symmetry

MDPI

Article

Quality Evaluation Method of Automatic Software Repair Using Syntax Distance Metrics

Yukun Dong *, **Daolong Tang, Xiaotong Cheng and Yufei Yang**

College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China
* Correspondence: dongyk@upc.edu.cn; Tel.: +86-1830-022-9553

Abstract: In recent years, test-based automatic program repair has attracted widespread attention. However, the test suites in practice are not perfect ways to guarantee the correctness of patches generated by repair tools, and weak test suites lead to a large number of incorrect patches produced by the existing repair tool. To reduce the number of incorrect patches generated by repair tools, we propose a patch quality evaluation method based on syntax distance metrics, which measures the syntax distance of patches through four evaluation features—variable, expression, structure, and repair location. By fusing the distance values of the four features, the quality of a patch can be evaluated. Our method evaluates 368 patches from multiple famous repair tools, such as jKali, Nopol, SimFix, DynaMoth, and CapGen; 95% of the correct patches were ranked in the top one of the plausible patches for each defect, which indicates our method can find high-quality patches.

Keywords: automatic program repair; program patch; quality evaluation; patch ranking; syntax distance

check for updates

Citation: Dong, Y.; Tang, D.; Cheng, X.; Yang, Y. Quality Evaluation Method of Automatic Software Repair Using Syntax Distance Metrics. *Symmetry* **2022**, *14*, 1751. http://doi.org/10.3390/sym14081751

Academic Editor: Alexander Shelupanov

Received: 28 June 2022
Accepted: 19 August 2022
Published: 22 August 2022

1. Introduction

Repairing the defects in software is a time-consuming process. According to relevant studies, the process of repairing the defects accounts for 50% of the overall development cost of software products [1]. In particular, the complexity and scale of modern software are increasing, making it harder to repair software defects. Therefore, automatic program repair technology has received more attention, and relevant methods and tools have played vital roles in ensuring software quality. The current automatic program repair methods are classified into the following two families:

(1) **Generation and verification methods**: This method uses a series of mutation operations to modify the buggy program and uses test cases as the standard to evaluate the correctness of the patch [2]. Test cases cannot cover all paths and functions of the program, which will lead to a large number of overfitting patches. Although the generated patches can pass all test cases, they still contain a large number of incorrect patches.

(2) **Semantically driven methods**: The method finds the correct specification of the program and generates constraints that can guide the process of repairing the buggy program. Firstly, It uses symbol execution technology to obtain program execution information and path constraints. Then it uses a constraint solver to convert path constraints into constraint conditions, and finally uses conditional synthesis to repair the original buggy program. A buggy program is turned into a constraint, and repair tools produce patches that satisfy the constraint [3]. However, the patches produced by these repair tools contain many incorrect patches.

Because test cases are not good specifications for evaluating the correctness of patches [4], the accuracy of existing repair tools is quite low [5]. We call a patch a plausible patch if it passes all test cases, and a patch is a correct patch if it only repairs the defect and does not introduce new defects. The patch is known as the correct patch if it only repairs the defect and does not introduce new defects. Existing studies [6,7] show that patches generated by existing repair tools contain large numbers of incorrect patches, the number of incorrect patches is greater than the number of correct patches. When plausible patches contain a large number of incorrect patches, the developer must verify the correctness of many plausible patches to find the correct patches. The verification process takes more time than it would take for a developer to directly repair the defects [8]. Although plausible patches can pass all test cases, these patches may still contain semantic defects, such as null pointer defects and array out-of-bounds defects, and contain program syntax defects, such as irregular expressions and overly complex conditional expressions.

Test suites in practice are not perfect specifications to guarantee the correctness of patches generated by repair tools, and weak test suites lead to a large number of incorrect patches produced by the existing repair tool. Facing the challenge of low accuracy of automatic repair tools, many researchers have proposed methods to judge the correctness of a patch; it can be either static or dynamic method depending on whether the test cases are executed. Dynamic methods generally need to generate new test cases. Xiong et al. proposed a method to generate new test cases according to the similarity of test cases [4]. A static method can prioritize or filter out plausible patches according to the static characteristics of patches. Tan et al. proposed anti-patterns to judge incorrect patches that meet specific static structures [9]. Dong et al. proposed a static analysis method to identify patches, including semantic defects, such as the null pointer reference [10]. SSDM uses semantic features to prioritize plausible patches [11]. CapGen [12] sorts patches based on genealogy structure, access variables, and semantic dependencies. Al-Batai et al. integrated static bug detection techniques with automated program repair to strengthen specifications and improve the quality of repairs [13]. Dong et al. proposed a static analysis method using the data flow information of the program and special specifications [14]; it can effectively detect semantic defects, such as the null pointer reference. Therefore, we integrated static analysis techniques with test cases to strengthen specifications, and propose an evaluation method, which analyzes the syntax distance of the patch to prioritize patches.

Symmetry and similarity before and after program repairs are mainly via three aspects: variable symmetry, expression symmetry, and structural symmetry. Variable symmetry means that the variables between fault statements and repair statements are similar. Expression symmetry means that the correct repair expression can be found in the buggy program. Structure symmetry means that the structure between the fault statement and the repair statement is similar. For the similarity and symmetry before and after program repair, a program patch quality evaluation method is proposed using syntax distance between the patch and buggy program; four features—variable, expression, structure, and repair location—are proposed to measure the syntax distance. For each feature, we propose a specific calculation formula to quantify the syntax distance between the repaired program and the original buggy program, and give a patch recommendation based on the syntax distance. The syntax distance mainly focuses on the text features of the programs, which indicates the difference before and after the program repair. We integrate four text features—variable, expression, structure, and repair location—to obtain the syntax distance between the buggy program and the patch program. Then the patch quality is evaluated according to the syntax distance and a patch with a higher syntax distance. Our method was evaluated on 368 patches generated by multiple automatic program repair tools, including Nopol [15], DynaMoth [16], CapGen, jKali [17], and so on. Most repair tools only generate a patch for each defect, such as Nopol, DynaMoth, jKali, SimFix [18], jGenProg [17], and ARJA [19], except CapGen. To verify the evaluation effect of our method, patches generated by multiple repair tools are prioritized for a defect. The experimental results show that the correct patches with higher rankings are recommended in priority and compared with incorrect patches, the recommended value of most correct patches is higher. Finally, our evaluation method is indeed advanced compared with other existing evaluation methods; comparison results show that our method successfully obtains high-quality patches and improves the repair accuracy.

The followings are the main contributions of this paper: (1) We found four heuristic rules to effectively distinguish correct patches from incorrect patches, which are variable similarity, expression similarity, structural similarity, and repair location; (2) A program patch quality evaluation method is proposed to prioritize patches, which uses four features—variable similarity, expression similarity, structural similarity, and repair location; (3) We integrate static analysis techniques with test suites to strengthen specifications.

This paper is organized as follows. The first section is the introduction. The second section introduces the related work of our study. The third section presents the patch recommendation method based on the syntax distance. The fourth section introduces the experiment and analyzes the results of the experiment. The fifth section presents the threats to validity. The sixth section presents the discussion and limitations. The seventh section presents the conclusions of our work.

2. Related Work

Recently, many researchers have focused on automatic program repair technology to reduce the time and effort of repairing defects [1], most of our earlier methods used redundancy assumptions to reduce the number of incorrect patches [6,7]. Current repair tools generally use test cases as the specifications to verify the correctness of patches. Patches that pass all test cases are not necessarily correct [20,21]. These patches contain a large number of incorrect patches, which increase the difficulty of manual judgment instead of achieving the purpose of the repair defect.

