] J. Ossher, H. Sajnani, and C. Lopes. File Cloning in Open Source Java Projects: The Good, The Bad, and The Ugly. In *ICSM'11*, pages 283–292.

[13] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The Strength of Random Search on Automated Program Repair. In *ICSE'14*, pages 254–265.

[14] B. Ray, D. Posnett, V. Filkov, and P. Devanbu. A large scale study of programming languages and code quality in github. In *FSE'14*, pages 155–165.

[15] J. Y. Sergey Mechtaev and A. Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *ICSE'15*, pages 448–458.

[16] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization. In *ICSE'09*, pages 45–55.