Automated Program Repair by Using Similar Code Containing Fix Ingredients

Tao Ji, Liqian Chen, Xiaoguang Mao, Xin Yi
College of Computer
National University of Defense Technology
Changsha, China
jitao_91@163.com, {lqchen, xgmao}@nudt.edu.cn, yixin_09@163.com

***Abstract*—Recently, much attention has been paid on program repair by reusing existing code from other software. However, the technique of reusing code needs to search fix ingredients which refer to the existing code that can be reused to form a fix, and the searching space tends to be huge. Finding out those code fragments that contain proper fix ingredients efficiently will largely improve repair efficiency. Based on the assumption that similar code fragments may contain fix ingredients, this paper proposes reusability metrics of similar code fragments for program repair. By combining the similarity and differentiality at the level of program syntax trees, reusablility metrics is able to help picking out the most suitable reusable candidate. In order to apply reusability metrics to automated program repair, we have implemented SCRepair, which can utilize the guidance of reusability metrics to automatically fix bugs. Experimental results indicate that SCRepair can improve repair efficiency by making use of the reusability metrics of similar code.**

***Keywords*-Program Repair; Similar Code; Reusability Metrics; Fix Localization**

I. Introduction

Recently, considerable attention has been paid on automated program repair, which aims at automatically generating patches without manual efforts [1]–[4]. Generally speaking, the automated repair work consists of three major steps: 1) fault localization; 2) generation of candidate patches; 3) validation of candidate patches. The process of generating repair candidates plays an important role in the repair work.

In the area of search-based repair, much attention has been paid on reusing existing code and reusing fix operations. Due to the habit of copy-and-paste in programming, similar code fragments may contain the same bug which may be neglected to fix during software maintenance. Under the hypothesis that recurring bugs are produced due to software reusing, the technique of reusing fix operations to repair faulty program is plausible [4] [5]. However, when the fix operations are not available (e.g., a functionally similar code fragment was implemented by other developers without any known bugs), we fail to use the technique of reusing fix operations to fix the faulty program. On the other hand, reusing existing code can fix the bug with no need of fix operations from historical versions of software. However, this repair technique is limited by the huge searching space of fix ingredients from other programs. Fix ingredients refer to the existing code that can be reused to fix the faulty program [6]. Finding out those code fragments that contain proper fix ingredients efficiently will improve a lot the efficiency of repair. In other words, if we can correctly rank the code fragments by the probability of containing fix ingredients, the repair work will achieve better performance.

Based on the assumption that similar code may contain fix ingredients, we propose reusability metrics of similar code fragments for program repair. In addition, to apply the reusability metrics of similar code to automated repair tools, we implement a prototype repair tool called SCRepair (Similar-Code-based Repair).

Specifically, this paper makes the following contributions:

- To our knowledge, this paper is the first to propose reusability metrics of similar code fragments for program repair (Section III). Experimental results show that reusability metrics can reduce the cost of finding out similar code fragments that contain fix ingredients.
- We implement a repair tool called SCRepair, which can leverage different techniques of fix localization (Section IV). Under the assumption that similar code may contain fix ingredients together with reusability metrics, this paper proposes two strategies of fix localization to guide the SCRepair to fix bugs. Experimental results indicate that SCRepair can fix bugs efficiently by using those two strategies.

II. Background and Related Work

A. Program Repair

Semantic-based repair can fix simple bugs by synthesizing correct code that meets the specifications. However, these semantic-based repair tools fail to scale well to large-scale programs due to the limitation of static analysis methods [2] [3]. **Search-based repair** transforms the problem of program repair into the problem of searching the correct repair. For example, GenProg uses genetic programming to generate candidate patches by inserting code from other locations in the faulty program [1]. CodePhage automatically locates correct code in one application, and then transfers that code into another application [4]. Search-based repair has advantages over semantic-based repair on the size and complexity of faulty programs.

B. Similar Code Detection

Similar code detection techniques have been applied in many fields successfully, such as modifying clones consistently, bug detection, software property protection and so on. In the literature, many detection tools were implemented based on different approaches, and most of them perform well to find the textual similar code fragments, as shown in Roy's work [7].

