symmetry

Article

Priority Measurement of Patches for Program Repair Based on Semantic Distance

Yukun Dong *, Meng Wu, Li Zhang, Wenjing Yin, Mengying Wu and Haojie Li

College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China; z19070061@s.upc.edu.cn (M.W.); s18070018@s.upc.edu.cn (L.Z.); z18070040@s.upc.edu.cn (W.Y.); z19070059@s.upc.edu.cn (M.W.); z19070058@s.upc.edu.cn (H.L.)

* Correspondence: dongyk@upc.edu.cn

Received: 23 October 2020; Accepted: 14 December 2020; Published: 17 December 2020

Abstract: Automated program repair is an effective way to ensure software quality and improve software development efficiency. At present, there are many methods and tools of automated program reapir in real world, but most of them have low repair accuracy, resulting in a large number of incorrect patches in the generated patches. To solve this problem, we propose a patch quality evaluation method based on semantic distance, which measures the semantic distance of patches by using features of interval distance, output coverage, and path matching. For each evaluation feature, we give a quantitative formula to obtain a specific distance value and use the distance to calculate the recommended patch value to measure the quality of the patch. Our quality evaluation method evaluated 279 patches from previous program repair tools, including Nopol, DynaMoth, ACS, jGenProg, and CapGen. This quality evaluation method successfully arranged the correct patches before the plausible but incorrect patches, and it recommended the higher-ranked patches to users first. On this basis, we compared our evaluation method with the existing evaluation methods and judged the evaluation ability of each feature. We showed that our proposed patch quality evaluation method can improve the repair accuracy of repair tools.

Keywords: automated program repair; patch quality evaluation; patch ranking; semantic distance

1. Introduction

Due to the increase in scale, complexity, and uncertainty of software, how to detect, locate, and repair defects in software becomes a problem which people are interested in. Related methods and tools of automated program repair play a key role in ensuring software quality. Especially in recent years, various automated program repair methods and related prototype tools have been proposed by researchers from different perspectives. The current automated program repair methods are mainly divided into two categories: (1) generation and verification method applies a series of modification operations to adjust the original faulty program, that is, a patch is generated and the correctness of the patch is verified by a large number of test cases; and (2) semantic-driven method aims to encode the faulty program, convert it into a constraint, and generate patches that meet the constraint. Patches generated by these repair tools contain a large number of errors, and the quality of the patches needs to be tested for better fixes.

The quality of a patch produced by an automated program repair tool determines the repair accuracy of the repair tool, but the repair accuracy of existing repair tools is very low [1]. In fact, patches generated by automated program repair tools have been manually reviewed, and only a few patches meet the developer's requirements. Li et al. [2,3] showed that GenProg, one of the most famous repair tools, produced possible correct patches for 55 defects, but only two of them were correct, which means that repair accuracy is only 4%. Patches generated by a repair tool might be verified by test cases to pass all test sets, while the patch that the developer needs is a high-quality patch that meets all the functions of the fix. Due to the complexity and diversity of repaired programs, it is very difficult for repair tools to generate high-quality patches. Therefore, finding a good patch quality evaluation method can improve the repair precision of the repair tools and can reduce the time for the developer to manually eliminate low-quality patches.

The quality of a patch is determined by many characteristics, such as retaining the expected function of the program in certain aspects; avoiding the introduction of new defects, side effects, or failures that directly cause system damage; and the maintainability of the patch. At present, the use of test cases is undoubtedly the most important method to verify the quality of automatically generated patches [4]. Moreover, developers have also used some other methods to verify patches, such as manual judgment [5,6], comparing with the developer's patch generated by historical errors [7,8], or using a specified number or excess test cases to evaluate the performance of the patched program [9]. Recently, Xiong et al. [10] provided a new direction for patch evaluation, filtering patches based on the behavior of the patched program relative to the original program in passing and failing tests. Regardless of which method we use, a high-quality patch is the simplest code fragment to ensure that the defects in the original program are fixed and the definitely correct function has not changed. To achieve this goal, we propose a method to rank the quality of patches based on semantic distance.

For a given set of plausible patches generated and verified by automated repair tools, the semantic distance between it and the original program should be determined. For a patch, semantically, it should fix the error function in the original program and ensure that other correct functions do not change. Based on this principle, interval distance, output coverage, and path matching are proposed to measure semantic distance. For each feature, a specific calculation formula is proposed to quantify the distance between the patched program and the original program, and the recommended value of the patch is given according to the distance. A quality evaluation tool SSDM covering the above functions is provided. Our quality evaluation method was evaluated on 279 plausible patches, which are from previous program repair tools, including Nopol [11,12], DynaMoth [13], ACS [14], jGenProg [15], and CapGen [16]. Experiments showed that our evaluation method successfully arranged the correct patches before the plausible but incorrect patches, and it recommended the higher-ranked patches to users first. Then, our quality evaluation method was compared with the existing evaluation methods. Finally, the recommended values of correct patches and incorrect patches generated by each tool were obtained through different features, and the evaluation effect of each feature on the patch was judged. The results show that the method proposed in this paper successfully obtained high-quality patches and improved the repair accuracy.

Section 2 introduces the relevant work of patch quality evaluation. Section 3 introduces the automated program repair evaluation model based on the semantic distance. Section 4 introduces the experiment and the results. Section 5 introduces the threats to validity. Section 6 is the conclusion.

2. Related Work

2.1. Repair Strategy

Different automated program repair tools have different repair strategies, and each repair strategy has different characteristics. The repair effect of the same defect is different for variety repair tools, so it is necessary to select the optimal patch to repair the defect. At this time, what is the optimal patch needs to be solved. Current repair tools generally use test cases to ensure that patches are correct, but patches that pass all test cases are not necessarily correct, and these patches are referred to as plausible patches. Although the plausible patch passes the selected test cases, it does not guarantee compliance with other program specifications outside the test cases. The repair tool produces a large number of plausible patches, which does not achieve the purpose of the repair program, but increases the difficulty of manual judgment.

For automated program repair tools based on generation and verification, most of them mutate programs in other locations to generate patch search space, then add patches to suspicious locations, and finally judge the correctness of patches through a large number of test cases. Le et al. [2,3] proposed a repair technology GenProg that uses genetic algorithms to guide patch generation and verification, it calculates the probability allocation of suspicious sentences based on a spectrum-based fault location algorithm, and generates a set of candidate patches through atomic change operations and single-point crossover. GenProg defines a fitness function that determines the quality of each candidate patch based on the number of passed and failed test cases. Wen et al. [16] proposed a repair tool, CapGen, which works at the abstract syntax tree level. At the same time, to reduce the search space, the context information of the AST node is used to estimate the correct probability of the candidate patch, and the patch is sorted. For the repair method based on generation and verification, the correct patches are sparse in the whole search space, while the number of plausible patches is relatively large. Therefore, on the one hand, it is difficult for the repair tool to find the correct patch from the search space, while, on the other hand, by increasing the search space, the number of correct patches can be increased, but the search difficulty is increased. Because more candidate patches increase the determination time, a larger number of the plausible patches hinders the discovery of the correct patches.

