Toward Improving Ability to Repair Bugs Automatically

–A Patch Candidate Location Mechanism Using Code Similarity–

Haruki Yokoyama, Yoshiki Higo, Keisuke Hotta, Takafumi Ohta,
Kozo Okano, and Shinji Kusumoto
Osaka University
1-5 Yamadaoka Suita Osaka, Japan
{y-haruki,higo,k-hotta,t-ohta,okano,kusumoto}@ist.osaka-u.ac.jp

ABSTRACT

Automated program repair is a promising way to reduce costs on program debugging dramatically. Some repair techniques with genetic algorithm have been proposed, and they were able to fix several dozen of actual bugs in open source software. However, existing techniques occasionally take a long time to generate a repaired version of a given program. The dominant factor for that is generating so many programs that do not pass given test cases. In this research, we are trying to generate a repaired program, which passes all the test cases, in less time. Our key idea is using code similarity to select code lines to be inserted into a given program. More concretely, we propose to select code lines in code regions similar to the faulty code regions. In existing techniques, code lines for insertion are randomly selected from a given program. Currently, we are still in an early stage of this research. In this paper, we report some promising results of our pilot study, which show that using code similarity is very effective to generate repaired programs in less time.

CCS Concepts

•**Software and its engineering → Automatic programming;** *Software notations and tools;* Software libraries and repositories;

Keywords

Automated program repair, Source code analysis, Genetic programming

1. INTRODUCTION

Program debugging is one of the main activities of software evolution. Developers generally use a suite of positive and negative test cases to check whether debugging has been completed or not. Debugging is a cost-consuming activity and there is a report that debugging consumes more than half of all the costs in software development [1].

SAC 2016,April 04-08, 2016, Pisa, Italy
©2016 ACM. ISBN 978-1-4503-3739-7/16/04…$15.00
DOI: http://dx.doi.org/10.1145/2851613.2851770

A promising way to reduce the cost of debugging is automating it. Debugging includes two tasks: one is locating the cause of a given fault; the other is changing the located code to remove the fault. Many research studies have been conducted on automating fault localization [5, 13]. On the other hand, automating code changes has been attracting much attention recently. Automating code changes to remove bugs is called as automated program repair. Automated program repair techniques generate a variant program that passes all the given test cases. Generating a variant program is an operation sequence of inserting, deleting, and replacing a code line on a given original program. If a variant program passes all the given test cases, it is regarded as a repaired version of the given program. If a variant does not, another variant program is generated until the bug is removed.

In recent years, some techniques using genetic programming have shown their capabilities for automated program repair [7, 8, 11]. Such techniques also generate variant programs with insertion, deletion, and replacement operations. In the case of deletion, a faulty code line identified by a fault localization technique is deleted, which is not time-consuming. 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. However, if a program includes a huge number of lines in its source code, it is difficult to realize generating a repaired version of programs with random selection in a short time. In other words, some strategies to select code lines for insertion is required for short-time generations of repaired programs.

In this paper, we report that using code similarity can be an effective strategy to select code lines for insertion. More concretely, after locating faulty code region, selecting code lines from code regions similar to the faulty code region is a way to generate a repaired program in less time than random selection.

2. RELATED WORK

There are some techniques for automatic program repair. The first technique is using genetic programming to generate repaired programs [7, 8, 11]. GenProg takes a suite of test cases as its inputs and generates a repaired version of the program with genetic programming [7, 8]. In GenProg, variant programs are generated by using three operations, *mutation*, *crossover*, and *selection* , which is shown in Figure 1.