C. Similar Code Detection used in Program Repair

SecureSync [5] measures the similarity between the reported vulnerable code and the given code. If the similarity between the reported vulnerable code and the suspicious code is greater than a given threshold, and the similarity will be decreased when the reported vulnerable code is fixed, the fix operation will be recommended to fix the vulnerable code. CodePhage [4] uses two instrumented executions of the donor to identify the correct code to transfer into the recipient: one positive testcase and one negative testcase enable CodePhage to isolate a single check that is presented in the donor but absented in the recipient. By comparing the donor with the recipient shown in [4], we find that these two code fragments are similar at the level of functions. The cases of SecureSync and CodePhage have shown the effectiveness of applying similar code to program repair.

III. Reusability Metrics

In this section, we present our reusability metrics of similar code fragments for program repair.

A. Motivating Example

Fig. 1 presents two code fragments from different students' answers during the final exam of a C programming course in our university. Students are required to compute the minimum value that meets the constraints by enumeration. The detailed constraints are as follows. Given a ($a\in[1,30]$), compute the minimum value of $b+c$ (b, $c\in$ [1,1000]), where a, b, c are integers, and a, b, c should satisfy the constraint: $|\arctan(1/a)-\arctan(1/b)-\arctan(1/c)|<10^{-8}$. The code fragment shown in Fig. 1(a) fails to compute the minimum value of $b+c$ and the IF-condition is wrong, which is correctly implemented in Fig. 1(b). On the other hand, the code fragment in Fig. 1(b) lacks the check for the value of a, which exactly appears in line 4 of Fig. 1(a).

Suppose that we have constructed a database of all students' code fragments. The technique of reusing fix operations obviously cannot be applied here to repair the faulty program, because fix operations are not available. Hence, we choose the technique of reusing existing code to repair faulty programs. GenProg randomly picks the inserted code that can pass the semantic check, which can filter out the code containing undefined variables. However, the cost of repair will still be high due to the similar naming rules in students' code. CodePhage requires two test cases to locate the correct IF statement. One positive testcase is used to locate the code that has similar property and one negative testcase ensures that the code does not have the same bug. However, it is disappointing that the positive testcase of Fig. 1(a) may failed in Fig. 1(b) and the positive testcase of Fig. 1(b) also may failed in Fig. 1(a). Consequently, CodePhage is unable to repair Fig. 1(a) and Fig. 1(b) by locating correct code from each other.

By comparing these students' code, we find that these code fragments are similar due to the same target functionality. The two code fragments have similar syntax trees, and the different parts exactly contain the fix ingredients.

To improve the efficiency of fix localization, we now propose the reusability metrics of similar code fragments considering the similarity and differentiality between them. After ranking the similar code by the probability of containing fix ingredients, software developers or automated repair tools can efficiently locate fix ingredients.

B. Metric Formula

Although those similar code detection tools have different match algorithms, they all compute the similarity between different code fragments. Similarity can represent the proportion of similar part well. For example, based on syntactic approaches, the literature [8] computes the syntax trees similarity using the following equation:

$$Sim = \frac{2 * S}{2 * S + L + R} \tag{1}$$

where *Sim* represents the similarity between two syntax trees, S represents the number of the same nodes, L and R represent the numbers of different nodes in each syntax tree respectively. Other approaches of calculating similarity between code fragments also can be applied here in reusability metrics.

In the process of similar code detection, a threshold of similarity is needed to control the number of candidates. Intuitively, the greater similarity we get, the higher likelihood of implementing the same function and containing fix ingredients will be in those similar code. However, the exactly identical code fragments that contain the same bug should be filtered out from the database of candidates.

It is obvious that the different parts of similar code fragments are more likely to contain the fix ingredients. Existing work [5] shows that similar code fragments that have high similarity often have the same bug, and the next version of this fragment that has relatively smaller similarity always contains fix ingredients. Based on this insight, not only the similarity, but also differences between similar code fragments should be considered for program repair.