The generation and verification methods apply a series of modification operations to repair the original buggy program. jKali involves the Java versions of Kali [22]; it attempts to repair defects by deleting statements in the original program. However, it will delete the functional code and change the semantics of the original program, which can lead to generate a large number of incorrect patches. jGenProg involves the Java versions of GenProg [6]; it transforms the original buggy program into an abstract syntax tree, and the nodes are randomly added, deleted, and replaced in the abstract syntax tree to generate patches. However, the patch validation of jGenProg requires too much time, and the accuracy of jGenProg is quite low. SimFix is a new automatic program tool; it mines an abstract search space from existing patches and obtains a specific search space from similar code fragments. CapGen is also one search-based tool, it not only mines code modifications in historical repair operations to guide patch generations, but also considers the context information of code modification. Moreover, it outperforms the existing state-of-the-art approaches. jMutRepair [17] is a repair tool-based mutation operator; it applies operators, such as relational operators, logical operators, and unary operators in conditional statements, and repairs programs by modifying these operators. Tbar [23], FixMiner [24], AVATAR [25], kPar [26], and Cardumen [27] are automatic program repair tools that use templates to repair patches. However, they cannot repair defects that cannot capture the patterns and their accuracies are also very low. Yang et al. [28] proposed an automatic repair method by using similar fix information based on genetic programming; it finds the fixing ingredient related to the most similar source to generate the patches. It can reduce the time to repair a defect and improve the quality of the patch. DTSFix [14] can precisely identify the data and control the dependencies of the buggy program to generate candidate patches. However, DTSFix can repair special semantic defects, but cannot repair non-semantic defects. DEAR [29] is an automated program repair model based on deep learning. In Table 1, repair tools based on the generation and verification method use different types of information to guide patch generation. In Table 2, we stated the version number of the repair tool, creator of the repair tool and the location from where the repair tool was sourced.

The semantically-driven method utilizes symbol execution technology to obtain execution information and path constraints. Then it uses the constraint solver to convert execution information and path constraints into constraint conditions; it finally uses conditional synthesis to generate the patch. Nopol can repair defects of if-conditions; it uses symbol execution technology to gather execution information and path constraints about if-conditions. Then the information is transformed into a problem of satisfiability modulo theories (SMT) [30], and correct if-conditions can be obtained by solving the SMT problem. However, the accuracy of the Nopol is very low. DynaMoth is an automatic program repair tool based on Nopol; it can solve the problem of method calls but Nopol cannot, and it gathers the dynamical runtime information, including parameters, variables, method calls, etc. The DynaMoth outperforms the Nopol. The repair tool ACS [31] uses three different kinds of information to sort the patches, including the conditional expressions, the report of the defect program, and the structure of the defect program; specifically, the patch is synthesized by the variable used in the conditional expression and the predicate executed on the variable. The ACS outperforms the DynaMoth. In Table 1, repair tools based on the semantically-driven method use different types of information to guide the patch generation.

To solve the problem of low accuracy of the automatic repair tools, many methods have been proposed. Tan et al. proposed anti-patterns [32] to judge incorrect patches that meet specific static structures; it can exclude the correct patches that do not meet the special pattern. SSDM [11] is a patch quality evaluation tool based on semantic distance, which uses features of interval distance, output coverage, and path matching to evaluate patches. SSDM can evaluate patches of defects involving multiple statements, but cannot evaluate patches of defects within a single statement. Xiong et al. proposed a method to generate new test cases according to the similarity of test cases. Because the generation of new test cases is very difficult, it only generates test cases for a small number of defects. Al-Batai et al. integrated static bug detection techniques with automated program repair to strengthen specifications and improve the quality of repairs [13]. Dong et al. proposed a static analysis method using the data flow information of the program and special specifications [14]; it can effectively detect semantic defects, such as null pointer reference, but it cannot detect defects without complete specifications. SimFix uses the syntax similarity of error codes and repair components to prioritize patches. Syntax similarity mainly focuses on text similarity, such as variable name similarity. CapGen uses three models to estimate the correct probability of patches and sort patches based on genealogy structure, access variables, and semantic dependencies. CapGen and SimFix cannot effectively evaluate patches of semantic defects.

There are some insurmountable problems with the above repair methods and tools, resulting in some plausible but incorrect patches in the generated patches, which cannot be detected by the test cases and need to be manually eliminated by the developer. Therefore, we integrated static analysis techniques with test cases to strengthen specifications and improve the quality of repairs, and propose an evaluation method that analyzes the distance between the buggy program and the patch from the syntax point of view to evaluate the quality of a patch.

Table 1. The main differences in the repair tools.

Available Information	Generation and Verification	Semantically-Driven
No Information	jKali, jGenProg, jMutRepair, ARJA	Nopol, DynaMoth
Syntax Information	CapGen, GPSBFI	-
Semantic Information	CapGen, SimFix, DTSFix, DEAR	ACS
Historical Information	CapGen, SimFix	-
Template Information	Tbar, FixMiner, AVATAR, kPar, Cardumen	-

Table 2. The main information of repair tools.

Repair Tools	Version Number	Creator	Location
jKali	2016	Martinez et al.	Orlando, FL, USA
jGenProg	2016	Martinez et al.	Orlando, FL, USA
jMutRepair	2016	Martinez et al.	Orlando, FL, USA
ARJA	2018	Yuan et al.	East Lansing, MI, USA
Nopol	2017	Xuan et al.	Wuhan, China
DynaMoth	2016	Durieux et al.	INRIA, France
CapGen	2018	Wen et al.	Hong Kong, China
SimFix	2018	Xiong et al.	Beijing China
DTSFix	2022	Deng et al.	Qingdao, China
ACS	2017	Xiong et al.	Beijing, China
Tbar	2019	Liu et al.	Luxembourg
FixMiner	2020	Koyuncu et al.	Luxembourg
AVATAR	2019	Liu et al.	Luxembourg
kPar	2019	Liu et al.	Luxembourg
Cardumen	2016	Martinez et al.	Orlando, FL, USA
DEAR	2022	Li et al.	New Jersey, USA

Version Number represents the version number of the repair tool, Creator represents the developer of the repair tool, Location represents the location where the repair tool was sourced.

3. Patch Recommendation Based on Syntax Distance

Figure 1 shows the overview of automatic program repair and evaluation; it works in four steps—fault localization, patch generation, patch quality evaluation, and patch prioritization. This paper analyzes the impact of syntax similarities on patch prioritization.

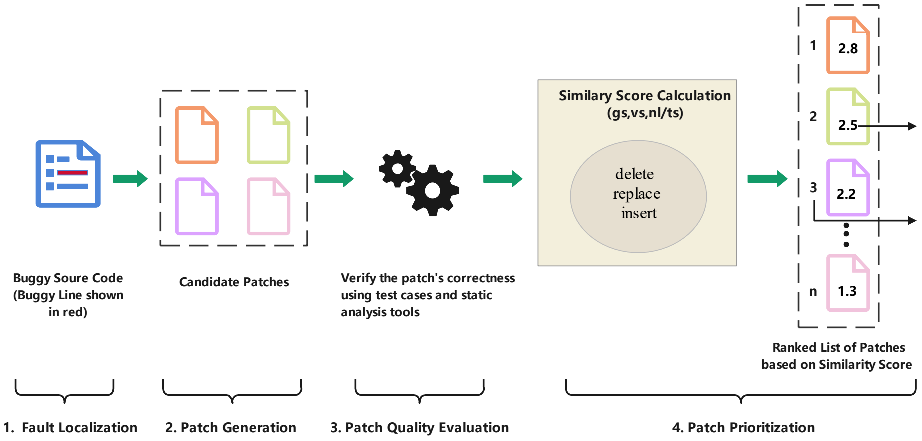
Buggy Soure Code (Buggy Line shown in red)
Candidate Patches
Verify the patch's correctness using test cases and static analysis tools
Similary Score Calculation (gs,vs,nl/ts)
delete
replace
insert
1
2.8
2
2.5
3
2.2
n
1.3
similarity score
rank
Ranked List of Patches based on Similarity Score
1. Fault Localization
2. Patch Generation
3. Patch Quality Evaluation
4. Patch Prioritization

Figure 1. Overview of automatic program repair and evaluation.

For a defect, fault localization will find the fault statement in the buggy program. In the patch generation step, the repair tool replaces fault statements with repair ingredients to generate patches; the syntax similarity between the fault statement and the repair statement is calculated. The patch qualify evaluation evaluates the qualify of patches through patterns obtained by the program defect detection tools. The patches are prioritized according to this value of the syntax similarity. The details are as follows.