Mutation: an operation making a variant from a given program

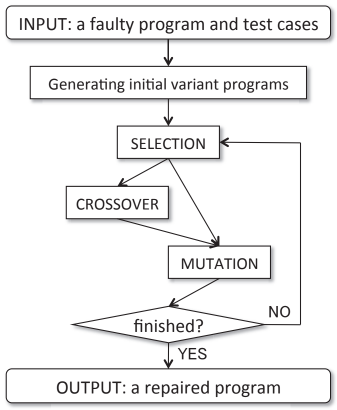
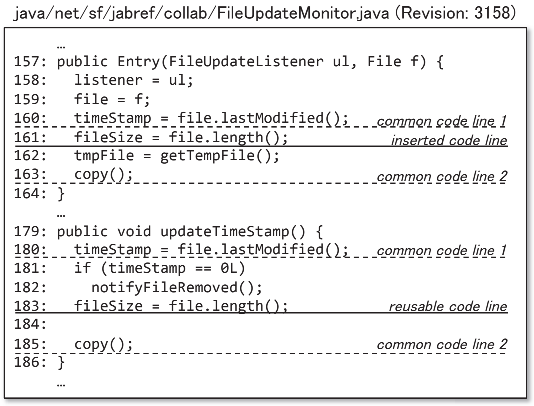

INPUT: a faulty program and test cases
Generating initial variant programs
SELECTION
CROSSOVER
MUTATION
finished?
NO
YES
OUTPUT: a repaired program

Figure 1: Overview of GenProg



java/net/sf/jabref/collab/FileUpdateMonitor.java (Revision: 3158)

```
     …
157: public Entry(FileUpdateListener ul, File f) {
158:   listener = ul;
159:   file = f;
160:   timeStamp = file.lastModified();     common code line 1
161:   fileSize = file.length();            inserted code line
162:   tmpFile = getTempFile();
163:   copy();                              common code line 2
164: }
     …
179: public void updateTimeStamp() {
180:   timeStamp = file.lastModified();     common code line 1
181:   if (timeStamp == 0L)
182:     notifyFileRemoved();
183:   fileSize = file.length();            reusable code line
184:
185:   copy();                              common code line 2
186: }
     …
```

Figure 2: A motivating example

by applying a small change. A small change is inserting, deleting, or replacing a code line.

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

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

All *n*-th generation variant programs made by *mutation* and *crossover* operations are tested with the given suite of test cases. If a variant program passed all the test cases, it is regarded as a repaired program. If there is no variant program that passes all the tests, a *select* operation is applied to the *n*-th generation variant programs. Selected variant programs are targets of *mutation* and *crossover* operations to make $(n+1)$-th generation variant programs.

GenProg runs all the test cases for all the variant programs, so that running test cases accounts for the dominant cost of GenProg's runtime. As a countermeasure for this, RSRepair has been proposed [11]. RSRepair generates only a single variant program and running test cases for it. Unlike GenProg, if a variant program does not pass any of the test cases, RSRepair does not run remaining test cases. This strategy has advantages and disadvantages on generating repaired programs. One of the advantages is that RSRepair's runtime is shorter than GenProg in cases where the both tools can generate repaired programs. One of the disadvantages is that RSRepair can generate repaired programs only if a given faulty program contains only a single bug. On the other hand, GenProg has a capability to generate repaired programs even if multiple bugs exist in a given faulty program.

Mutation operation is common to GenProg and RSRepair. In the operation, one of the following manupilations is randomly applied.

Deletion: a code line in the faulty code region of a given program is deleted.

Insertion: a code line in the given program is randomly selected, and it is inserted to the faulty region of the program.

Replacement: a combination of *deletion* and *insertion*. More concretely, a code line in the faulty region of a given program is deleted and then a code line in the given program is selected and inserted to the deleted position.

In *insertion* and *replacement* operations, code lines that already exist 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]. They revealed that 43% of inserted code lines in the development histories of the 12 software had existed in their source code.

SemFix is a technique based on programming semantic theory [10]. It generates repaired programs by using logical expressions of properties that a given program must satisfy. SemFix is able to generate repaired programs for faulty programs where GenProg is not able to generate repaired programs. SemFix solves logical expressions with SMT solver [3]. SMT problems are NP-complete, so that it is difficult to solve complex logical expressions.

PAR is a technique using human-written patches. It extracts code patterns in patches and uses the patterns to generate a repaired version of a given faulty program [6]. Literature [6] reported PAR was able to generate repaired versions for given faulty programs where GenProg was not. However, there are critical discussions on PAR's experimental design [9].