Now the question is how to compute the value of *Dif*, which represents the metric value of differentiality. First, we use the notion of change action to refer to a kind of source code modification as shown in [9]. ChangeDistiller [10] is

```
1: int solution(int a){
2:   int b,c;
3:   int sum=0;
4:   if(a>=1&&a<=30){
5:   for(b=0;b>=0&&b<=1000;b++){
6:     for(c=0;c>=0&&c<=1000;c++){
7:       if(fabs(b-c)<1.0e-8&&atan(1/a)==atan(1/b)+atan(1/c))
8:         sum=b+c;
9:     }
10:  }
11:  return sum;}
12:  else return 0;}
```

(a) a code fragment that does not compute *b+c* correctly

```
1:int solution(int a)
2:{
3:  int min=100000;
4:  for(int b=1;b<1000;b++)
5:  for(int c=1;c<1000;c++)
6:  {
7:    double m=atan(1.0/a)-atan(1.0/b)-atan(1.0/c);
8:    if(fabs(m)<0.00000001)
9:      if(min>b+c)
10:       min=b+c;
11:  }
12: return min;}
```

(b) a code fragment that misses the check for the value of *a*

Figure 1: Two Code Fragments from Different Students' Answers during An Exam

a fine-grained AST differencing tool for Java. It expresses fine-granularity source code changes using a taxonomy of 41 source changes types, such as "statement insertion" and "if conditional change". Since change actions vary when the types of bugs are different, each change action should make different contributions to the value of *Dif*. While some change actions are common, other change actions are rare in bug fix activities. For example, imagine that both B and C are similar with the buggy code fragment A. The change action from A to B is "insertion of IF statement" whereas the change action from A to C is "change of variable name". In most cases, to fix the bug, software developers prefer B to C. In the metrics, we calculate the sum of the weight values of all change actions. However, the higher number of change actions does not mean the higher probability of containing fix ingredients. Based on these insights, we put forward the mean value of the weights of all change actions to calculate *Dif*.

$$Dif = \frac{\sum_{i=1}^{n} Weight_i}{n * Weight_{max}} \tag{2}$$

where $Weight_i$ represents the weight value of the *i*-th change action, *n* represents the number of change actions, $Weight_{max}$ is the normalization divisor and represents the maximum weight value of all change actions.

As described earlier, *Sim* and *Dif* are two key parameters in reusability metrics for program repair. We expect reusable similar code fragments to have the greater *Sim* and *Dif* values than others in reusability metrics. Based on the analysis above, we propose the following reusability metrics for program repair:

$$R = \begin{cases} 0 & Sim<TH \text{ or } Sim=1, \\ \frac{1}{2} * (Dif + \frac{Sim-TH}{1-TH}) & 1>Sim\geq TH. \end{cases} \tag{3}$$

where *R* represents the recommendation value of code fragment (where $1>R\geq 0$, and the higher the value, the higher the likelihood of containing fix ingredients), *TH* represents the threshold of *Sim* which can be empirically given by similar code detection tools.

The weight model of change actions to determine $Weight_i$ in reusability metrics should efficiently reflect the reusability of change actions. This paper will use a weight model of change actions from the previous work [9] to calculate *Dif*.

C. Experimental Design

In our experiments, we use NICAD [11] to compute *Sim* and the change action model CTET [9] to compute *Dif*. In addition, we set the weight value of each change action by the "ALL" sample of CTET.

We randomly picked out six groups of similar code fragments from Tomcat70[1] to construct the benchmark. TABLE I shows detailed information of this benchmark. The column "Buggy Code Fragment" shows the buggy method of a particular version. For example, "java.org.apache.coyote.ajp.AjpProcessor.process" represents the method name, and "88108dd387" represents the part of commit number which represents the particular version of this method. The column "Candidates" shows the method that may contain fix ingredients, while the column "*Sim*≥0.3" shows the number of candidates whose similarity values are greater than 0.3.

Note that during the process of computing reusability metrics, we have filtered out the debugging statements that do not affect program behaviors.

D. Results and Analysis

Fig. 2 shows *R* values of candidates from "java.org.apache .coyote.ajp.AjpAprProcessor.process" by using reusablity metrics. We use the *NCC* (Number of Checked Candidates) to evaluate the effectiveness of reusability metrics. *NCC* denotes the number of candidates that we have checked before we find one reusable candidate. TABLE II shows the *NCC* scores of given candidates by different methods. The