(1) **Fault Localization:**It finds the buggy node in the abstract syntax tree of the original program. Similar to the research by Chen and Monperrus [33], we assume that the fault location identifies the correct fault statement because the main concern is the patch prioritization. For a given defect, the fault statement is determined based on the difference

between the original buggy program and the correct patch. Next, the fault expression is extracted from the fault statement. Figure 2 shows the defect Math#80 from project Defects4J. So, line 1135 is the location of the fault statement, the fault expressions from the fault statement, such as $4 * n - 1$, are extracted.

(2) **Patch Generation:** It replaces the fault ingredient with repair ingredients in the original buggy program to generate patches or insert repair ingredients to generate patches. After identifying the fault statement, we use repair expressions obtained in the original buggy program as repair ingredients. Next, the fault expression is replaced by the repair ingredient to generate patches. For example, a patch replaces $4 * n - 1$ with $4 * (n - 1)$ from line 1132 in Figure 2.

(3) **Patch Quality Evaluation:** We apply the static analysis tool DTSJava [10] to detect semantic and syntax defects of the repaired programs, to ensure semantic correctness of the patch and maintainability of the repaired program. A patch with a semantic defect is given in Figure 3. The code annotation indicates the value range of the variables in the program, analyzed by DTSJava. The value range of the variable i is from -2147483648 to +2147483647, the upper bound of the array buff is 10. This patch still has an array out-of-bounds defect because the value of expression i can be greater than 10. We recommend patches that can meet the pattern obtained by DTSJava to developers and discard patches that do not meet the pattern obtained by DTSJava.

(4) **Patch Prioritization:** This step of the patch generation generates a large number of patches, which contain many incorrect patches. To early find the correct patch from all patches, the patches generated by repair tools are prioritized. Both syntax similarities and symmetry between the fault statement and repair ingredient are used to prioritize patches. The four features—variable, expression, structure, and repair location—are used to calculate the syntax distance, which can effectively distinguish between correct patches and incorrect patches. Finally, the patches are prioritized according to the syntax distance.

```
1132  double d= work[4*(n-1)+pingPong];   //repair ingredient
1135  int j=4*n-1;                        //buggy statement
1135  int j=4*(n-1);                      //repaired statement
```

Figure 2. Buggy statement, repaired statement, and repair ingredient of the defect Math#80 in Defects4J.

```
  void example(){
  int buffer[10];     //buffer:Array{[10,10]}
     int i = 0;       //i:[0,0]
     int y = input();//y:[-2147483648, +2147483647]
     while(i<y){      //i:[-2147483648, +2147483647]
   -  buffer[i]=i;    //buffer:Array{[10,10]}
   +  buffer[i-1]=i; //buffer:Array{[10,10]}
      i++;            //i: [-2147483648, +2147483647]
      }
    }
```

Figure 3. A patch with a semantic defect.

3.1. Variable Similarity

Variables are the main components of an expression and the differences in statements can be captured through variables and constants. Variables between the correct repaired code and buggy code are very similar [12,34], and the variable similarity indicates the difference of variables and constants between the repaired code and buggy code. When a defect has multiple repair codes, the smaller the difference between the repaired code and the buggy code, the more likely the repaired code will become the correct repaired code. To calculate the variable similarity, we use e to represent the buggy expression and p to represent the repaired expression of a patch. We extract the variables and constants from the original expression e and put them into the set V_e. At the same time, we obtain the variables and constants in the repaired expression p and put them into the set V_p. For this feature, the higher the variable similarity, the better the quality of the patch. Let L^{var} represent the syntax distance between the patch and the buggy program statement on the variable similarity and let P^{var} indicate the recommended value of the patch in terms of variable similarity. The calculations of the two are shown in Formulas (1) and (2), respectively.

$$L^{var} = 1 - \frac{|V_e \cap V_p|}{|V_e \cup V_p|} \tag{1}$$

$$P^{var} = \frac{1}{1 + L^{var}} \tag{2}$$

Example 1. *Figure 4a shows the program that was repaired by the correct patch generated manually; the variable sets of the defect and the repair expression are V_e = { x , prev, 0 , '-' } and V_p={ x , prev , negativeZero , 0 , '-' }, respectively. The variable sets of the defect and the repair expression in Figure 4b are V_e = { x , prev , 0 , '-' } and V_p={ prev , '-' }, respectively. We can use the above formula to calculate the patch recommendation values in Figure 4a,b, respectively:*

$$L^{var}_{human} = 0.2, \qquad L^{var}_{Nopol} = 0.5,$$
$$P^{var}_{human} \approx 0.83, \qquad P^{var}_{Nopol} \approx 0.67,$$

We can see that the recommended value of the correct patch manually repaired is significantly higher than the incorrect patch generated by Nopol.

```
char prev = getLastChar();
boolean negativeZero = isNegativeZero(x);
+if ((x<0||negativeZero)&&prev=='-'){
-if (x<0 && prev=='-') {
  add(" ");
 }
...
```

(**a**) Correct human patch

```
char prev = getLastChar();
boolean negativeZero = isNegativeZero(x);
+if (prev=='-') {
-if (x<0 && prev=='-') {
  add(" ");
}
...
```

(**b**) Incorrect patch generated by Nopol

Figure 4. Patches of Closure#38 in Defects4J.

3.2. Expression Similarity

Expressions are the main components of a statement; the functional differences of statements can be captured through expressions. For a program, the same function is implemented in slightly different ways in multiple locations; defects can be repaired by using repair ingredients obtained from the same project [20]. The repair ingredients obtained in the source file are better than those synthesized by repair tools. We call the expression containing the similar repair ingredient of the patch as the contribution expression; the contribution expression is the expression obtained in the source file and contains the repair ingredient of the patch. First, we search for a contribution expression similar to the given repaired expression by utilizing a clone detection technique [28] and obtaining the contribution expression of the most similar repaired expression. To find the contribution expression, we transformed each expression into token sequences and compared token sequences between expressions and the given repaired expression using the longest common subsequence (LCS). Then, we regarded the expression with the highest LCS value as the contribution expression. The expression similarity indicates the function difference between the repair expression and the contribution expression. For this feature, it is obvious that the higher the expression similarity, the better the quality of the patch. Let L^{exp} represent the syntax distance between the repair expression of the patch and the contribution expression on the expression similarity and P^{exp} indicate the recommended value of the patch in terms of expression similarity. We used *ex* to represent the repair expression of the patch, and *sc* to represent the contribution expression. The calculations of the two are shown in Formulas (3) and (4), respectively.

$$L^{exp} = \frac{|LCS(ex, sc)|}{min(|ex|, |sc|)} \tag{3}$$

$$P^{exp} = L^{exp} \tag{4}$$

where, *LCS*(*ex*,*sc*) represents the common sub-sequence of the maximum length between *ex* and *sc*, *min*(| *ex* | , | *sc* |) represents the minimum length between *ex* and *sc*.

Example 2. *Figure 5a shows the program that was generated by CapGen, the repair expression ex is* $4 * (n - 1)$*, and the contribution expression sc is* $4 * (n - 1) + pingPong$ *in the first line, so the common sub-sequence of the maximum length between the repair expression ex and the contribution expression sc is 7. Figure 5b shows the incorrect patch generated by CapGen, the repair expression ex is* $4 * pingPong - 1$*, and the contribution expression sc* $4 * n - 3 - pingPong$ *in the source file, so the common sub-sequence of the maximum length between the repair expression ex and the contribution expression sc is 8. We can use the above formula to calculate the patch recommendation values in Figure 5a,b, respectively:*

$$L^{exp}_{correct} = \frac{7}{7} = 1, \qquad L^{exp}_{incorrect} = \frac{8}{12} = 0.67,$$

$$P^{exp}_{correct} = 1, \qquad P^{exp}_{incorrect} \approx 0.67,$$

```
double d = work[4*(n-1)+pingPong];
- int j = 4*n-1;
+ int j = 4*(n-1);
...
```

(**a**) Correct patch generated by CapGen.

```
- int j = 4*n-1;
+ int j = 4*pingPong-1;
...
work[4*n-3-pingPong] = d;
```

(**b**) Incorrect patch generated by CapGen.