3. RESEARCH MOTIVATION

We manually investigated many bug-fix commits in open source software. We browsed inserted code lines into repaired versions and the code regions including code lines identical to the inserted ones. As a result, we found that such code regions are similar to the code region surrounding the faulty code in many cases. Figure 2 shows an actual example. This is a part of a source file in JabRef project at revision 3158. The 161st code line has been inserted in the changes between revisions 3157 and 3158. We found that there was a code line identical to the inserted code line in the same source file. Besides, there are two common code lines in the two code regions: the first one is a code region surrounding the inserted code line; the second one is a code region surrounding the code line identical to the inserted one.

This fact lets us come up with an idea that using code similarity might be a good strategy to select code lines for automatic program repair. If so, we can shorten the time to generate repaired programs when using automatic program repair techniques such as GenProg [7, 8] or RSRepair [11].

In order to evaluate the strategy to use code similarity, we investigate the following research questions.

RQ1: are code lines necessary for a repaired program located in code regions similar to the faulty code region?

RQ2: is it possible to calculate quickly similarities between the faulty code region and code regions in the given program?

4. EXPERIMENT

We conducted an experiment to answer the two RQs described in Section 3.

4.1 Data Collection

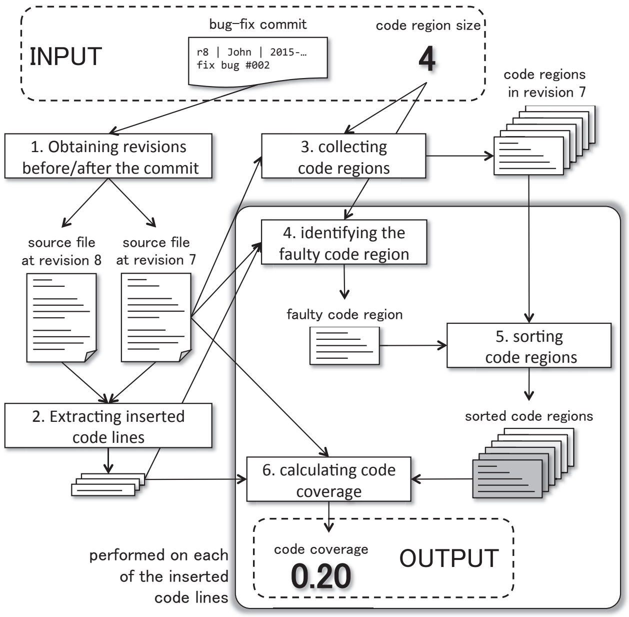
In this experiment, we used bug-fix commits, which are shown in Table 1, in four open source software. Two of them (Apache HTTP Server and CBMC) are written in C/C++ language and the other two (JabRef and jEdit) are written in Java language.

The bug-fix commits were collected with two-phase operations. In the first phase, we picked up several dozen of commits satisfying both the two conditions in an automated way.

- The commit message of a given commit includes at least one of the following words: *fix*, *fixed*, and *fixing*.
- At least one of the inserted code lines in a given commit is *graftable*. Here, a code line inserted by a given commit is *graftable* if there exists a code line in the source code before committed that is identical to the inserted one.

Table 1: Commits used in the experiment

Software	Commit			New	Graftable
	Revision	LOC	Date	lines	lines
HTTP	r1240181	129,965	3/Feb/2012	3	2
	r1377647	131,536	27/Aug/2012	13	9
	r1421780	136,938	12/Dec/2012	8	3
	r1422373	136,982	16/Dec/2012	13	7
	r1526473	142,042	26/Sep/2013	1	1
CBMC	r4950	125,598	31/Dec/2014	3	1
	r4952	125,649	31/Dec/2014	4	1
	r4979	125,673	1/Jan/2015	3	2
	r5028	126,398	8/Jan/2015	2	2
	r5167	128,734	8/Feb/2015	1	1
JabRef	r3155	61,238	2/Feb/2010	9	4
	r3158	61,250	4/Feb/2010	1	1
	r3331	62,522	15/Sep/2010	1	1
	r3354	62,606	4/Nov/2010	6	4
	r3486	63,321	25/Mar/2011	11	5
	r3528	64,429	11/May/2011	3	1
	r3599	65,187	12/Aug/2011	2	1
	r3686	69,657	10/Oct/2011	3	2
jEdit	r21720	77,198	27/May/2012	3	2
	r21768	77,101	6/Jun/2012	2	2
	r21800	77,127	14/Jun/2012	5	2
	r21918	77,190	8/Jul/2012	1	1
	r21933	77,203	13/Jul/2012	8	8
	r22000	77,318	14/Aug/2012	9	4
Total	–	–	–	115	67