For semantics-based automated program repair methods, the information given by test cases is mainly used to synthesize a constraint expression, and then the expression is solved. Xuan et al. [11] developed a repair tool named Nopol, which is used to fix single condition errors that occur in conditional and loop statements. Nopol uses angel repair location technology to identify potential repair locations, and reverses the boolean values of condition statements that did not pass the test cases. Nopol collects the boolean values that make the target conditional statement pass all test cases and the variable values within the range of possible error conditional statements by running the test cases, according to which the expression is synthesized and the solution satisfying the expression is obtained. Durieux et al. [13] proposed a new tool named DynaMoth based on Nopol, which solves the problem that Nopol cannot synthesize conditions containing method calls. It addition, it can collect the runtime context for suspicious conditions, including variables, method calls, parameters, etc. Xiong et al. [14] proposed a program repair tool named ACS that generates precise patches. Its patch synthesis is mainly divided into two steps: (1) variable selection determines what variables should be used in the conditional expression; and (2) predicate selection determines which predicates should be executed on the variables. They proposed three sorting variables and predicates technologies of dependency-based ordering, document analysis, and predicated mining on this basis. For these semantic-based repair methods, due to the limited number of test cases used to synthesize constraints, the solved patches are a set of plausible patches that satisfy the limited test cases. If the number of test cases is further increased, the number of plausible patches will be significantly reduced, and the correct rate of patches will increase. It is very difficult to obtain a complete specification of the program to be fixed, so it is hard to achieve the quality of the patch through a complete test set.

There are some insurmountable problems in the above repair methods, 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 propose a patch quality evaluation method, which analyzes the distance between the defect program and the patch from the semantic point of view to detect the quality of the patch.

2.2. Patch Prioritization

Many repair methods use internal sorting components to rank patches according to the correct probability. On the one hand, correct and incorrect patches can be distinguished by setting appropriate thresholds, turning the sorting problem into a patch classification problem. On the other hand, different patches have different thresholds; a perfect patch prioritization method does not necessarily produce a perfect patch classification method [10]. Patch prioritization methods can be roughly divided into three categories. The first category is based on the frequency of patches in historical error repairs. Le et al. [7]

proposed a new technology named HDRepair, which uses error repair across the project development history to provide effective guidance and driver repair process. The second category uses the number of passing test cases to sort patches, which cannot effectively sort patches that may be correct. The third category uses the syntax and semantic distance between the original program and the patched to sort the patches. Elixir [17], ssFix [18], SimFix [19,20], and CCA [21] use the syntax similarity of error codes and repair components to prioritize patches. Syntax similarity mainly focuses on the text similarity, such as variable name similarity. CapGen uses three models to estimate patch probabilities and rank patches based on genealogical structure (such as the ancestors of abstract syntax tree nodes), access variables, and semantic dependencies [16]. Long [8] proposed a new patch generation system, Prophet, which works with a set of successful patches obtained from open source software libraries to learn the probability of correct code, and it is independent of the application. Our prioritization method is based on the semantic distance between the patched program and the original program to ensure the integrity of the correct function in the original program.

3. Evaluation of Patch Quality Based on Semantic Distance

Due to the low repair accuracy of the current automated program repair methods, the patches generated by current automated program repair methods have errors. Those generated patches can be divided into three categories: insertion, deletion, and modification. Based on categories, we provide a quality evaluation method according to patch features, which can filter low-quality patches and can improve the repair accuracy of the repair tools. In this section, the patch quality evaluation method based on semantic distance is proposed. The semantic distance between the patched and the original program is analyzed through interval distance, output coverage, and path matching. Finally, we convert a patch quality evaluation problem into a prioritization problem by the patch recommendation value that we obtained by distance.

To describe our evaluation model, the original program is assumed to be $\mathcal{P}$, and m is used to represent the method in the program, V is the set of variables, C is the set of constants, and e is the expression in the method. We assume that there is a set of test cases T for the original program $\mathcal{P}$, and $t_j \in \mathrm{T}$ is the jth test case. If there exists a defect at program point l, there is a patch set Q generated by the tool to fix the defect, $p_i \in \mathrm{Q}$ is the ith plausible patch, and the fix location is d_i.

3.1. Interval Distance

In many repairs [11–13], the patch is expressed as a boolean expression of the constraint execution path. We use the interval distance to measure the degree of "inconsistency" between any two boolean expressions, in other words the feasible region of the variable changes when the two expressions are true. When a defect appears in the boolean expression of the original program, the feasible interval between the patched and the original expression will change. However, if the change exceeds a given value, the semantics of the original program may be changed, which may cause new defects.

We use $Defect\left(\mathcal{P}, m, e, t_j\right)$ to represent the defect found by the test case t_j in the boolean expression e, which belongs to the method m of the original program $\mathcal{P}$. There may be many test cases that can detect this defect and put it into the test case set T. A patch p_i is given to the defect, so as to obtain the defect repair $Defect\left(\mathcal{P}, m, p_i\left(e\right), t_j\right)$. When the repair tool repairs a defective conditional statement, it will collect the set of available variables within the scope of the suspicious condition or will search for available patches in the patch space. In other words, the local variables in the method and the global variables defined outside the method will become the repair composition of the patch. Once the repair patch passes the test, the tool considers the patch to be the correct. After this, the patch may introduce a variable a that does not exist in the original expression, this patch p_i is put into the patch set Q, and its interval distance $\mathcal{L}_i$ is set to ∞. In addition, when the boolean expression of the patch can be directly judged as true (false) , the expression e can be replaced with true (false) , i.e., $e = subs\left(e, true\right)$, and the patch is not put into the set Q. Then, the interval distance $\mathcal{L}$ between $p_i \notin \mathrm{Q}$ and e is calculated by the formula $\left(p_i\left(e\right) \wedge \neg e\right) \vee \left(\neg p_i\left(e\right) \wedge e\right) = true$. This formula answer is an interval. In the calculation, if the boolean expression on both sides of the logic and ($\wedge$) is solved as a numerical interval, the specific interval range can be obtained; if there are unknown or unsolvable expressions in the boolean expression, the unknown expressions are regarded as a constant value and brought into the interval to represent the interval range; if there is an expression in the boolean expression that is not an interval range but a boolean value calculated by boolean expression, the boolean value is directly brought into the whole boolean expression to do the logical operation. Finally, the interval on both sides of the logic and ($\wedge$) is taken to intersect, and the interval on both sides of the logic or ($\vee$) is taken to be the union, so as to obtain the solution interval. There are infinite solutions that satisfy the formula in the interval, in order to make it easier to quantify the distance, we only take the integer that satisfies the interval, that is $|(p_i(e) \wedge \neg e) \vee (\neg p_i(e) \wedge e) = true| \sim \mathcal{L}$, $|\cdots|$ means rounding. $\mathcal{L}_i^{mod}$ is used to represent the semantic distance between patch i and the original program in the interval distance feature, and P_i^{mod} is the recommended value of patch i in the interval distance feature. Their calculations are shown in Equations (1) and (2), respectively.