Figure 5. Patches of math#80 in Defects4J.

We can see that the recommended value for the correct patch generated by CapGen is significantly higher than the incorrect patch generated by CapGen.

3.3. Structure Similarity

Structure similarity concerns the difference in the control structure between the repaired code and the buggy code. The statement containing the contribution expression is the contribution statement; the contribution statement is the statement obtained in the source file containing the repair ingredient. In practice, we found that the control structure between the contribution statement and buggy statement was very similar. For this feature, it can be considered that the higher the structural similarity, the better the quality of the patch. We extracted a vector $\boldsymbol{V}(k_1, k_2, k_3)$ from the code, where k_1, k_2, k_3 is the loop, judgment, and return numbers, respectively. The vector of the buggy statement is $V'(k'_1$, k'_2, $k'_3)$, and the vector of the contribution statement is $\boldsymbol{V}(k_1$, k_2, $k_3)$; then we calculated the structural similarity between the two vectors by using Euclidean distance. Let L^{stu} represent the syntax distance between the contribution statement and the buggy statement in the structural similarity and let P^{stu} indicate the recommended value of the patch in terms of structural similarity. The calculations of the two are shown in Formulas (5) and (6), respectively.

$$L^{stu} = \sqrt{\sum_{x=1}^{3}(k'_x - k_x)^2} \tag{5}$$

$$P^{stu} = \frac{1}{1 + L^{stu}} \tag{6}$$

Example 3. *Figure 6a shows the correct patch generated by CapGen, the buggy statement is* ***if****(dataset != **null**), containing an **if**, the vector of the buggy statement is (0, 1, 0), the contribution statement is **if**(dataset == **null**) in the ninth line, containing an **if**, the vector of the contribution statement is (0, 1, 0). Figure 6b shows the incorrect patch generated by jKali, the buggy statement is **if** (dataset != **null**), the vector of the buggy statement is (0, 1, 0), the contribution statement is "return false;" in the ninth line, containing a **return**, the vector of the contribution statement (0, 0, 1).*

$$L^{stu}_{CapGen} = \sqrt{(0-0)^2 + (1-1)^2 + (0-0)^2} = 0, \qquad P^{stu}_{CapGen} = 1,$$

$$L^{stu}_{jKali} = \sqrt{(0-0)^2 + (0-1)^2 + (1-1)^2} = 1, \qquad P^{stu}_{jKali} = 0.5,$$

```
...
918   if (dataset == null) {
...
1794  CategoryDataset dataset =
          this.plot.getDataset(index);
    +   if (dataset == null) {
1795 -  if (dataset != null) {
1796          return result;
          }
...
```

(**a**) Correct patch generated by CapGen.

```
...
1794  CategoryDataset dataset =
          this.plot.getDataset(index);
    +     if (false) {
1795 -    if (dataset != null) {
              return result;
          }
...
1990  return false;
...
```

(**b**) Incorrect patch generated by jKali.

Figure 6. Patches of Chart#1 in Defects4J.

We can see that the recommended value of the correct patch generated by CapGen is significantly higher than the incorrect patch generated by jKali.

3.4. Repair Location

Defects will be activated and program faults will occur only when certain conditions are met. However, the continuous backward transmission of defects will cause program failure. The repair location affects the quality of a patch to some extent; only the correct repair location can eliminate program defects, while the error repair location leaves more defects in the error propagation chain, which may lead to the generation of other defects.

The error propagation chain $Z(l, f, N)$ of a program is obtained by dynamic slicing, which represents the propagation chain of the sequence N from the defect location l to the failure location f. The patch quality is judged by the repair location d of the patch in the error propagation chain, the higher the repair location is in the error propagation chain, the larger the syntax distance between the original program and the repaired program. The further you go, the more defects you can repair, and the greater the difference from the original buggy program. For this feature, the earlier the patch is in the error propagation chain, the better the quality of the patch is; otherwise, the worse the quality of the patch is. In other words, the greater the syntax distance between the repaired program and the original buggy program, the better the patch quality.

Let $[l, d]$ be the dynamic execution path from defect location l to repair location d, and $|[l, d]|$ be the statement number of the execution path from the defect location l to repair the location d; Let $[l, f]$ be the dynamic execution path from the defect location l to the failure location f, and $|[l, f]|$ be the statement number of execution paths from defect locations l to the failure location f. Let L^{loc} represent the syntax distance between the patch and the original program in the feature of the repair location and P^{loc} represent the recommended value of the patch in the feature of the repair location. The calculations are shown in Formulas (7) and (8).

$$L^{loc} = \begin{cases} 0, & if\, d \notin N \\ 1 - \frac{|[l,d]|}{|[l,f]|}, & if\, d \in N \end{cases} \tag{7}$$

$$P^{loc} = L^{loc} \tag{8}$$

Example 4. *Figure 7a shows a buggy program, when index >= 0 && allowDuplicateXValues = true and autoSort = true, the defect will activate in the second line of Figure 7a, and the array will be out of bounds in the sixth line of Figure 7a. We can obtain the error propagation chain [1, 4, 5], |[l, f]| = |[1, 5]| = 2. Figure 7b shows the correct patch generated manually, the repair location d is in the third line of Figure 7b, it is the correct repair location, |[l, d]| = |[1, 1]| = 0. Figure 7c shows the incorrect patch generated by DynaMoth, the repair location d is the third line of Figure 7c, it is the incorrect repair location, |[l, d]|=|[1, 4]| = 1. Applying Formulas (7) and (8), we can obtain:*

$$P^{loc}_{human} = 1 - \frac{0}{2} = 1, \qquad P^{loc}_{DynaMoth} = 1 - \frac{1}{2} = 0.5,$$

```
...
if (index >= 0&&!allowDuplicateXValues) {
...
}else {
    if (autoSort) {
        data.add(-index - 1, new XYDataItem(x, y));
    }
}
...
```

(**a**) The buggy program.

```
...
-if (index>=0&&!allowDuplicateXValues){
+if (index >= 0) {
...
```

(**b**) Correct human patch.

```
...
-if (autoSort) {
+if (equals((Object) x)) {
...
```

(**c**) Incorrect patch generated by DynaMoth.

Figure 7. Patches of Chart#5 in Defects4J.

We can see that the recommended values of the manual patches are significantly higher than the incorrect patch generations by DynaMoth; DynaMoth generates patches but introduces new functional defects.

3.5. The Recommendation Value of the Joint Feature

We integrated four syntax features—variable, expression, structure, and repair location—to obtain the syntax distance between the buggy program and the patch program. Then the patch quality was evaluated according to the syntax distance. Four features made similar contributions to the joint feature evaluation, and a patch with higher syntax distance was ranked higher. For the three operations of modification, insertion, and deletion, we treated them differently according to their different changes to the program.

For modification, we combined all four features; thus, the recommendation value of the patch P is specifically defined as:

$$P = P^{var} \times P^{exp} \times P^{stu} \times P^{loc} \tag{9}$$

However, for insertion, the patch is inserted into the program, and there are no buggy statements, so there is no need to consider the variable feathers. Therefore, for insertion, the recommendation value of the patch P is defined as follows:

$$P = P^{exp} \times P^{stu} \times P^{loc} \tag{10}$$

For deletion, the case is more complicated since we are only going to delete the statement in the buggy program. In this case, we first locate the buggy line and compute the recommendation value of the repair location. Then we compute the variable similarities between the deleted statement and the defect statement. It makes no sense to include the expression model and structure model in this case since the statement is only deleted. Therefore, for deletion, the recommendation value of the patch P is defined as follows:

$$P = P^{var} \times P^{loc} \tag{11}$$

Example 5. *Figure 5a shows the correct patch that is generated by CapGen. The variable sets of the defect and the repair expression are both V_e = { n, 1, 4 }, so the syntax distance of the variable similarity is 1. Example 2 shows that the syntax distance of the expression similarity is 1. In Figure 5a, the buggy statement is in line 2 and the contribution statement is in line 1. The buggy statement and the contribution statement do not contain any control structures, so the vectors of the buggy statement are both (0, 0, 0). Therefore, the syntax distance of the structural similarity is 1. The defect is activated in line 2 and the repair location is in line 3, so the syntax distance of the repair location is 1. Figure 5b shows the incorrect patch generated by CapGen. The variable sets of the defect and the repair expression are V_e = { n, 1, 4 } and V_p = { pingPong, 1, 4, } respectively, so the syntax distance of the variable similarity is 0.67. Example 2 shows the syntax distance of expression similarity is 0.67. In Figure 5b, the buggy statement is in line 1 and the contribution statement is in line 4. The buggy statement and the contribution statement do not contain any control structures, so the syntax distance of the structural similarity is 1. The defect is activated in line 1 and the repair location is in line 2, so the syntax distance of the repair location is 1. We can use the above Formula (9) to calculate the patch recommendation values in Figure 5a,b, respectively:*

$$P_{\text{Correct}} = P^{var} \times P^{exp} \times P^{stu} \times P^{loc} = 1 \tag{12}$$

$$P_{\text{Incorrect}} = P^{var} \times P^{exp} \times P^{stu} \times P^{loc} \approx 0.45 \tag{13}$$

We can see that the syntax distance value of the correct patch generated by CapGen is significantly higher than the incorrect patch generated by CapGen.