INPUT
bug-fix commit
r8 | John | 2015-…
fix bug #002
code region size
4
code regions in revision 7
1. Obtaining revisions before/after the commit
3. collecting code regions
source file at revision 8
source file at revision 7
4. identifying the faulty code region
faulty code region
5. sorting code regions
2. Extracting inserted code lines
sorted code regions
6. calculating code coverage
performed on each of the inserted code lines
code coverage
0.20
OUTPUT

Figure 3: Overview of code coverage calculation

In the second phase, we manually checked if the picked-up commits are bug-fix commits or not one by one.

With the above phases, we collected 24 bug-fix commits from the four software. Table 1 includes the revision number just after the commits, so that readers can easily use the same commits in their research.

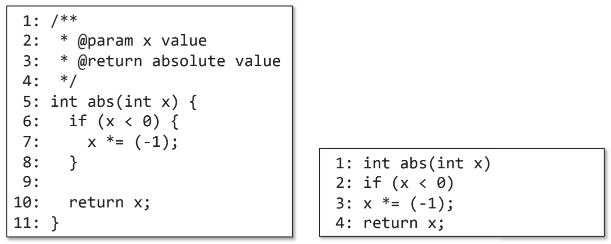
The fourth column of Table 1 entitled "New lines" means the number of inserted code lines in the commits. The last column entitled "Graftable lines" means the number of *graftable* code lines in the commits.

4.2 Data Analysis

We investigated whether each of the *graftable* lines in Table 1 was in code regions similar to the faulty code region with an automatic way, which is shown in Figure 3. We used four lines, six lines, and eight lines as sizes of code region.

The procedure of the investigation for each bug-fix commit consists of the following six steps. A bug-fix commit and a size of code region are the inputs to the procedure. The investigation method reports a value of code coverage for each of the inserted lines in the given bug-fix commit. A code coverage is a ratio of code lines covered with a given set of code regions against the whole source code. In other words, a code coverage for an inserted line means how many code lines should be examined to find the inserted line. A low value of the code coverage means that a code line in code regions similar to the faulty code region is identical to the inserted code line in a given commit. The whole procedure is executed in an automatic way.

STEP1: obtaining revisions before/after a given commit.

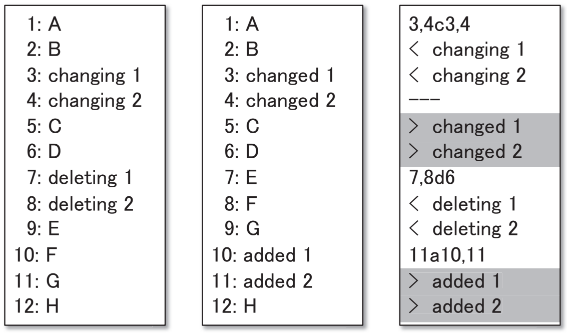


```
 1: /**
 2:  * @param x value
 3:  * @return absolute value
 4:  */
 5: int abs(int x) {
 6:   if (x < 0) {
 7:     x *= (-1);
 8:   }
 9:
10:   return x;
11: }
```

```
1: int abs(int x)
2: if (x < 0)
3: x *= (-1);
4: return x;
```

(a) Before (b) After

Figure 4: An example of the normalization in the experiment



```
 1: A
 2: B
 3: changing 1
 4: changing 2
 5: C
 6: D
 7: deleting 1
 8: deleting 2
 9: E
10: F
11: G
12: H
```

```
 1: A
 2: B
 3: changed 1
 4: changed 2
 5: C
 6: D
 7: E
 8: F
 9: G
10: added 1
11: added 2
12: H
```

```
3,4c3,4
< changing 1
< changing 2
---
> changed 1
> changed 2
7,8d6
< deleting 1
< deleting 2
11a10,11
> added 1
> added 2
```

(a) Before (b) After (c) `diff` result

Figure 5: Identifying inserted lines with `diff` tool

STEP2: extracting inserted code lines.