$$\mathcal{L}_i^{mod} = \begin{cases} \infty, p_i \in \mathrm{Q} \\ |(p_i(e) \wedge \neg e) \vee (\neg p_i(e) \wedge e) = true|, p_i \notin \mathrm{Q} \end{cases} \tag{1}$$

$$\mathrm{P}_i^{mod} = \begin{cases} 0, if\ \mathcal{L}_i^{mod} = 0\ or\ \mathcal{L}_i^{mod} = \infty \\ \frac{1}{1+\mathcal{L}_i^{mod}}, otherwise \end{cases} \tag{2}$$

Example 1. *The program in Figure 1a is the correct program after the manual repair, in which Lines 3–4 of code are deleted, and Lines 5–6 of patch are added.* $charno < sourceExcerpt.length()$ *is changed to* $charno \leq sourceExcerpt.length()$. *Figure 1b shows the program repairing by the faulty patch generated by DynaMoth, it expands the scope of boolean expressions in* $if\,(excerpt.equals(LINE)\ \&\&\ 0 \leq charno\ \&\&\ charno < sourceExcerpt.length())$ *to true. We can use the above formula to calculate the semantic distance between the manually repaired program and the original program, and the semantic distance between the repaired program generated by automated repair tool and the original program:*

$$\begin{aligned}\left(\mathcal{L}_i^{mod}\right)_{human} &= |(charno \leq sourceExcerpt.length() \wedge \neg charno < sourceExcerpt.length()) \\ &\vee (\neg charno \leq sourceExcerpt.length() \wedge charno < sourceExcerpt.length()) \\ &= (charno = sourceExcerpt.length()) = true| = 1\end{aligned}$$

$$\begin{aligned}\left(\mathcal{L}_i^{mod}\right)_{auto} &= |((excerpt.equals(LINE)\,\&\&0 \leq charno\&\&charno < sourceExcerpt.length()) \wedge false) \\ &\vee (\neg(excerpt.equals(LINE)\,\&\&0 \leq charno\&\&charno < sourceExcerpt.length()) \wedge true) \\ &= (excerpt.equals(LINE)\ \|\ 0 > charno\ \|\ charno \geq sourceExcerpt.length()) \\ &= (-\infty, 0) = true| = \infty\end{aligned}$$

$$\left(\mathrm{P}_i^{mod}\right)_{human} = \left|\frac{1}{1+\left(\mathcal{L}_i^{mod}\right)_{human}}\right| = 0.5 \qquad \left(\mathrm{P}_i^{mod}\right)_{auto} = 0$$

It is obvious that the recommended value of manual repair patches is 0.5, while the recommended value of automatically generated patches by repair tools is 0. When the patch generated by DynaMoth is at $charno > sourceExcerpt.length()$, *it will cause a null pointer exception. This indicates that the interval change of the patch should be controlled within a certain range, otherwise the defect will not be fixed and new errors will be introduced.*

```
if(sourceExcerpt!=null){
...
-if(excerpt.equals(LINE)&&0<=charno
- && charno<sourceExcerpt.length()){
+if(excerpt.equals(LINE)&&0<=charno
+ &&charno<=sourceExcerpt.length()){
for(int i = 0; i<charno; i++){
char c = sourceExcerpt.charAt(i);
if(Character.isWhitespace(c)){
...
}
```

(**a**) Human Patch

```
if(sourceExcerpt!=null){
...
-if(excerpt.equals(LINE)&&0<=charno
- &&charno<sourceExcerpt.length()){
+if(error.equals((Java.lang.Object)
+ error)){
for(int i = 0; i<charno;i++){
char c = sourceExcerpt.charAt(i);
if(Character.isWhitespace(c)){
...
}
```

(**b**) Patch #78 DynaMoth

Figure 1. Patch for Closure 62 in Defects4J.

3.2. Output Coverage

Some patches generated in automated program repair tools can be equivalent to code deletion [22]. In some cases, the deletion of redundant code fixes the original program defects and does not create new defects. However, in more cases, the defective code is not redundant, and its deletion will affect the realization of the program function as well as generate new defects. To achieve this, we first obtain the patch of deletion type or equivalent to deletion type. For each patch, it is contextually analyzed to determine whether it is semantically equivalent to deletion: (1) a single statement or the entire code block is deleted; (2) the code is put into an if judgment statement, and its judgment condition is equal to false; or (3) a single return or exit statement is inserted. Then, we implement symbolic execution on the original program $\mathcal{P}$ and terminate the execution at code deletion, in which the symbol state and symbol path constraints can be received. The passing test case $t_j \in \mathrm{T}$ that satisfies the path constraint is obtained from the test case set T (if there are few test cases, you can generate additional test cases that satisfy the constraints) and execute in the original program $\mathcal{P}$ and the patched program p_i to get the different output value sets $\mathrm{C}_{\mathcal{P}}$ and C_{p_i} of the original program and the patched program. Finally, what percentage the output value set C_{p_i} of the ith plausible patch can cover the original output value set $\mathrm{C}_{\mathcal{P}}$ is called output coverage θ_i. After the code is deleted, whether the function implemented by the original code has been changed can be judged by the output coverage θ_i. If the coverage rate θ_i is high, the code can be considered as redundant code, which means the code function will not be affected and the original defect of the program is repaired. If the coverage θ_i is low, the function of the program is affected by the code, and the corresponding patch is of low quality. We use θ_i to represent the output coverage of the ith patch, and C_{p_i} represents that the patched program covers the number of different outputs in the input–output example, $\mathrm{C}_{\mathcal{P}}$ represents the number of different outputs in the original input–output example, and P_i^{cov} represents the recommended value of patch i on the output coverage. The calculation of θ_i and P_i^{cov} is shown in Formulas (3) and (4).

$$\theta_i = \frac{\mathrm{C}_{p_i} \cap \mathrm{C}_{\mathcal{P}}}{\mathrm{C}_{\mathcal{P}}} \tag{3}$$

$$\mathrm{P}_i^{cov} = \theta_i \tag{4}$$

Example 2. *Figure 2a shows the program with the incorrect patch, which is equivalent to deleting the statement on Line 8 and Figure 2b shows the program with the correct patch. We can analyze the data flow of the original program to obtain the path constraint* $\{a < 0 \; || !b = false, c = true\}$ *at the statement on Line 8. The passing test cases that meet this constraint in Table 1 are* $\{-1, true, true, \{4,2,5\}\}$ *and* $\{-3, false, true, \{4,2,5,7,1\}\}$*. We use these test cases to test the patched programs in Figure 2a,b, respectively, and then calculate their output coverage:*

$$(\theta_i^{cov})_{Incorrect} = 0 \qquad (\mathrm{P}_i^{cov})_{Incorrect} = 0$$

$$(\theta_i^{cov})_{Correct} = 1 \qquad (\mathrm{P}_i^{cov})_{Correct} = 1$$

The recommended value of the correct patch is 1 and the recommended value of the incorrect patch is 0. This shows that the output coverage has significance for patch quality evaluation.

```
example(int a,boolean b,boolean c,int[] m){
int i=0;
if(a>=0&&!b){
i=a;
}else {
-if (c){
+if(false){
i=m[-a];
}else {
i=m[a];
}
}
return i;
}
```

(**a**) Incorrect Patch

```
example(int a,boolean b,boolean c,int[] m){
int i=0;
if(a>=0&&!b){
i=a;
}else {
-if (c){
+if(c&&a<0){
i=m[-a];
}else {
i=m[a];
}
}
return i;
}
```

(**b**) Correct Patch

Figure 2. A buggy program with two possible repairs.