[1] git://git.apache.org/tomcat70.git

Table I: Six Groups of Similar Code

No.	Buggy Code Fragment	Candidates	Sim≥0.3
1	java.org.apache.coyote.ajp.AjpProcessor. process-88108dd387	java.org.apache.coyote.ajp. AjpAprProcessor.process	33
2	java.org.apache.coyote.http11.InternalAprInputBuffer. parseRequestLine-6e05c982b4	java.org.apache.coyote.http11. InternalInputBuffer.parseRequestLine	6
3	test.org.apache.catalina.tribes.demos.IntrospectionUtils. setProperty-93df08d1f2	java.org.apache.tomcat.util. IntrospectionUtils.setProperty	10
4	java.org.apache.jasper.compiler.TldLocationsCache. tldScanResourcePaths-e78978d6ce	java.org.apache.catalina.startup. TldConfig.tldScanResourcePaths	12
5	java.org.apache.coyote.ajp.AbstractAjpProcessor. asyncDispatch-6e05c982b4	java.org.apache.coyote.http11. AbstractHttp11Processor.asyncDispatch	5
6	java.org.apache.catalina.ha.session.DeltaSession. isValid-2acefbd810	java.org.apache.tomcat.util. net.AprEndpoint.isValid	7

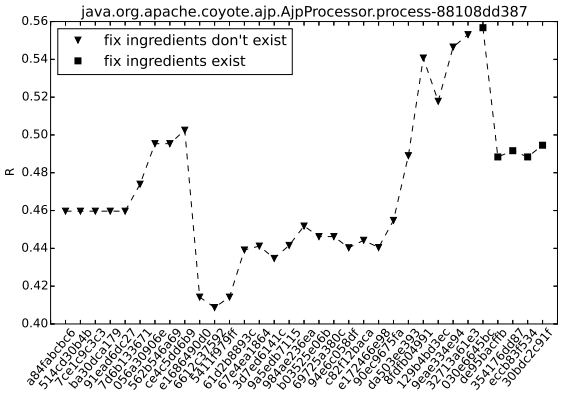

Figure 2: Candidates from java.org.apache.coyote.ajp.AjpAprProcessor.process

column "Randomness" represents the mean value of *NCC* by randomly picking out from all similar fragments; the column "Similarity" represents the *NCC* by using rankings of similarity values; the column "Reusability Metrics" represents the *NCC* by using rankings of reusability metrics values.

Table II: *NCC* of Picking Out Randomly, Guiding by Similarity and Reusablity Metrics

No.	Randomness	Similarity	Reusability Metrics
1	5.7	3	1
2	3.5	2	1
3	2.8	3	2
4	3.2	2	2
5	3	2	2
6	2.7	3	2

The results showed in TABLE II indicate that each buggy code fragment's *NCC* decreases obviously after using reusability metrics. Then, we use nonparametric statistics to analyze the result data. For nonparametric Mann-Whitney-Wilcoxon test, the null hypothesis is that result data from sample$_1$ and sample$_2$ share the same distribution; the alternate hypothesis is that the two group data have different distributions. We say the improvement of sample$_1$ over sample$_2$ is statistically significant when we reject the null hypothesis at a 0.05 significance level and the mean value of sample$_1$ is smaller than sample$_2$'s. We compute the *p*-values of Mann-Whitney-Wilcoxon test between these three samples. The *p*-value between "Randomness" and "Reusability Metrics" is 0.004267. The *p*-value between "Similarity" and "Reusability Metrics" is 0.03788. Given the 0.05 significance level, the improvement of "Reusability Metrics" over "Randomness" and "Similarity" is statistically significant.

In the candidates of "java.org.apache.tomcat.util.net.AprEndpoint.isValid", there are two reusable candidates. One reusable candidate ranks second in reusability metrics values. However, the metrics value of the other reusable candidate is the smallest. After analyzing the faulty program and this reusable candidate, we find that the error in computing the similarity of ordering IF-statements leads to the smallest metrics value. Moreover, we have used the reusability metrics to conduct experiments on the students' programs as shown in the Fig. 1. However, the *NCC* score is not low enough, because students' code has many different fragments that own high weight values of change actions in reusability metrics, such as updating of variable names and changing loops. Because "ALL" is composed of all commits, the probability values of change actions cannot reflect the frequency over activities of fixing one specialized bug. After increasing the weight of updating IF condition and decreasing the weight of updating variable names, the *NCC* score decreased. We believe that reusability metrics can perform better if we can set the weight value more reasonable according to bug types.

Given the analysis above, we find that the reusability metrics is effective for program repair when similar code contains fix ingredients.

IV. SCREPAIR

This section firstly describes the implementation of SCRepair, a prototype repair tool that can leverage different fix localization techniques. Then we present two fix localization strategies, which are based on the rankings of similarity

and reusability metrics. Finally, we analyze the experimental results.