STEP3: collecting code regions.

STEP4: identifying the faulty code region.

STEP5: sorting the code regions.

STEP6: calculating a code coverage.

In the remainder of this subsection, we describe each of the steps in detail.

STEP1 is an easy operation. By specifying the ID (revision number) 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 STEP4. Obtained source files are normalized with the following rules.

- Open and clone brackets ("{" and "}") are removed.
- Comments, blank lines, and indents are removed.

Figure 4 shows an example of our normalization.

STEP2 is identifying 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. Figure 5 shows an example of `diff` application. The code lines starting with ">" are inserted ones.

STEP3 is collecting code regions from the whole source code of the before-commit revision. We extract all code regions with a given window size one line by one line. Figure 6 depicts how code regions are extracted from a source file.

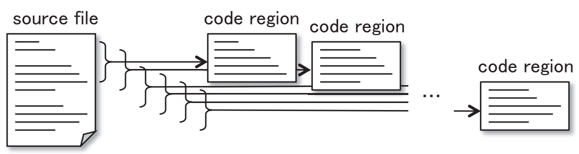


Figure 6: Collecting code regions from a source file

STEP4, STEP5, and STEP6 are performed on each of the inserted code lines identified in STEP2. STEP4 is identifying the faulty code region. The code region surrounding the inserted line is extracted as a given size. Note that the inserted code line itself is not included in the faulty code region. A faulty code region is identified to each of the inserted code lines. In the case of Figure 5, if the size of code region is 4, the faulty code region for "`changed 1` " and "`changed 2`" is "`ABCD`" and the faulty code region for "`added 1`" and "`added 2` " is "`EFGH`".

STEP5 is sorting code regions collected in STEP3 based on their similarities to the faulty code identified in STEP4. The similarity calculation between two code regions is performed with the following formula.

$$Similarity(T_a, T_b) = \frac{|LCS(T_a, T_b)| \times 2}{|T_a| + |T_b|} \quad (1)$$

T_a, T_b: token sequences in given two code regions a and b.

$LCS(T_a, T_b)$: a longest common subsequence between two token sequences T_a and T_b.

$|T|$: the number of tokens in a given token sequence T.

After STEP5, the most similar code region is located at the top of the sorted code regions. Here, we describe the sorted code regions as a list R.

$$R = \{r_1, r_2, \cdots, r_k, \cdots, r_n\} \quad (2)$$

STEP6 is calculating code coverage. We check sorted code regions from the top of the list whether a code region includes the inserted code line or not.

If code region r_k is the nearest to the top in code regions including the inserted code line, code regions in the sublist $\{r_1, r_2, \cdots, r_k\}$ are used for calculating code coverage. Code coverage is calculated by the following formula.

$$Coverage(R_k) = \frac{|\bigcup_{r \in R_k} L(r)|}{|\bigcup_{r \in R} L(r)|} \quad (3)$$

R_k: $\{r_1, r_2, \cdots, r_k\}$, which is a sublist from the top to code region r_k in R.

$L(r)$: a set of code lines in code region r.

4.3 Result and Discussion

Figure 7 shows boxplots of code coverages for the three sizes of code regions. Most of the code coverages are close to 0, which means that code regions having *graftable* lines appeared almost the top of the similar regions lists. For all the sizes, the 75 percentile is less than 0.1, which means 75% of *graftable* line is obtainable from