Table 1. Test cases for the program in Figure 2.

		Input		**Output**
a	**b**	**c**	**m**	**i**
2	true	true	4,2,5	error
−1	true	true	4,2,5	2
−3	false	true	4,2,5,7,1	7
3	true	true	4,2,5,7,1	error
3	false	true	4,2,5,7,1	3

3.3. Path Matching

This feature is mainly used to compare the execution path of the program before and after being repaired and to prioritize the patches according to the execution path. The execution path of the passing test case before and after the patch is the same, which means that the patch does not bring path error to the program, and the original function of the program is not changed. However, the execution path of the failing test cases is changed and comes to the correct output, which indicates that the original error path has been repaired, with the program function restored. This is using the behavior of the original program on passing test cases as a partial description of the required behavior of the program [23]. Therefore, we hope to have a reasonable patched program that can not only pass all test cases, but also make the execution path of the patch similar to the original program on passing test cases, and which have difference with the execution path on the failed test cases.

We use $Path\left(\mathcal{P},m,t_j\right)$ to represent the execution path that the method m in the program $\mathcal{P}$ is tested by the test case t_j. $\left|Path\left(\mathcal{P},m,t_j\right)\right| \geq 1$ means that the method m is not only called once. When the defect in method m of program $\mathcal{P}$ is fixed with a patch p_i, the execution path is expressed as $Path\left(\mathcal{P},p_i\left(m\right),t_j\right)$, where $p_i\left(m\right)$ refers to the application of the ith plausible patch p_i in method m of program $\mathcal{P}$. The patched program generally does not change the number of method calls in the original program. Thus, for the test case t_j, the number of calls to method m in the patched program are equal to the original one, that is $\left|Path\left(\mathcal{P},m,t_j\right)\right| = \left|Path\left(\mathcal{P},p_i\left(m\right),t_j\right)\right|$. If $\left|path\left(\mathcal{P},m,t_j\right)\right| \neq \left|path\left(\mathcal{P},p_i\left(m\right),t_j\right)\right|$, which indicates that there are differences in the repair location of the patch. Such patch p_i is stored in the illegal patch set W, the patch location is analyzed, and the quality of the patch is evaluated on the patch location feature. Then, we compare the similarity of the execution path which is performed on the patch $p_i \in \mathrm{Q} - \mathrm{W}$. We convert the program statements on the execution path $Path\left(\mathcal{P},p_i\left(m\right),t_j\right)$ of the patched program into a string $s_{p_i(m)}$ and compare it with the string s_m of the program statement on the execution path $Path\left(\mathcal{P},m,t_j\right)$ of the original program. Because the execution path of the test case t_j in the patched program and the original program is different, the length of the string is also different. The edit distance we used is to measure the difference between strings. The edit distance is the minimum number of operands required to convert the string s_1 to the string s_2 using character operations, which can compare strings of different lengths. In general, if the edit distance between two strings is smaller, the similarity of them will be the higher.

We obtain the passing test set T_{true} and the failing test set T_{false} from the original test set. The maximum value $max\left(Lev\left(s_{p_i(m)}, s_m\right)\right)$ in the edit distance *Lev* of the execution path of $t_j \in \mathrm{T}_{true}$ is used to represent the maximum impact of the patch p_i on the function of the original program, and the average value $ave\left(Lev\left(s_{p_i(m)}, s_m\right)\right)$ in the edit distance of the execution path of $t_j \in \mathrm{T}_{false}$ is used to represent the modification size of the patch p_i on the defective program. For passing cases, incorrect patches may affect execution path of minority passing test cases; we use maximum value of edit distance to focus on those test cases. However, for the failing test cases, patches can change the execution path of them, so we use average of edit distance to acquire the effects of patches on execution path. For a high-quality patch, the correct function of the original program can be guaranteed, with the repaired defect at meanwhile. Based on this point of view, we also consider the impact of the patch p_i on the function of the original program, and the modification of the patch p_i on the defective program. At same time, it requires relatively small impact on the original function and relatively large impact on modification of the defective program. Therefore, we propose specific Formulas (5) and (6), where $\mathcal{L}_i^{Lev}$ represents the semantic distance of patch *i* from the original program by the path matching feature and P_i^{Lev} represents the recommended value of patch *i* by the path matching feature.

$$\mathcal{L}_i^{Lev} = \frac{1}{2} * \left(max\left(Lev\left(s_{p_i(m)}\right), s_m\right) + ave\left(Lev\left(s_{p_i(m)}, s_m\right)\right)\right) \tag{5}$$

$$\mathrm{P}_i^{Lev} = \begin{cases} 0, if\ ave\left(Lev\left(s_{p_i(m)}, s_m\right)\right) = 0\ or\ \dfrac{max\left(Lev\left(s_{p_i(m)}, s_m\right)\right)}{ave\left(Lev\left(s_{p_i(m)}, s_m\right)\right)} \geq 1 \\ 1 - \dfrac{max\left(Lev\left(s_{p_i(m)}, s_m\right)\right)}{ave\left(Lev\left(s_{p_i(m)}, s_m\right)\right)}, otherwise \end{cases} \tag{6}$$

Example 3. *We use the test cases in Table 1 to analyze the execution paths of the passing test cases and the failing test cases in Figure 2a,b and compare their path matching degrees. The execution path is shown in Table 2. The maximum value of the edit distance in the execution path of the passing test cases and the average value of the edit distance in the execution path of the failing test cases can be calculated, respectively, and then the semantic distance and the recommended value of the patch can be obtained further by them:*

$$\left(\mathcal{L}_i^{Lev}\right)_{Incorrect} = \frac{1}{2} * (12 + 13) = 12.5 \qquad \left(\mathrm{P}_i^{Lev}\right)_{Incorrect} = 1 - \frac{12}{13} \approx 0.08$$
$$\left(\mathcal{L}_i^{Lev}\right)_{Correct} = \frac{1}{2} * (5 + 13) = 9 \qquad \left(\mathrm{P}_i^{Lev}\right)_{Correct} \approx 1 - \frac{5}{13} = 0.6$$

It can be seen that the recommended value of the correct patch is higher than the recommended value of the incorrect patch. This indicates that path matching has a significant effect on the quality evaluation of the patch.

Table 2. The execution paths of different test cases for the program in Figure 2.