4. Results

To verify the effectiveness of our evaluation method, Nopol, DynaMoth, SimFix, jKali, ACS, jGenProg, kPar, TBar, ARJA, AVARTAR, jMutRepair, Cardumen, FixMiner, and CapGen were selected as the analysis objects, and Defects4J [10] was chosen as the benchmark. Among these tools, Nopol is a repair tool that synthesizes constraints based on semantic information and generates conditions and loop statements, DynaMoth is a repair tool that can further synthesize method call conditions on the basis of Nopol. SimFix is a repair tool for generating patches by searching similar codes in the same project through code structure and semantic features. jKali is a Java language implementation of Kali that only deletes functionalities. CapGen is a context-aware repair tool at the level of fine-grained abstract syntax tree nodes. In this paper, the repair tools selected cover two patch synthesis methods: ACS, Nopol, and DynaMoth cover semantic-based patch synthesis methods; TBar, FixMiner, kPar, jGenProg, ARJA, AVARTAR, jMutRepair, Cardumen, jKali, SimFix, and CapGen cover patch synthesis methods based on generation and verification. The plausible and correct patch numbers of the multiple tools on Defects4J are shown in Table 3.

Table 3. The plausible and correct patch numbers in the multiple tools.

Project	NO	DY	SI	JK	CA	JG	TB	KP	AC	AR	AV	FI	CR	JM
Chart	1/6	1/6	4/9	0/6	5/66	0/5	7/17	3/13	2/2	1/10	5/12	5/12	2/4	1/4
Lang	3/7	1/6	5/16	0/0	9/29	0/2	6/21	1/18	3/3	0/2	4/13	0/2	0/0	0/2
Math	1/21	1/28	12/28	1/10	14/152	3/11	8/22	4/21	12/17	3/13	3/17	7/14	1/6	2/11
Time	0/1	0/5	1/2	0/2	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0/1	0/0	0/0
Total	5/35	2/50	22/55	1/18	28/247	3/18	21/60	8/52	17/22	4/25	12/42	12/29	3/10	3/17
Precision	14%	4%	40%	5%	11%	17%	35%	15%	77%	16%	29%	41%	30%	18%

A/B: A represents the number of correct patches, B represents the number of plausible patches. AC represents ACS, AR presents ARJA, AV presents AVATAR, KP presents kPar, TB represents TBar, DY presents DynaMoth, JK presents jKali, NO presents Nopol, SI presents SimFix, JG presents jGenProg, CA represents CapGen, FI presents FixMiner, CR presents Cardumem, JM represents jMutRepair.

4.1. Ability to Evaluate Real Patches

Since a repair tool does not generate plausible patches for all defects or only generates a single plausible patch, we only selected the defects with multiple plausible patches as the dataset. Then, we used multiple repair tools to repair defects separately and obtained all plausible patches that could repair the defect. We evaluated them through our evaluation method and obtained the result shown in Table 4. In Table 4, 95% of the correct patches were ranked in the top one and all of the correct patches were ranked in the top two by our evaluation method, which means that our evaluation method does have an obvious evaluation effect.

4.2. Ability to Evaluate Patch Quality by Different Features

To judge the contribution of each individual evaluation feature, we separately studied the evaluation ability of each feature for the same patches. Here, the patches we evaluated were the plausible patches in Table 4, and the recommended values of the patches were obtained by evaluating the 103 correct patches and 265 incorrect patches, respectively, with four evaluation features. If a feature had a better evaluation ability for the patches, it had a higher recommendation for correct patches and a lower recommendation for incorrect patches. Moreover, if the recommended value of this feature for the correct patch was higher than other features, it means that this feature made a greater contribution to the joint feature evaluation.

Figure 8 shows the effectiveness evaluations of the variable, expression, structure, and repair locations, respectively, and with average values 0.82, 0.75, 0.96, and 0.96. Similarly, we can find that the recommended values of the correct patch and the incorrect patch under these features are quite different, which means that the four features can well distinguish the correct patches. The recommended values of the four features for the correct patches are similar, which means that the four features made similar contributions to the joint feature evaluation.

Table 4. Effectiveness evaluations of the patches generated by multiple repair tools.

BugID	Rank	Rv	AC	AR	AV	KP	TB	DY	JK	NO	SI	JG	CA	FI	CR	JM
Chart1	1	0.38	-	0/1	1/1	1/1	1/1	0/1	0/1	-	1/1	-	1/1	1/1	-	1/1
Chart4	1	1.0	-	-	1/1	1/1	1/1	-	-	-	-	-	-	1/1	-	-
Chart7	2	0.33	-	0/1	0/1	0/1	0/1	-	-	-	1/1	0/1	-	0/1	-	0/1
Chart8	1	0.75	-	-	-	-	-	-	-	-	-	-	1/62	-	-	-
Chart9	1	0.67	-	-	-	-	1/1	-	-	0/1	-	-	-	-	-	-
Chart11	1	0.67	-	-	1/1	-	1/1	-	-	-	-	-	1/1	1/1	1/1	-
Chart12	1	1.0	-	1/1	-	0/1	0/1	-	-	-	0/1	-	-	0/1	-	-
Chart14	1	1.0	1/1	0/1	0/1	0/1	0/1	0/1	-	-	-	-	-	0/1	-	-
Chart19	1	1.0	1/1	0/1	1/1	-	1/1	-	-	-	-	-	-	1/1	-	-
Chart20	1	0.42	-	-	-	-	1/1	-	-	-	1/1	-	-	-	-	-
Chart24	1	0.71	-	-	1/1	-	1/1	-	-	-	-	-	1/1	1/1	1/1	-
Math4	1	1.0	1/1	-	1/1	-	-	-	-	0/1	-	-	-	-	-	-
Math5	1	0.5	1/1	-	-	-	1/1	-	-	-	-	-	1/4	-	-	-
Math30	1	0.33	-	-	-	-	-	-	-	-	-	-	1/1	1/1	-	-
Math33	1	0.8	-	-	-	-	-	-	-	0/1	-	-	1/1	-	-	-
Math35	1	1.0	1/1	1/1	-	-	-	-	-	-	-	-	-	-	-	-
Math41	1	1.0	-	-	-	-	-	-	0/1	0/1	1/1	-	-	-	-	-
Math50	2	0.5	-	0/1	0/1	0/1	0/1	1/1	1/1	-	-	0/1	-	0/1	-	0/1
Math53	1	1.0	-	-	-	-	-	-	-	-	1/1	1/1	2/2	-	-	-
Math57	1	1.0	-	-	0/1	-	1/1	-	-	-	1/1	-	1/1	1/1	-	0/1
Math58	1	1.0	-	1/1	-	1/1	-	0/1	-	-	-	-	1/1	1/1	1/1	-
Math59	1	1.0	-	-	1/1	-	-	-	-	-	1/1	-	1/1	-	-	-
Math63	1	0.67	-	-	-	0/1	0/1	-	-	-	0/1	-	1/9	0/1	0/1	-
Math65	1	1.0	-	-	-	-	1/1	-	-	-	-	-	1/1	-	-	-
Math70	1	0.67	-	-	-	1/1	1/1	-	-	-	1/1	1/1	1/1	-	-	-
Math71	1	0.67	-	-	-	-	-	0/1	-	-	1/1	-	-	-	-	-
Math75	1	1.0	-	-	-	1/1	1/1	-	-	1/1	1/1	1/1	1/1	1/1	-	
Math79	1	1.0	-	-	1/1	-	1/1	-	-	-	-	-	1/1	1/1	1/1	-
Math80	1	1.0	-	0/1	0/1	0/1	0/1	0/1	-	0/1	0/1	0/1	1/125	0/1	-	1/1
Math85	1	0.41	1/1	0/1	-	-	0/1	0/1	-	-	0/1	0/1	1/4	0/1	0/1	1/1
Math89	1	0.15	1/1	-	1/1	1/1	1/1	-	-	-	-	-	-	-	-	-
Lang6	1	0.38	-	-	1/1	-	-	-	-	-	-	-	1/1	-	-	-
Lang7	1	1.0	-	-	1/1	-	1/1	-	-	-	-	-	1/1	1/1	1/1	-
Lang10	1	0.5	-	-	0/1	0/1	1/1	-	1/1	-	0/1	-	-	-	-	-
Lang24	1	1.0	1/1	-	-	0/1	0/1	-	-	-	-	-	-	-	-	-
Lang33	1	1.0	-	-	-	-	1/1	-	-	-	1/1	-	-	-	-	-
Lang39	1	1.0	-	-	0/1	-	0/1	-	-	-	1/1	-	-	-	-	-
Lang43	1	1.0	-	-	-	0/1	0/1	-	-	-	-	1/1	-	-	-	-
Lang57	1	0.33	-	-	1/1	0/1	1/1	-	-	-	-	-	3/3	-	-	1/1
Lang58	1	0.5	-	-	0/1	0/1	0/1	-	-	-	-	-	0/1	1/1	-	-
Lang59	1	0.8	-	-	0/1	0/1	1/1	-	-	-	-	-	1/20	-	-	1/1
Lang60	1	0.75	-	-	-	-	0/1	-	-	-	1/1	-	-	-	1/1	-