Table 2: Detail code coverage for rare lines

Software	Revision	Inserted code line	# of the same code lines	Code coverage		
				4 code lines	6 code lines	8 code lines
HTTP	r1422373	`int i, rv;`	8	7.76%	16.19%	36.83%
	r1377647	`pbktEOS = apr_bucket_eos_create(c->bucket_alloc);`	7	0.05%	0.03%	0.01%
	r1377647	`apr_brigade_cleanup(pbbIn);`	16	0.56%	0.26%	0.39%
	r1377647	`ctx->tmpBucket = apr_brigade_create(r->pool, c->bucket_alloc);`	16	3.00%	2.37%	1.34%
	r1377647	`apr_bucket_delete(pbktIn);`	16	7.60%	0.12%	0.03%
	r1377647	`ret = apr_bucket_read(pbktIn, &data, &len, eBlock);`	16	0.53%	0.06%	0.05%
CBMC	r5028	`if (base_type_eq(expr_type, op_type, *this))`	8	5.62%	5.10%	9.07%
	r4979	`dest += convert(it->op0());`	16	0.01%	0.09%	0.06%
JabRef	r3686	`return pageParts[0];`	4	0.01%	0.01%	0.02%
	r3599	`return input;`	20	39.66%	0.68%	0.02%
	r3528	`sb.append(lastNamePart);`	6	0.01%	0.01%	0.03%
	r3486	`pr.setInvalid(true);`	16	36.45%	16.30%	19.52%
	r3486	`pr.setFile(file);`	19	36.45%	1.92%	19.51%
	r3486	`ParserResult pr = new ParserResult(null, null, null);`	8	0.58%	1.92%	19.51%
	r3354	`if (postFormatter != null)`	16	0.28%	0.43%	0.25%
	r3354	`value = "";`	8	100.00%	96.83%	81.80%
	r3331	`fileSize = file.length();`	16	0.10%	0.03%	0.88%
	r3158	`fileSize = file.length();`	8	0.01%	0.01%	0.05%
	r3155	`List<BibtexEntry> entries;`	8	96.51%	82.81%	70.33%
	r3155	`data[0].replaceAll("~", System.getProperty("user.home")));`	13	0.01%	0.01%	0.01%
jEdit	r21933	`if (obj instanceof VFSFile)`	16	24.06%	13.07%	4.15%
	r21933	`Object obj = parentDirectories.getModel();`	16	6.03%	13.07%	4.15%
	r21933	`VFSFile dirEntry = (VFSFile) obj;`	16	16.87%	17.99%	4.15%
	r21933	`focusOnFileView();`	8	8.47%	15.73%	4.15%
	r21933	`browser.setDirectory(dirEntry.getPath());`	8	8.47%	15.73%	4.15%
	r21918	`case Frame.MAXIMIZED_BOTH:`	8	13.23%	3.97%	0.73%
	r21800	`CharsetEncoding utf8 = new CharsetEncoding("UTF-8");`	8	8.79%	7.08%	23.93%
	r21720	`VFS vfsSrc = VFSManager.getVFSForPath(path);`	8	56.92%	55.14%	54.79%

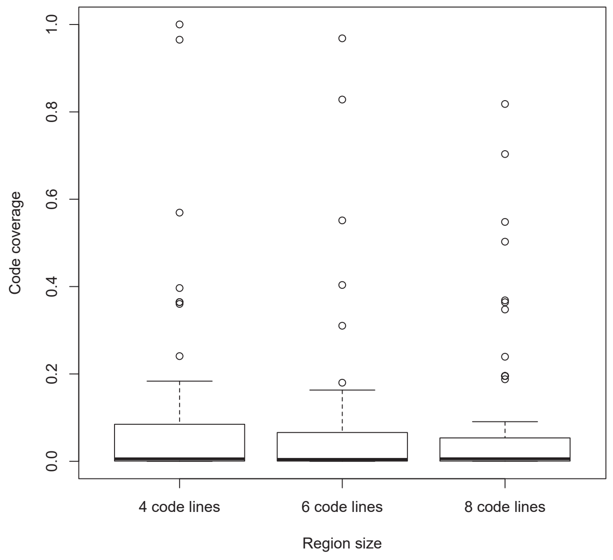


Figure 7: Code coverages on the target commits

10% lines of the source code. The 10% lines are in code regions similar to the give faulty code regions.