	Input				Output		
	a	**b**	**c**	**m**	**Incorrect**	**Correct**	**Original**
Passing	3	false	true	4,2,5,7,1	3,4,5,14	3,4,5,14	3,4,5,14
	−1	true	true	4,2,5	3,4,8,11	3,4,8,9,14	3,4,7,9,14
	−3	false	true	4,2,5,7,1	3,4,8,11	3,4,8,9,14	3,4,7,9,14
Failing	3	true	true	4,2,5,7,1	3,4,8,11,14	3,4,8,11,14	3,4,7,9
	2	true	true	4,2,5	3,4,8,11,14	3,4,8,11,14	3,4,7,9

The recommended values of the patch obtained by the above three evaluation features range from 0 to 1. To obtain the final joint recommendation value of the patch, we obtain the recommended value of each feature and take its mean value [20].

Example 4. *The test cases in Table 1 are used to obtain the recommended values of the incorrect patch and the correct patch in Figure 2a,b under the output coverage and path matching features, as shown in Examples 2 and 3. Then, the recommended value under the interval distance feature is obtained. Its mean value is taken to obtain the joint recommended value of the patches:*

$$\left(\mathcal{L}_i^{mod}\right)_{Incorrect} = |\,(c \wedge \neg false) \vee (\neg c \wedge false) = true| = 1 \qquad \left(\mathrm{P}_i^{mod}\right)_{Incorrect} = 0.5$$
$$\left(\mathcal{L}_i^{mod}\right)_{Correct} = |\,(c \wedge \neg\,(c\&\&a < 0)) \vee (\neg c \wedge (c\&\&a < 0)) = true| \approx \infty \qquad \left(\mathrm{P}_i^{mod}\right)_{Correct} = 0$$
$$\left(\mathrm{P}_i^{joint}\right)_{Incorret} = \frac{0.5 + 0 + 0.08}{3} \approx 0.1933 \qquad \left(\mathrm{P}_i^{joint}\right)_{Corret} = \frac{0 + 1 + 0.6}{3} \approx 0.5333$$

It is obvious that the joint recommendation value of the correct patch is greater than the incorrect patch, which means that our approach has a better effect.

4. Experiment and Result Analysis

4.1. Method Implementation

We implement the patch quality evaluation tool SSDM through the Java language; it can evaluate the quality of patches produced by defects in Java programs. When using it, the user can directly add the quality evaluation tool after the repair tools and do post-processing for the plausible patches generated by the repair tools to get the quality ranking of the patches. In addition, the patch generated by any repair strategy tool can be evaluated by SSDM, because we have no specific requirements for the repair tool there. The original program and test case set are required when evaluating the patches, and SSDM can directly call the input programs and test set in the repair tool, or the developer can input them manually.

In implementation, we use Java to achieve an automated evaluation of the features which do not have suitable automation tools, such as interval distance. The rest of the features are automated using existing tools. When implementing the output coverage feature, we use the existing symbolic execution tool LimeTB to obtain symbol state and path constraints of the symbol, and also obtain test case set that meets the constraints. After that, test case set was executed in the original program and the patched program to get the output coverage. Under the path matching feature, the execution statement of the test case will be converted into a string by using Javalang [24]. Then, we calculate the similarity and difference of the execution path of the patched program and the original program by using the edit distance under the same test case execution. According to the similarity and difference of the execution path, the semantic distance under the path matching feature is judged.

4.2. Experimental Data Collection

Nopol, DynaMoth, ACS, jGenProg, and CapGen were selected as the analysis objects of the quality evaluation method. Among these tools, Nopol is a repair tool that synthesizes constraints based on semantic information and generates conditions and loop statements, and DynaMoth is a repair tool that can further synthesize method call conditions on the basis of Nopol. ACS is a repair tool for statistical and heuristic repair of incorrect condition statements based on multiple information sources. jGenProg is a Java language implementation of GenProg that uses genetic algorithms to guide the patch generation and verification. As for CapGen, it is a context-aware repair tool at the level of fine-grained abstract syntax tree nodes. The specific repair details can be found in the relevant work section. In this article, the repair tools selected cover two patch synthesis methods: Nopol, DynaMoth, and ACS cover semantic-based patch synthesis methods and jGenProg and CapGen cover patch synthesis methods based on generation and verification. Analyzing the effects of the quality evaluation methods we proposed in different types of repair tools can more comprehensively measure the evaluation capabilities of the quality evaluation methods.

We collected the patch dataset of these tools from the existing articles [13,14,25], and the statistics of the datasets are shown in Table 3. In the experiment of Martinez et al. [25], all repair tools used Defects4J as the baseline pool for patch generation. Nopol can generate a corresponding patch for 35 defects in Defects4J, and jGenProg can generate a patch for 27 defects. In addition, the author judged the correctness of the patches generated by each tool through manual analysis and obtained that the 35 patches generated by Nopol contained 25 incorrect patches and 5 correct patches, and the correctness of the remaining patches was unknown. Among the 27 patches generated by jGenProg, 5 are correct, 18 are incorrect, and the correctness of the remaining patches is unknown. In DynaMoth's report, the dataset of the generated patches were given. To obtain the number of correct patches, we manually analyzed the correctness of these patches (https://github.com/wls860707495/DynaMoth). Because DynaMoth also uses Defects4J as a benchmark program, and the manual repair patch for defects in Defects4J has been given, we comparef the manual repair patch with the DynaMoth generated patch for syntax and semantics. It esd obtained that DynaMoth generated 50 patches for the defects in Defects4J, of which 2 are correct patches and 48 are incorrect patches. The number of patches generated by ACS esd obtained from the evaluation report of ACS [14], and the report indicates that the number of patches that ACS generated is 23, of which the number of incorrect patches is 5 and the number of correct patches is 18. According to CapGen's report [16], it generated plausible patches for 21 bugs, 21 of which were correctly fixed, and 15 of which have never been fixed by existing methods. In total, 247 plausible correct patches were generated by CapGen, 28 of which were detected as correct patches. CapGen also uses Defects4J as a benchmark pool for patch generation.This means that all the evaluation tools we used are patching with Defects4J as a benchmark pool, which facilitates our next step.

Table 3. The number of plausible patch and correct patch in the tools.

Project	Nopol	DynaMoth	ACS	jGenProg	CapGen
Chart	1/6	1/11	2/2	0/7	5/66
Lang	3/7	0/6	3/4	0/0	9/29
Math	1/21	1/28	12/16	5/18	14/152
Time	0/1	0/5	1/1	0/2	0/0
Total	5/35	2/50	18/23	5/27	28/247
Precision	14.3%	4%	78.3%	18.5%	11.3%

X/Y: X is the number of correct patches generated by the approach and Y is the number of plausible patches generated by the approach which pass all test cases.

Table 3 shows that most of the patches generated by the repair tool are incorrect patches after syntax and semantic comparison with the correct patches written by developers. These patches have passed the judgment of the test cases in the repair tool and recommended them to the developers. However, due to the inadequacy of test cases, many patches that pass the test set may still be incorrect patches, and the developer needs to further judge the quality of the patches. The low repair accuracy of the repair tool increases the workload of the developers. Sometimes, many incorrect patches are generated, which may lead to a higher judgment time than the time to directly repair the patch [26].