A/B: A represents the number of correct patches, B represents the number of plausible patches, RV represents the recommendation value of the correct patch, "-" indicates that there is no plausible or correct patch for the defect and it cannot be sorted. AC represents ACS, AR presents ARJA, AV presents AVATAR, KP presents kPar, TB represents TBar, DY presents DynaMoth, JK presents jKali, NO presents Nopol, SI presents SimFix, JG presents jGenProg, CA represents CapGen, FI presents FixMiner, CR presents Cardumem, JM represents jMutRepair.

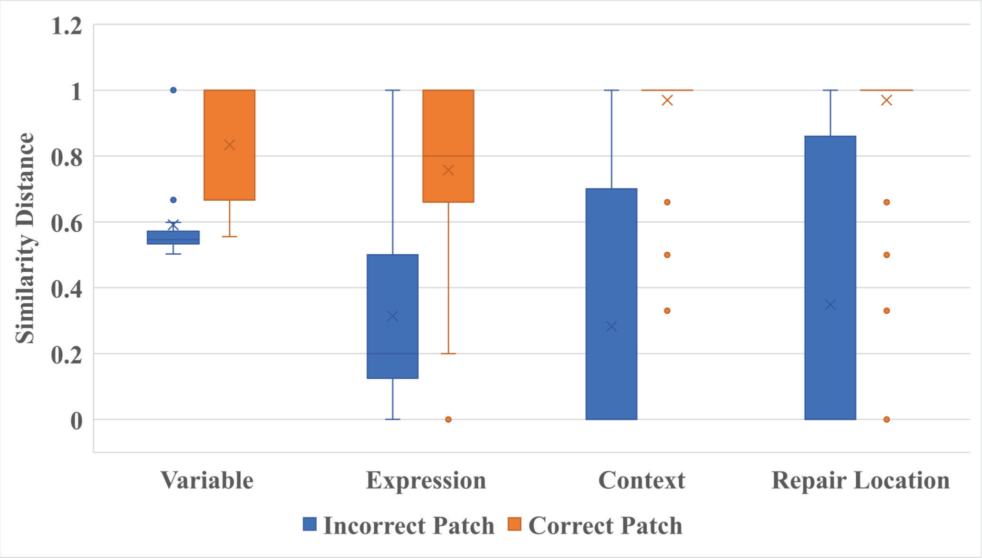

Figure 8. Effectiveness evaluations of different features on the patches.

4.3. Comparison with Existing Methods

At present, only CapGen can generate multiple patches for defects with a patch-sorting mechanism. SSDM is the patch quality evaluation tool that uses path matching, output coverage, and internal distance to evaluate the patches. We selected CapGen and SSDM to compare with our evaluation method (PPSD); the evaluation data are plausible patches in Table 4. There were 43 defects in the dataset in Table 4. CapGen ranked the correct patches of the 36 defects first. PPSD ranked the correct patches of the 41 defects first. SSDM ranked the correct patches of the 36 defects first. In the PPSD, the number of defects with correct patches ranked first was more than that of CapGen and SSDM. The comparison results of CanGen, SSDM, and PPSD are shown in Table 5. In Table 5, we only show that the defects of CapGen, SSDM, and PPSD cannot rank the correct patch first. To save space in Table 5, we do not show that the defects of CapGen, SSDM, and PPSD can rank the correct patches first. There are 36 defects not shown in Table 5.

We used the CapGen to evaluate the plausible patches in Table 4. Most of the correct patches were evenly ranked in the top one. At the same time, it can be seen that the correct patch for Math 80 was ranked in third place. The incorrect expression is $4 * n - 1$ in Figure 5, which affects the correctness of the program. CapGen applies frequency and context similarity of the variable and the expression to prioritize patches. The frequency of the variable *pingPong* is much higher than that of the expression $4 * (n - 1)$ in the buggy file and there is little difference in context similarity between variables *pingPong* and expressions $4 * (n - 1)$, so the correct patch for Math 80 is ranked in third place. CapGen does not consider the impact of expression similarity and structural similarity on patch quality. Many repairing expressions can be synthesized by CapGen, which does not contain the functionality that the developer wants; $4 * n - 1$ is replaced by $4 * pingPong - 1$ that does not appear in the program, which does not contain the functionality that the developer wants; the expression similarity is 1. The expression similarity, structural similarity, variable similarity, and recommendations of the repair locations are 0.67, 1.0, 0.67, and 1.0 respectively, so the recommendation value of the incorrect patch is 0.45. Moreover, $4 * n - 1$ is replaced by $4 * (n - 1)$, which appears in the program and contains the functionality that the developer wants; the expression similarity of the patch is 1.0. The expression similarity, structural similarity, variable similarity, and recommendation values of the repair locations are 1.0, 1.0, 1.0, and 1.0, so the recommendation value of the correct patch is 1.0. Therefore, our method ranks the patch first, but this correct patch cannot be judged as a high-quality patch by CapGen.

We used the SSDM to evaluate the plausible patches in Table 4, it can be seen that most of the correct patches were evenly ranked in the top one. At the same time, it can be seen that the correct patch for Math 63 was ranked in third place. In Figure 9, there is only one return statement in the "*equals*" function, which affects the correctness of the program. Since the patch does not affect the number of method calls, the execution path in the "*equals*" method is the only one that needs to be judged, and the method contains only a return statement, which makes it unable to reflect obvious differences in the path-matching feature. SSDM does not consider the impact of expression similarity, structural similarity, variable similarity, and recommendation of the repair location on the patch quality. The expression (Double.isNaN(x) && Double.isNaN(y)) | | x == y is replaced by the expression equaling (x, y, 1), which appears in the program and contains the functionality that the developer wants; the expression similarity of the patch is 1.0, the structural similarity is 1.0, the variable similarity is 0.75, and the recommendation value of the repair location is 1, so the recommendation value of the correct patch is 0.75. The expression (Double.isNaN(x) && Double.isNaN(y)) is replaced by the expression x == y | | x == y that does not appear in other locations of the buggy file, which does not contain the functionality that the developer wants; the expression similarity is 0.3, the structural similarity is 1.0, the variable similarity is 1.0, and the recommendation value of the repair location is 1.0. The recommendation value of the incorrect correct patch is 0.3. Therefore, our method ranks the patch first; the correct patch cannot be judged as a high-quality patch by the SSDM. We combine SSDM with our proposed method to achieve a better effect of the patch quality evaluation.