We also checked the number of code lines that are reusable in the faulty code lines because if so many code lines are reusable the random selection is a nice strategy, which is used in existing works [8, 11]. As a result, we revealed that only a small number of code lines are reusable in many cases. Table 2 shows detail information for the cases where 20 or fewer code lines are reusable.

For each inserted code line in the bug-fix commits, code coverage was calculated in an automatic way. We also measured its runtime. Especially, in the runtime measurement, we focused on STEP4, STEP5, and STEP6 because those steps are not included in existing techniques. In other words, if we add the proposed technique to GenProg, those steps are additional operations in its execution. Table 3 shows the runtime of the sum of those steps for all the bug-fix commits on three sizes of code regions. We can see that every

Table 3: Runtime for the Commits used in the experiment

Software	Region size	Runtime (seconds)		
		Minimum	Maximum	Average
HTTP	4 code lines	0.803	1.944	1.201
	6 code lines	1.619	3.063	2.193
	8 code lines	1.910	4.935	3.344
CBMC	4 code lines	0.681	2.174	1.566
	6 code lines	1.214	3.947	2.829
	8 code lines	1.915	6.655	4.344
JabRef	4 code lines	0.446	1.311	0.787
	6 code lines	0.719	1.718	1.292
	8 code lines	1.274	3.106	2.103
jEdit	4 code lines	0.589	1.024	0.746
	6 code lines	0.890	3.671	1.656
	8 code lines	1.623	3.176	2.395

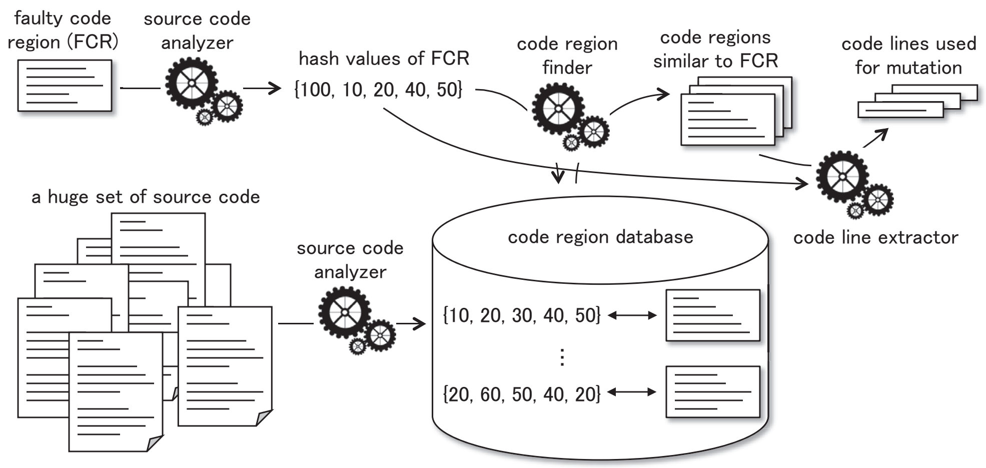


Figure 8: Concept of our future research

calculation took only a few minutes.

Our answer to the RQ1 is YES. For more than 75% of *graftable* lines, there is a reusable code line in 10% of source code, which is similar to the faulty code region.

Our answer to the RQ2 is also YES. It takes a few seconds to calculate code similarity between the faulty code region and all the code regions in the source code.

Because of the positive answers to the RQs, we can conclude that using code similarity is effective and practical to select code lines for insertion.

4.4 Threats to Validity

Here, we describe threats to validity in the experiment.

Bug-fix commit: we collected bug-fix commits by checking commits one by one. However, some commits might have been misclassified as bug-fix in the experiment because we are not the developers of the target software. In order to avoid such a mistake, a pair of the authors checked commits together.

Target software: we conducted the experiment only four software. We selected the software as our targets because of three reasons. The first reason is that they have been used in existing research related to automatic program repair. The second reason is their diversities of domains and programming languages. The third reason is that it is not difficult for the authors to understand what each commit did in their repositories because the commit messages were well written. In the future, we are going to conduct experiments on more software in the same fashion.