4.3. Ability to Evaluate Real Patches

The five representative semantics-based and generation-verification-based repair tools were selected to evaluate the quality of the patches generated by the Defects4j benchmark pool. First, we evaluate the quality of the patches produced by the repair tool CapGen. Since the tool does not generate plausible patches for many defects or only generates a single plausible patch, we only selected the defects with multiple plausible patches, and the results are shown in Table 4. Then, we used five repair tools to repair the defects separately, obtained all the plausible patches that can repair the defect. We evaluated them through our evaluation method and obtained the result shown in

Table 5. If our evaluation method can arrange the correct patch produced by a single repair tool before other plausible patches, as well as arrange the correct patch before other plausible patches in a large number of patches that generated by multiple repair tools, our evaluation method has good evaluation capability. The correct patch here is the patch with high similarity compared with the manual repair patch. The evaluation data are available online https://github.com/wls860707495/Dataset.

Table 4 shows that the correct patches are ranked in the top three by our evaluation method, and most of the correct patches are even ranked in the top one, which means that our evaluation method does have obvious evaluation effect. At the same time, it can be found that the correct patch for Math 63 is ranked in third place. In Figure 3, there is only one return statement in the "equals" function, which affects the correctness of the program. The patch here can only be evaluated with the path matching feature, while the other features are not suitable for the patch type. Since the patch does not affect the number of method calls, the execution path in the "equals" method is the only one that need to be judged, and the method contains only a return statement, which makes it unable to reflect obvious differences in path matching feature. Therefore, this correct patch cannot be judged as a high quality patch by our evaluation method.

```
416 public static boolean equals(double x, double y) {
417 -return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
417 +return org.apache.commons.math.util.MathUtils.equals(x, y, 1);
418 }
```

(**a**) Correct Patch

```
416 public static boolean equals(double x, double y) {
417 -return (Double.isNaN(x) && Double.isNaN(y)) || x == y;
417 +return x == y || x == y;
418 }
```

(**b**) Plausible Patch

Figure 3. Patch for Math 63 in Defects4j generated by CapGen.

Table 4. Evaluation effect of the patches generated by CapGen.

Project	Bug ID	Crt	Rv	Plausible	Rank
Chart	8	1	0.625	62	1
Chart	11	2	0.6525	2	1
Lang	43	3	0.58	4	1
Lang	57	3	0.7025	3	1
Lang	59	1	0.52	20	1
Math	5	1	0.7825	4	1
Math	53	2	0.7367	2	1
Math	63	1	0.415	9	3
Math	80	1	0.6667	125	1
Math	85	1	0.9333	4	1

Crt represents the number of the correct patches. **Rv** indicates the recommended value for the first correct patch. **Plausible** represents the number of the plausible patches. **Rank** indicates that the first correct patch is sorted in the plausible patch.

At the same time, Table 5 shows that the correct patches generated by the five repair tools are also ranked in the top three by our evaluation method. Due to the weak repair capability of the current repair tools and the limited number of patches for defects, it can generate only a single plausible patch for many defects or even no plausible patch. Therefore, when a defect cannot get plausible patches with one tool, we can use multiple repair tools to repair it. Then, the quality evaluation through our method can still get a better evaluation effect.

Table 5. Evaluation effect of the patches generated by five repair tools.

Project	Bug ID	Rank	Rv	Plausible				
				Nopol	DynaMoth	ACS	jGenProg	CapGen
Chart	1	2	0.384	-	0/1	-	0/1	1/1
Chart	3	-	-	0/1	-	-	0/1	-
Chart	5	1	0.4585	1/1	0/1	-	0/1	-
Chart	8	1	0.625	-	-	-	-	1/62
Chart	11	1	0.6525	-	-	-	-	2/2
Chart	13	-	-	0/1	0/1	-	0/1	-
Chart	14	1	0.875	-	-	2/2	-	-
Chart	15	-	-	-	0/1	-	0/1	-
Chart	25	-	-	0/1	0/1	-	0/1	-
Math	5	1	0.8333	-	0/1	1/1	1/1	1/4
Math	33	1	0.4444	0/1	0/1	-	-	1/1
Math	40	-	-	0/1	0/1	-	0/1	-
Math	50	1	0.95	1/1	0/1	-	1/1	-
Math	53	1	0.875	-	-	-	1/1	2/2
Math	57	1	0.4444	0/1	0/1	-	-	1/1
Math	58	1	0.5	0/1	0/1	-	-	1/1
Math	63	3	0.415	-	-	-	-	1/9
Math	73	2	0.381	0/1	-	0/1	1/1	-
Math	80	1	0.6667	0/1	0/1	-	0/1	1/125
Math	82	1	0.9167	0/1	0/1	1/1	0/1	-
Math	85	1	0.9333	0/1	0/1	1/1	0/1	1/4
Lang	59	1	0.52	-	-	-	-	1/20

X/Y: X represents the number of correct patches and Y represents the number of plausible patches. "-" indicates that there is no plausible or correct patch for the defect, and cannot be sorted.

4.4. Comparison with Existing Methods

At present, the existing patch quality evaluation methods are more about embedding them into a repair tool to improve the repair accuracy of the tool. It is difficult to transplant these methods to other repair tools. Therefore, we try to choose evaluation methods with a larger applicable scope here and compare these existing methods with our quality evaluation method. The evaluation data here are the plausible patches in Table 5.

First, we used the anti-pattern to evaluate the plausible patches in Table 5. The strategy of anti-pattern is different from that of human patch pattern. The human patch patterns are to produce patches that are closer to human repairs, while anti-pattern essentially uses patterns to capture unreliable modifications to defective programs, so that the repairs are more correct and complete. To apply anti-patterns to existing datasets, we found seven kinds of anti-patterns defined by t Tan et al. [27] and manually checked which patches belong to these anti-patterns. Using the anti-pattern, we filtered 20 plausible patches, of which 17 were incorrect patches and 3 were correct patches. These three correct patches are ranked in the top three by our evaluation method and are recommended to developers first. As shown in Figure 4, although this patch is the correct patch, it is excluded by the sixth anti-pattern, which does not allow the insertion of return and go to statements at any position, except after the last statement of the CFG node. Our method ranks the patch first and recommends it to developers first. The anti-pattern is also a regular summary of existing defects, which does not apply to all defects. There are countless variations in the form of defects in a program, so this limitation allows it to exclude some of the correct patches.

```
152 MathUtils.checkNotNull(rhs);
153 +if (isNaN || rhs.isNaN) {
153 +return NaN;
153 +}
153 return createComplex(real + rhs.getReal(),
154 imaginary + rhs.getImaginary());
```

Figure 4. Patch for Math 53 in Defects4j generated by jGenProg.