Table 5. Comparison of the evaluation effect of CapGen, SSDM, and PPSD in the patches generated by multiple repair tools.

Bug ID	Plausible	CapGen Rank	PPSD Rank	SSDM Rank
Chart1	10	1	1	2
Chart7	8	2	2	2
Chart14	7	4	1	7
Chart19	5	2	1	2
Math50	9	2	2	3
Math63	14	1	1	3
Math73	3	3	1	2
Math80	135	3	1	1
Math82	4	2	1	1

Plausible represents the correct and incorrect patch numbers for the defect, CapGen Rank presents the rank of the correct patch given by CapGen, SSDM Rank presents the rank of the correct patch given by SSDM, and PPSD Rank represents the rank of the correct patch given by our method.

To prove that our method is superior to similar methods, the following evaluation metrics were inspected:

(1) **Average Space Reduction**: It presents how much patches can be reduced to find the correct patch from all patches [33].

$$Avg\ Space\ Reduction = (1 - \frac{Mean\_Correct}{Mean\_Total}) \times 100\% \tag{14}$$

where *Mean_Correct* presents the mean of the correct patch rank, and *Mean_Total* represents the mean number of all patches generated by the repair tools.

(2) **Perfect Repair**: It represents the percentage of defects that the correct patch ranks first [33].

The SSDM uses four semantic features, including interval distance, output coverage, and path matching to evaluate the patches. SSDM can evaluate patches that modify the execution path of the original program. However, SSDM cannot evaluate patches that do not modify the execution path of the original program and modify the expression in a single statement. Our method uses four syntax features (variable, expression, structure, and repair location) to evaluate the patches, so it can evaluate patches at the fine-grained level. Our method can evaluate patches that SSDM cannot evaluate. The CapGen sorts patches based on genealogy structure, access variables, and semantic dependencies. However, CapGen cannot evaluate patches of semantic defects. Our method integrates static analysis techniques with test suites to strengthen specifications, so it can effectively evaluate the quality of patches of semantic defects. Figure 10 shows that our method is effective in average space reduction, compared to CapGen and SSDM. For our method, the search space can be reduced, while for the average space reduction of our method, SSDM and CapGen were 87.7%, 84.0%, and 85.0%, respectively. In terms of perfect repair, our method outperformed CapGen and SSDM, the values were 95.3%, 83.7%, and 83.7%, respectively. The comparisons reveal that our method is better than SSDM and CapGen.

```
public static boolean equals(double x, double y) {
 -return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
 +return equals(x, y, 1);
}
 ...
 return equals(x, y, 1);//contribution expression
```

(**a**) Correct Patch

```
public static boolean equals(double x, double y) {
 -return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
 +return x == y || x == y;
}
...
return !((x == null) && (y == null));//contribution expression
```

(**b**) Plausible Patch

Figure 9. Patches for Math #63 in Defects4j generated by CapGen.

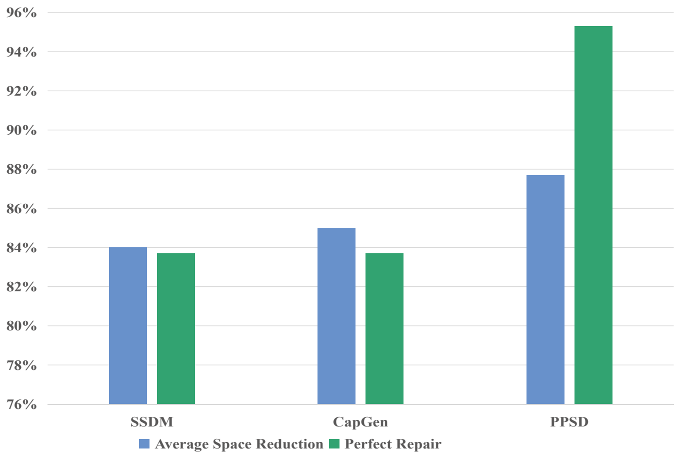

Figure 10. Comparison of SSDM, CapGen, and PPSD.

5. Threats to Validity

Threats to External Validity. The external threat is that the tools we use are all repaired in the Defect4j benchmark pool to generate patches, which leads us to only evaluate the quality of the patches generated by defects in Defect4j. This implies that our evaluation method may not be extended to other datasets. Therefore, we attempted to use the IntroClassJava [35] as the benchmark pool, which is the Java version of the IntroClass [36]. First, we used the above multiple repair tools to generate patches for the defects in the IntroClassJava benchmark pool. Then, we used these patches to prove whether our evaluation method could be extended to other datasets.

Threats to Internal Validity. This paper assumes that the fault location identifies the correct fault statement, but this assumption may not always be true [26]. However, the main concern is the patch prioritization. The study of fault localization technology is beyond the scope of our study. Similarly, verifying the correctness of the patch is a study in itself, which was explored by Xin et al. [24].

6. Discussion and Limitations

According to the existing automatic repair research, the test suites are not perfect ways to guarantee the correctness of patches generated by repair tools and weak test suites lead to a large number of incorrect patches. We integrated static analysis techniques with test suites to strengthen specifications, which can effectively find the correct patches of semantic defects. For non-semantic defects, we propose a patch quality evaluation method, which measures the syntax distances of patches through four evaluation features—variable, expression, structure, and repair location. It evaluates all patches of defects and gives the sorted patch set to developers. Our method can reduce the time for developers to find the correct patch. It improves the efficiency of developers. Our method can be incorporated into the field of continuous integration [37,38]. In the continuous integration phase, our method can be used as the last step of code quality verification. Our method can verify the quality of the code after the repaired defects. Our method can reduce the risk of introducing semantic defects. At the same time, our method can help developers quickly find the correct repaired code to achieve rapid continuous integration.

However, our method has some shortcomings. It evaluates the patch quality at the fine-grained level, such as the expression control structure and variables, without considering the dependence between statements, so our method can effectively evaluate patches of the defect involving the single statement but it cannot effectively evaluate patches of defects involving multiple statements.

7. Conclusions

This paper proposes a patch quality evaluation method based on syntax distance metrics. Variable, expression, structure, and repair location features are used to measure the syntax distances of patches. Meanwhile, this method integrates static analysis techniques with test suites to strengthen specifications. This method was verified on 368 patches generated by existing repair tools, and it successfully ranked 95% of the correct patches in the top one. This shows that it is very effective for judging the quality of the patch generated by the repair tool. Our method does not consider the dependence between statements, so it cannot effectively evaluate patches of defects involving multiple statements. In the future, we hope that our method can be combined with the dynamic test method, as well as further expand the existing work.

Author Contributions: Conceptualization, D.T. and Y.D.; methodology, D.T. and Y.D.; software, D.T. and Y.D.; validation, D.T., X.C. and Y.D.; formal analysis, D.T. and Y.D.; investigation, D.T. and Y.D.; resources, D.T., Y.Y. and Y.D.; data curation, D.T., Y.Y. and Y.D.; writing—original draft preparation, D.T. and Y.D.; writing—review and editing, D.T. and Y.D.; visualization, D.T. and Y.D.; supervision, Y.D.; project administration, Y.D.; funding acquisition, Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Natural Foundation of Shandong Province under grant ZR2021MF058, Fundamental Research Funds for the Central Universities (20CX05016A), and the Major Scientific and Technological Projects of CNPC under grant ZD2019-183-007.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Nilizadeh, A. Automated Program Repair and Test Overfitting: Measurements and Approaches using Formal Methods. In Proceedings of the 2022 IEEE Conference on Software Testing, Verification and Validation, Valencia, Spain, 4–14 April 2022.
2. Gao, X.; Wang, B.; Duck, G.J.; Ji, R.; Xiong, Y.; Roychoudhury, A. Beyond tests:Program Vulnerability Repair via Crash Constraint Extraction. In *ACM Transactions on Software Engineering and Methodology*; Association for Computing Machinery: New York, NY, USA, 2021.