A. Implementation

We implement SCRepair on top of RSRepair [12]. Like GenProg, RSRepair fails to leverage other fix localization techniques to find out fix ingredients efficiently. SCRepair adapts the same fault location strategy of RSRepair to locate faulty code. After setting each statement the probability value of containing errors, SCRepair uses the same mutation operators of RSRepair to modify the faulty program. Note that the insertion operator needs code from other locations, which is the main difference between SCRepair and RSRepair. In the process of generating candidate repairs of SCRepair, each statement of the faulty program and all candidates owns the probability value given by fix localization. The greater the probability value is, the more likely the statement will be chosen to insert into the fault location. After generating the candidate repair, SCRepair uses test cases to ensure that the bug has been eliminated.

Note that similar code may contain some different variables and code structures that cannot be directly reused in faulty program. At present, SCRepair does not support the normalization of variables and other syntactic elements.

B. Strategies of Fix Localization

To apply metrics in automated repair, we present two fix localization strategies based on the rankings of code fragments by similarity values and reusability metrics results.

SCRepair-sim, is a fix localization strategy, which is based on the insight that similar code fragments may contain fix ingredients. After picking out the similar code fragments, we set the statements in those remaining fragments the same value $prob_0$, as the probability of reusing, defined as

$$prob_0 = \frac{1}{2n} \tag{4}$$

where *n* represents the number of all statements. We set the statements from the similar code fragments the reusing probability $prob_{sim}$, defined as

$$prob_{sim} = \frac{1}{2\sum_{i=1}^{m} l_i} + \frac{1}{2n} \tag{5}$$

where *m* represents the number of similar code fragments, l_i represents the number of statements from the *i*-th similar code fragment.

SCRepair-metric, is a fix localization strategy, which uses reusability metrics to guide SCRepair to repair faulty programs automatically. This strategy set the same values as SCRepair-sim for those statements that are not similar to faulty programs. However, unlike SCRepair-sim, SCRepair-metric sets different probability values for the statements from those similar code. Essentially, the similar code fragment can be divided into two parts: one similar part and one different part. The similar part always does not contain any fix ingredients at all, while the different part often contains the fix ingredients exactly, which is depicted in Fig. 1. Hence, we set the same probability value given by equation (4) for those similar parts and set different probability value $prob_j$ for those statements from the different part of the similar code fragment whose *NCC* score is *j*. $prob_j$ is defined as

$$prob_j = \frac{m-j+1}{2\sum_{k=1}^{m}(d_k * (m-k+1))} + \frac{1}{2n} \tag{6}$$

where d_k represents the number of statements from the different part of the similar code fragment whose *NCC* score is *k*.

C. Experimental Design

SCRepair relies on test cases to validate the candidate repair, but the test cases of Tomcat70 are not available. As a result, we choose the Introclass [13] to conduct repair experiments. In this benchmark, there are six assignments that each student should complete. Due to the same goal of each assignment, students' programs are functionally similar to some degree. We randomly choose three faulty programs from different assignments to measure the effectiveness of presented fix localization strategies. For each buggy program, to construct the database of candidates, we randomly choose three programs whose similarity values are greater than 0.3 given by NICAD and six programs whose similarity values are smaller than 0.3. All students' programs have been rewritten from C to Java without changing semantics so that programs can be handled by ChangeDistiller.

Because of the randomized algorithm in SCRepair, we use SCRepair to repair every faulty program for 100 times. Meanwhile, we have limited the number of candidate patches produced to reduce the cost of computation. As described in [12], when more than 400 patches are produced, we consider this case as that the repair fails to find a valid patch.