At the same time, we used fix patterns to evaluate the quality of patches. We used the common bug fix patterns discovered by Yuan et al. [7] from 3000 bug fixes in more than 700 open source projects on GitHub to judge the correctness of the patches in Table 5 and obtained the patch ranking by the frequency of these patterns. The fix patterns ranked the correct patch first from 7 out of the 22 bugs that can be fixed correctly, while our evaluation method ranked the correct patch for 14 bugs first. The results show that our method is better than the ranking method using the fix patterns. As shown in Figure 5, the correct patch is placed at the back by the repair pattern. When there is a correct patch that is rarely in the historical bug fix data, the fix pattern method will give it a very low score and arrange it at a very low position because the fix patterns are based on historical fixes to sort the patches. For a defect, there are many ways to repair it. If only judging the common patches as correct, many correct and uncommon patches will be missed.

```
186 if (x == x1) {
187 final double delta = FastMath.max(rtol * FastMath.abs(x1),atol);
188 -x0 = 0.5 * (x0 + x1 - delta);
189 f0 = computeObjectiveValue(x0);
190 }
```

Figure 5. Patch for Math 50 in Defects4j generated by jGenProg.

4.5. Ability to Evaluate Patch Quality by Different Features

To judge the contribution of each evaluation feature to the patch quality evaluation, we separately studied the evaluation ability of each feature for the patch. Here, the patches we evaluated are the plausible patches in Table 5, and the recommended values of the patches were obtained by evaluating the 25 correct patches and 254 incorrect patches, respectively, with three evaluation features. If this feature has a better evaluation ability for the patches, it has a higher recommended value for correct patches and a relatively low recommended value for incorrect patches; conversely, if this feature has poor evaluation ability for the patches, its recommendation value for the correct patch is lower than that for the incorrect patch. Moreover, if the recommended value of this feature for the correct patch is higher than other features, it means that this feature has made a greater contribution in the joint feature evaluation. Due to the different types of patches, the required evaluation features are different, and each feature can only evaluate the patch that is compatible with it, which leads to a different number of patches evaluated by each feature, but this does not affect the effect analysis. The evaluation results are shown in Figure 6.

Figure 6 shows that the path matching feature has the best evaluation effect on the correct patch, with an average value higher than 0.7. At the same time, the recommended value of the correct patch and the incorrect patch under this feature is quite different, which means that this feature can well distinguish the correct patch. However, the evaluation effect of the interval distance feature is relatively poor, which is close to the median of the recommended value for the correct patch with the output coverage. However, the median difference between the recommended values for the correct patch and the incorrect patch is small, which indicates that this feature cannot distinguish the correct patch from the incorrect patch. The key to the effect of interval distance is that we limit the evaluation criteria of interval. We believe that when the change of the feasible interval between the original program and the patched program is within a certain range, the higher is the probability that the patch is correct and the better is the quality. However, some patches have a great change in the feasible interval compared with the original program, but they are correct patches. These patches affect the evaluation effect of interval distance. As shown in Figure 7, the boolean expression of the correct patch repaired by humans is $(startIndex >= source.length())||(endIndex > source.length())||$; the feasible interval between it and the error code varies greatly, but also ensures correctness. To ensure the correctness of the patch, we limit the feasible interval change to a certain range, but there are still exceptions. As long as there are special cases, it may affect the evaluation effect, which does not mean that the rule is wrong. Compared with the interval distance, the output coverage has a better evaluation effect. Under this feature, the median of the recommended value of the correct patch and the incorrect patch is quite different, and the recommended value of the correct patch is higher, with the median exceeding 0.5.

The results show that the path matching feature has the best evaluation effect in the patch evaluation based on semantic distance, which can clearly distinguish the correct patch and significantly contribute to the joint feature evaluation of the patch. However, the evaluation effect of interval distance feature is relatively poor, the recommended value for the correct patch is low, and it is difficult to distinguish the correct patch.

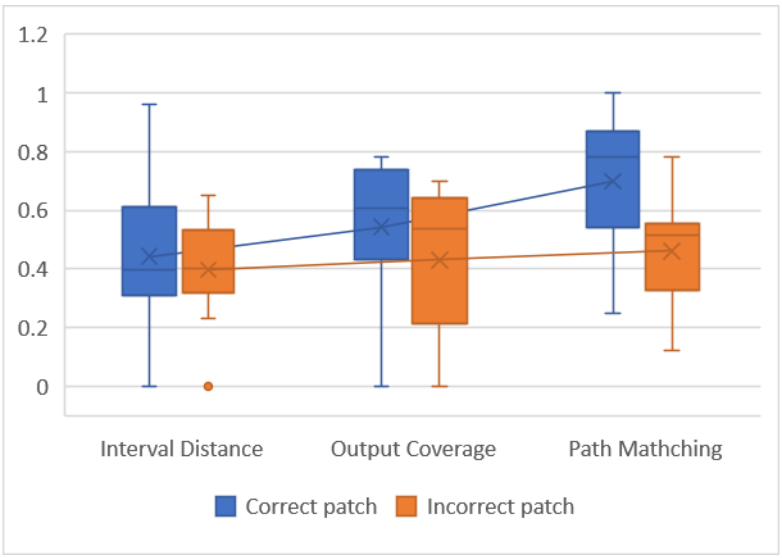

Figure 6. Evaluation effect of different features on patches.

```
375 startIndex = pos.getIndex();
376 int endIndex = startIndex + n;
377 -if(
377 +if((startIndex >= source.length())||(endIndex > source.length())||
378 source.substring(startIndex, endIndex).compareTo(
379 getImaginaryCharacter()) != 0) {
```

Figure 7. Patch for Math 101 in Defects4j generated by human.

5. Threats to Validity

Threats to External Validity. The main 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 patches generated by defects in Defects4j. This means that our evaluation results may not be generalized to other datasets. Therefore, we try to use INTROCLASSJAVA [28] as the benchmark pool, which is the Java version of the IntroClass benchmark pool [29]. This benchmark pool contains 297 bugs in student work and has a set of suitable test cases, which is very convenient for our evaluation method.

First, we use the above five repair tools to generate a large number of patches for the defects in the INTROCLASSJAVA benchmark pool, and then test them through a set of test cases to obtain plausible patches that meet the requirements of the test cases. Then, we use our evaluation method to evaluate whether our evaluation method can be extended to other defect patches. The experimental results show that our evaluation method successfully ranked most of the correct patches in the top three. This shows that our evaluation method can be extended to other defect patches.

Threats to Internal Validity. The internal threat is that, when we evaluate the patch, there may be some patches that make different types of modifications to the program in different locations. These modifications may be in different methods, leading to our evaluation method is not applicable, and these patches can only be discarded from the data set. These discarded patches may affect the evaluation results to some extent, resulting in selection bias. When we analyzed the patches generated by the five repair tools to Defects4j, we found that only a small part of the fixes are in different methods, and most of the fixes are in the same method. Therefore, we believe that this threat is not serious. Compared with the entire dataset, the discarded data are very few, and the evaluation effects of these patches are unlikely to change the overall evaluation results.