3. Yi, J.; Ismayilzada, E. Speeding up constraint-based program repair using a search-based technique. *Inf. Softw. Technol.* **2022**, *146*, 106865. [CrossRef]
4. Xiong, Y.; Liu, X.; Zeng, M.; Lu, Z.; Gang, H. Identifying patch correctness in test-based program repair. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018.
5. Xu, T.; Chen, L.; Pei, Y.; Zhang, T.; Pan, M.; Furia, C.A. Restore: Retrospective Fault Localization Enhancing Automated Program Repair. *IEEE Trans. Softw. Eng.* **2022**, *48*, 309–326. [CrossRef]
6. Cao, H.; Liu, F.; Shi, J.; Chu, Y.; Deng, M. Automated Repair of Java Programs with Random Search via Code Similarity. In Proceedings of the IEEE 21st International Conference on Software Quality, Reliability and Security Companion, Haikou, China, 6–10 December 2021.
7. Kechagia, M.; Mechtaev, S.; Sarro, F.; Harman, M. Evaluating Automatic Program Repair Capabilities to Repair API Misuses. *IEEE Trans. Softw. Eng.* **2022**, *48*, 7. [CrossRef]
8. Goues, C.L.; Pradel, M.; Roychoudhury, A. Automated program repair. *Commun. ACM* **2019**, *62*, 56–65. [CrossRef]
9. Yi, W. Anti-patterns for Java automated program repair tools. In Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 21–25 December 2020.
10. Dong, Y.; Zhang, L.; Pang, S.; Yin, W.; Li, H. Automatic repair of semantic defects using restraint mechanisms. *Symmetry* **2020**, *12*, 1563. [CrossRef]
11. Dong, Y.; Wu, M.; Zhang, L.; Yin, W.; Wu, M.; Li, H. Priority Measurement of Patches for Program Repair Based on Semantic Distance. *Symmetry* **2020**, *12*, 2102. [CrossRef]
12. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S.C. Context-Aware Patch Generation for Better Automated Program Repair. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018.
13. Al-Bataineh, O.I.; Grishina, A.; Moonen, L. Towards More Reliable Automated Program Repair by Integrating Static Analysis Techniques. In Proceedings of the 2021 IEEE 21st International Conference on Software Quality Reliability and Security, Haikou, China, 6–10 December 2021.
14. Dong, Y.; Sun, Y.; Wang, X. Automatic Repair Method for Null Pointer Dereferences Guided by Program Dependency Graph. *Symmetry* **2022**, *14*, 1555. [CrossRef]
15. Xuan, J.; Martinez, M.; DeMarco, F.; Clément, M.; Marcote, S.L.; Durieux, T.; Le Berre, D.; Monperrus, M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* **2017**, *43*, 34–55. [CrossRef]
16. Li, D.; Wong, W.E.; Jian, M.; Geng, Y.; Chau, M. Improving Search-Based Automatic Program Repair with Neural Machine Translation. *IEEE Access* **2022**, *10*, 51167–51175. [CrossRef]
17. Nilizadeh, A.; Leavens, G.T.; Le, X.-B.D.; Păsăreanu, C.S.; Cok, D.R. Exploring True Test Overfitting in Dynamic Automated Program Repair using Formal Methods. In Proceedings of the 2021 14th IEEE Conference on Software Testing, Verification and Validation, Porto de Galinhas, Brazil, 12–16 April 2021.
18. Jiang, J.; Xiong, Y.; Zhang, H.; Gao, Q.; Chen, X. Shaping Program Repair Space with Existing Patches and Similar Code. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, Amsterdam, The Netherlands, 16–21 July 2018.
19. Yuan, Y.; Banzhaf, W. Arja: Automated repair of java programs via multi-objective genetic programming. *IEEE Trans. Softw. Eng.* **2018**, *46*, 10. [CrossRef]
20. Yu, Z.; Martinez, M.; Danglot, B.; Durieux, T.; Monperrus, M. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *Empir. Softw. Eng.* **2019**, *24*, 33–67. [CrossRef]
21. Yang, D.; Liu, K.; Kim, D.; Koyuncu, A.; Kim, K.; Tian, H.; Lei, Y.; Mao, X.; Klein, J.; Bissyandé, T. Where were the repair ingredients for Defects4j bugs? *Empir. Softw. Eng.* **2021**, *26*, 122. [CrossRef]
22. Wang, Y.; Gao, F.; Wang, L.; Yu, T.; Zhao, J.; Li, X. Automatic Detection, Validation, and Repair of Race Conditions in Interrupt-Driven Embedded Software. *Empir. Softw. Eng.* **2022**, *48*, 346–363. [CrossRef]
23. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T. TBar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019.
24. Koyuncu, A.; Liu, K.; Bissyandé, T.; Kim, D.; Klein, J.; Monperrus, M.; Le, T.Y. Fixminer: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* **2020**, *25*, 1980–2024. [CrossRef]
25. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering, Hangzhou, China, 24–27 February 2019.
26. Liu, K.; Koyuncu, A.; Bissyandé, T.; Kim, D.; Klein, J.; Le, T.Y. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program rep systems. In Proceedings of the 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), Xi'an, China, 22–27 April 2019.
27. Martinez, M.; Monperrus, M. Ultra-large repair search space with automatically mined templates: The Cardumen mode of Astor. In Proceedings of the International Symposium on Search Based Software Engineering, Montpellier, France, 8–10 September 2018.
28. Yang, G.; Jeong, Y.; Min, K.; Lee, J.-W.; Lee, B. Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair. *Symmetry* **2018**, *10*, 92. [CrossRef]
29. Li, Y.; Wang, S.; Nguyen, T.N. DEAR: A Novel Deep Learning-based Approach for Automated Program Repair. In Proceedings of the 2022 IEEE/ACM 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022. [CrossRef]

30. Malyshev, N.; Dudina, I.; Kutz, D.; Novikov, A.; Vartanov, S. SMT Solvers in Application to Static and Dynamic Symbolic Execution. In Proceedings of the 2019 Ivannikov Ispras Open Conference, Moscow, Russia, 5–6 December 2019. [CrossRef]
31. Zhang, J.; Liu, K.; Kim, D.; Li, L.; Liu, Z.; Klein, J.; Bissyandé, T.F. Revisiting Test Cases to Boost Generate-and-Validate Program Repair. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution, Luxembourg, 27 September–1 October 2021. [CrossRef]
32. Tan, S.H.; Yoshida, H.; Prasad, M.R.; Roychoudhury, A. Anti-patterns in Search-Based Program Repair. In Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Seattle, WA, USA, 13–18 November 2016. [CrossRef]
33. Asad, M.; Ganguly, K.; Sakib, K. Impact of Combining Syntactic and Semantic Similarities on Patch Prioritization. In Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution, Cleveland, OH, USA, 29 September–4 October 2019. [CrossRef]
34. Etemadi, K.; Tarighat, N.; Yadav, S.; Martinez, M.; Monperrus, M. Estimating the potential of program repair search spaces with commit analysis. *J. Syst. Softw.* **2022**, *188*, 111263. [CrossRef]
35. Lin, B.; Wang, S.; Wen, M.; Mao, X. Context-Aware Code Change Embedding for Better Patch Correctness Assessment. *ACM Trans. Softw. Eng. Methodol.* **2022**, *31*, 3. [CrossRef]
36. Durieux, T.; Monperrus, M. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Ph.D. Thesis, Universite Lille 1, Villeneuve-d'Ascq, France, 2016. [CrossRef]
37. Saidani, I.; Ouni, A.; Mkaouer, M.W.; Palomba, F. On the impact of continuous integration on refactoring practice: An exploratory study on travistorrent. *Inform. Softw. Tech.* **2021**, *138*, 106618. [CrossRef]
38. Górski, T. Continuous delivery of blockchain distributed applications. *Sensors* **2022**, *22*, 128. [CrossRef]