Size of code region: we used four, six, and eight code lines as sizes of code regions. The reason we used such small sizes as code regions is to calculate code region similarities with low calculation costs. The runtime with eight code lines was approximately three times as long as the runtime with four code lines (see Table 3). On the other hand, the code coverages with eight code lines were slightly lower than the code coverages with four code lines (see Figure 7). Thus, the time to generate repaired programs with eight code lines may be shorter than the time with four code lines. In the future, we are going to conduct more experiment with larger code regions to investigate runtime and code coverages.

5. DIRECTIONS TO FUTURE RESEARCH

In the future, we are going to enlarge the selection target for code line insertion. There is existing research reporting that 43% of inserted code lines in code changes is obtainable from its source code base [2]. If we enlarge the selection target from a single software to a huge set of source files, more code changes will be able to be supported by reusing existing code. We have conducted another experiment on 122 Java projects on Apache Software Foundation and found that about half of code changes are common to multiple projects [4].

Figure 8 shows a concept of our future research. It takes a faulty code region and outputs code lines used for mutation. However, it is not realistic to detect code regions similar to the given faulty code region quickly by analyzing a huge set of the source code itself. One promising way should be creating a code region database before detection. In the database, each code region in the huge dataset is stored as a record. A record holds a set of hash values calculated from its code region. A hash value is calculated from each code line in a code region. When a faulty code region is given, hash values are calculated from its code lines. The hash values are used as a query to obtain similar code regions from the database. By using this kind of scheme, we will be able to obtain quickly code lines used for mutation from a huge set of source code.

6. CONCLUSION

GenProg [8] and RSRepair [11] are breakthrough for automated program repair. However, they occasionally take a long time to generate repaired programs. In this paper, in order to generate a repaired version of a given program in less time, we proposed to use code lines located in code regions similar to a given faulty code region. We conducted a pilot study to evaluate its effectiveness on generations of repaired programs in less time. The followings are the main findings.

- More than 75% of *graftable* code lines are obtainable from 10% lines of the source code. The 10% lines are located in code regions similar to the faulty code region.
- Calculating similarities between the faulty code and code regions in the source code takes only a short time. This processing does not become a big overhead to generate repaired programs.

Currently, our research is still in an early stage. In the future, firstly, we embed the proposed technique to GenProg and measure its actual performance with the proposed technique. We also plan to select code lines for insertion from not only the same program but also other programs. Enlarging reuse targets realize generating repaired programs for more bugs. Using code similarity will be a key technique to select patch candidates from a large source code data.

We are also going to enhance our technique to reuse code lines with identifier normalizations. In another project, we have already confirmed the usefulness of the normalizations [12].

7. ACKNOWLEDGEMENTS

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

8. REFERENCES

[1] J. Baker. The gpl-violations.org project. http://www.jbs.cam.ac.uk/media/2012/cambridge-news-experts-battle-192-bn-loss-to-computer-bugs/, 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 *Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pages 306–317, 2014.

[3] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, 2008.

[4] Y. Higo, A. Ohtani, S. Hayashi, H. Hata, and S. Kusumoto. Toward Reusing Code Changes. In *Proceedings of the 12th Working Conference on Mining Softwre Repositories*, pages 52–56, 2015.

[5] J. A. Jones and M. J. Harrold. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 273–282, 2005.

[6] 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*, pages 802–811, 2013.

[7] 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 *Proceedings of the 34th International Conference on Software Engineering*, pages 3–13, 2012.

[8] 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.

[9] 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 *Proceedings of the 36th International Conference on Software Engineering*, pages 234–242, 2014.

[10] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. SemFix: Program Repair via Semantic Analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781, 2013.

[11] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering*, pages 254–265, 2014.

[12] S. Sumi, Y. Higo, K. Hotta, and S. Kusumoto. Toward Improving Graftability on Automated Program Repair. In *accepted by the 31st International Conference On Software Maintenance and Evoluation*, page (to appear), 2015.

[13] X. Wang, S. C. Cheung, W. K. Chan, and Z. Zhang. Taming Coincidental Correctness: Coverage Refinement with Context Patterns to Improve Fault Localization. In *Proceedings of the 31st International Conference on Software Engineering*, pages 45–55, 2009.