6. Conclusions

This paper proposes a patch quality evaluation method based on semantic distance. Interval distance, output coverage, and path matching features are used to measure the semantic distance of patches. This method was verified on 279 patches generated by existing repair tools, and it successfully arranged the correct patches before the plausible but incorrect patches. At the same time, we compared our evaluation method with the existing evaluation methods and judged the evaluation ability of each feature. This showed that it is very effective for judging the quality of the patch generated by the repair tool from the semantic perspective by using the most basic features of the program. We hope to obtain more comprehensive evaluation features based on this in the future, as well as further expand the existing work.

Author Contributions: Conceptualization, Y.D. and M.W. (Meng Wu); methodology, Y.D. and M.W. (Meng Wu); software, M.W. (Meng Wu) and L.Z.; validation, M.W. (Mengying Wu), W.Y. and L.Z.; formal analysis, L.Z. and W.Y.; investigation, M.W. (Meng Wu); resources, Y.D.; data curation, H.L.; writing—original draft preparation, Y.D. and M.W. (Meng Wu); writing—review and editing, Y.D. and M.W. (Meng Wu); visualization, M.W. (Mengying Wu); supervision, M.W. (Meng Wu); project administration, Y.D.; funding acquisition, Y.D. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the National Natural Science Foundation of China (61873281) and the Fundamental Research Funds for the Central University under grant No.19CX02028A and No.20CX05016A.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

References

1. Gazzola, L.; Micucci, D.; Mariani, L. Automatic Software Repair: A Survey. *IEEE Trans. Softw. Eng.* **2019**, *45*, 34–67. [CrossRef]
2. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [CrossRef]
3. Le Goues, C.; Dewey-Vogt, M.; Forrest, S.; Weimer, W. A Systematic Study of Automated Program Repair: Fixing 55 out of 105 Bugs for $8 Each. In Proceedings of the International Conference on Software Engineering, Zurich, Switzerland, 2–9 June 2012.
4. Wang, S.; Wen, M.; Chen, L.; Yi, X.; Mao, X. How Different Is It Between Machine-Generated and Developer-Provided Patches? An Empirical Study on the Correct Patches Generated by Automated Program Repair Techniques. In Proceedings of the 13th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, Porto de Galinhas, Brazil, 19–20 September 2019.

5. Qi, X.; Steven, R. Identifying Test-Suite-Overfitted Patches through Test Case Generation. In *ISSTA*; ACM: New York, NY, USA, 2017. [CrossRef]
6. Le, D.X.; Bao, L.; Lo, D.; Xia, X.; Li, S. On Reliability of Patch Correctness Assessment. In Proceedings of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019.
7. Le, X.B.D.; Lo, D.; Goues, C.L. History Driven Program Repair. In Proceedings of the IEEE International Conference on Software Analysis, Suita, Japan, 14–18 March 2016.
8. Long, F.; Rinard, M. Automatic Patch Generation by Learning Correct Code. *ACM Sigplan Notices* **2016**, *51*, 298–312. [CrossRef]
9. 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]
10. Xiong, Y.; Liu, X.; Zeng, M.; Zhang, L.; Huang, G. Identifying Patch Correctness in Test-based Program Repair. In *Proceedings of the International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018*; ACM: New York, NY, USA, 2018. [CrossRef]
11. Xuan, J.; Martinez, M.; Demarco, F.; Clement, 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]
12. DeMarco, F.; Xuan, J.; Le Berre, D.; Monperrus, M. Automatic Repair of Buggy If Conditions and Missing Preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis, CSTVA 2014, Hyderabad, India, 31 May 2014*; ACM: New York, NY, USA, 2014. [CrossRef]
13. Durieux, T.; Monperrus, M. DynaMoth: Dynamic Code Synthesis for Automatic Program Repair. In *Proceedings of the International Workshop on Automation of Software Test*; ACM: New York, NY, USA, 2016. [CrossRef]
14. Xiong, Y.; Wang, J.; Yan, R.; Zhang, J.; Han, S.; Huang, G.; Zhang, L. Precise Condition Synthesis for Program Repair. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017.
15. Matias, M.; Martin, M. ASTOR: A Program Repair Library for Java. In *Proceedings of the International Symposium on Software Testing & Analysis, Saarbrücken, Germany, 18–20 July 2016*; ACM: New York, NY, USA, 2016.
16. 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 (ICSE), Gothenburg, Sweden, 27 May–3 June 2018.
17. Saha, R.K.; Lyu, Y.; Yoshida, H.; Prasad, M.R. Elixir: Effective Object-oriented Program Repair. In Proceedings of the 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), Urbana, IL, USA, 30 October–3 November 2017; pp. 648–659.
18. Xin, Q.; Reiss, S.P. Leveraging Syntax-related Code for Automated Program Repair. In Proceedings of the IEEE/ACM International Conference on Automated Software Engineering, Urbana, IL, USA, 30 October–3 November 2017; pp. 660–670.
19. Nguyen, H.D.; Qi, D.; Roychoudhury, A.; Chandra, S. SemFix: Program Repair via Semantic Analysis. In Proceedings of the International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013.
20. Jiang, J.; Xiong, Y.; Zhang, H.; Gao, Q.; Chen, X. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the International Symposium on Software Testing & Analysis*; ACM: New York, NY, USA, 2018; pp. 298–309.
21. Lou, Y.; Ghanbari, A.; Li, X.; Zhang, L.; Zhang, H.; Hao, D.; Zhang, L. Can Automated Program Repair Refine Fault Localization? A Unified Debugging Approach. In *ISSTA'20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*; ACM: New York, NY, USA, 2020; pp. 75–87.
22. Qi, Z.; Long, F.; Achour, S.; Rinard, M. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 12–17 July 2015*; ACM: New York, NY, USA, 2015; pp. 24–36.
23. Tao, Y.; Kim, J.; Kim, S.; Xu, C. Automatically Generated Patches as Debugging Aids: A Human Study. In *FSE 2014: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*; ACM: New York, NY, USA, 2014; pp. 64–74.

24. javalang. Available online: https://github.com/c2nes/javalang (accessed on 30 March 2020).
25. Martinez, M.; Durieux, T.; Sommerard, R.; Xuan, J.; Monperrus, M. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *Empir. Softw. Eng.* **2016**, *22*, 1936–1964. [CrossRef]
26. Smith, E.K.; Barr, E.T.; Le Goues, C.; Brun, Y. Is the Cure Worse Than the Disease? Overfitting in Automated Program Repair. In *ESEC/FSE 2015: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*; ACM: New York, NY, USA, 2015; pp. 532–543.
27. Tan, S.H.; Yoshida, H.; Prasad, M.R.; Roychoudhury, A. Anti-patterns in Search-Based Program Repair. In *FSE 2016: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*; ACM: New York, NY, USA, 2016; pp. 727–738.
28. Thomas, D.; Martin, M. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Ph.D. Thesis, Universite Lille 1, Villeneuve-d'Ascq, France, 2016.
29. Claire, L.G.; Neal, H.; Edward, K.S.; Yuriy, B.; Premkumar, D.; Stephanie, F.; Westley, W. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Softw. Eng.* **2015**, *41*, 1236–1256.