D. Results and Analysis

Table III: the Success Rate of Different Tools

Faulty Program	RSRepair	SCRepair-sim	SCRepair-metric
digits-d120480a-002.c	86%	98%	100%
grade-f5b56c79-012.c	91%	93%	100%
median-07045530-001.c	100%	100%	100%

TABLE III shows the success rate when using one repair tool to fix each faulty program. "digits-d120480a-002.c" represents the program was completed by the student whose ID was hashed into "d120480a" in the third commit to solve the problem of "digits". The data of the experiments on fixing the "digits-d120480a-002.c" and "grade-f5b56c79-012.c" indicates that the success rate increases when we apply two strategies respectively. The data of the experiment on fixing the "median-07045530-001.c" indicates that SCRepair works at least as well as RSRepair. Fig. 3 depicts

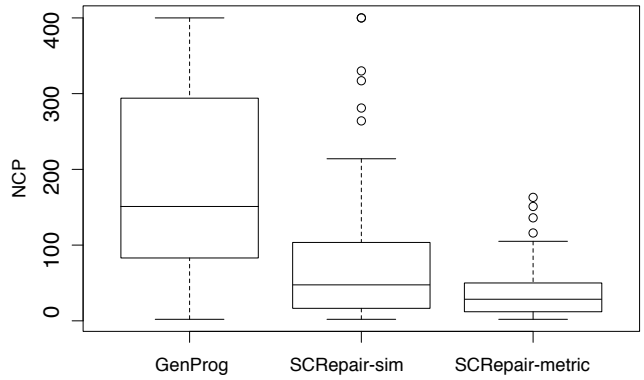

Figure 3: Boxplot on *NCP*s of digits-d120480a-002.c

the detailed results of the experiment on fixing the "digits-d120480a-002.c" in terms of *NCP* (Number of Candidate Patches) [12].

After analyzing experimental results, we find that the mean value of *NCP*s decreases obviously in each experiment. Then, we use nonparametric Mann-Whitney-Wilcoxon test to analyze the result data. In the experiments on fixing the "digits-d120480a-002.c" and "grade-f5b56c79-012.c", the p-values are much smaller than the given significance level, which indicates that the repair effectiveness is significant after using proposed fix localization techniques. However, in the experiment on fixing the "median-07045530-001.c", the p-value between "SCRepair-sim" and "SCRepair-metric" is 0.693, which means that the difference between distributions is not statistically significant in the significance level 0.05. After analyzing the faulty program and its candidates manually, we find that all similar candidates contain fix ingredients, which leads to the almost equal reusing probability values of fix ingredients.

The analysis above shows that SCRepair can efficiently fix bugs with the guidance of reusability metrics.

V. Threats to Validity

We leverage NICAD to compute the similarity and adapt the probability model "ALL" of CTET to compute the differetiality. These two parameters are crucial for metrics result. Moreover, the size of these benchmark programs is small and the errors are simple. In future work, we plan to conduct more experiments to validate the effectiveness of reusablity metrics and SCRepair.

VI. Conclusion

Similar code detection techniques become playing an important role in the field of program repair activities. This paper presents the reusability metrics to rank the similar code fragments so that developers or automated repair tools can efficiently find out the fragments that contain fix ingredients. The results of our experiments indicate that reusability metrics can perform better than picking out randomly and ranking by similarity. We implement an automated repair tool called SCRepair, which can be guided by different fix localization techniques. Based on the rankings by similarity values and reusability metrics values, this paper presents two fix localization strategies and carries out experiments to compare the repair effectiveness. The results indicate that using the reusability metrics to guide SCRepair can achieve better repair effectiveness.

Acknowledgment

This research is supported by the National Natural Science Foundation of China under Grant No.91318301 and No.61379054.

References

[1] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *IEEE TSE*, vol. 38, no. 1, pp. 54–72, 2012.

[2] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," in *ICSE'10*. ACM, 2010, pp. 61–72.

[3] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *ICSE'13*. IEEE Press, 2013, pp. 772–781.

[4] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *PLDI'15*. ACM, 2015, pp. 43–54.

[5] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *ASE'10*. ACM, 2010, pp. 447–456.

[6] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *ICSE'14*. ACM, 2014, pp. 492–495.

[7] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

[8] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM'98*. IEEE, 1998, pp. 368–377.

[9] M. Martinez and M. Monperrus, "Mining software repair models for reasoning on the search space of automated program fixing," *Empirical Software Engineering*, vol. 20, no. 1, pp. 176–205, 2015.

[10] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE TSE*, vol. 33, no. 11, pp. 725–743, 2007.

[11] C. K. Roy and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *ICPC'08*. IEEE, 2008, pp. 172–181.

[12] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *ICSE'14*. ACM, 2014, pp. 254–265.

[13] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The manybugs and introclass benchmarks for automated repair of c programs," *IEEE TSE*, vol. 41, no. 12, pp. 1236–1256, 2015.