symmetry

MDPI

Article

Automated Program-Semantic Defect Repair and False-Positive Elimination without Side Effects

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

College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China; Z19070059@s.upc.edu.cn (M.W.); pangsc@upc.edu.cn (S.P.); s18070018@s.upc.edu.cn (L.Z.); z18070040@s.upc.edu.cn (W.Y.); z19070061@s.upc.edu.cn (M.W.); z19070058@s.upc.edu.cn (H.L.)

* Correspondence: dongyk@upc.edu.cn

Received: 22 November 2020; Accepted: 7 December 2020; Published: 14 December 2020

Abstract: The alarms of the program-semantic defect-detection report based on static analysis include defects and false positives. The repair of defects and the elimination of false positives are time-consuming and laborious, and new defects may be introduced in the process. To solve these problems, the safe constraints interval of related variables and methods are proposed for the semantic defects in the program, and proposes a functionally equivalent no-side-effect program-semantic defect repair and false-positive elimination strategy based on the test-equivalence theory. This paper realizes the automatic repair of the typical semantic defects of Java programs and the automatic elimination of false positives by adding safe constraint patches. After the repair, the program functions are equivalent and the status of each program point is within the safety range, so that the functions before and after the defect repair are consistent, and the functions and semantics before and after the false positives are eliminated. We have evaluated our approach by repairing 5 projects; our results show that the repair strategy does not require manual confirmation of alarms, automated repair of the program effectively, shortened the repair time greatly, and ensured the correctness of the program after the repair.

Keywords: automated program repair; false-positive elimination; program-semantic defect

1. Introduction

Automated Program Repair is a rapidly developing field and has been gradually drawing increasing attention from people. It is a common understanding in academia-industry to eliminate program defects as soon as possible. The process of automatic repair includes defect location, candidate patch generation and patch validation. The program repair method obtains the suspected error locations through the defect location technology, and generates the candidate patches. Finally, the patched program is verified by the test set, and it is manually judged whether to accept the repair patches. Program defects are by-products of the software development process that cannot be eliminated. According to statistics, the program contains an average of 3–10 defects per thousand lines of code after compilation and traditional software tests. Static analysis tools are effective means to detect program bugs; alarms are detected by current static analysis tools that include real defects and false positives. The false-positive rate is between 35–91%, and the confirmation of the alarm and the repair of the defect requires a lot of manpower. According to the repair strategy, automatic repair is divided into two categories. One is semantic-based program automatic repair [1–4], and the other is search-based program automatic repair [5–9]. Nguyen et al. [1] proposed SemFix, which uses Tarantula technology to detect defects and uses program synthesis to generate repair statements. However, SemFix is not enough to repair multiple lines of code errors. Le et al. [5] proposed that GenProg uses a

test suite to encode defects. The results show that it fixes 55 of 105 defects, and Qi et al. [6] found that only two of the 55 patches is semantically correct.

A large number of false positives will appear in the defect report generated by the defect-detection tool. High false-positive rate will consume a lot of energy in manual confirmation, and increase the probability of introducing new errors when eliminating false positives, thus greatly increasing the difficulty of automated program repair. Alarms include bugs and false positives. For real bugs, we want the repair tools to automatically repair the program without human involvement and achieve the desired function. For false positives, it is hoped that the defect-detection tools can ignore them, then smoothly execute the following report without changing the function and semantics of statements related to false positives. False alarms can cause incorrect repairs in defect-detection tool [10], which will bring unnecessary semantic effects and unexpected semantic changes, making the program unable to complete the specified behavior within the specified time. At the same time, after repairing false positives, the execution path of program may be changed, which makes the program violate the initial expectation of developers. New bugs may also be introduced during the repair process, resulting in secondary rework by developers, increasing the development time significantly and delaying the project's final completion date.

To avoid manual determination of alarms, realizing the full automation of manual repair, we propose a strategy of automatic program-semantic defect repair and false positives [11] elimination without side effects. Based on the safe constraint mechanism, semantic defect patches of Java programs are generated, and DTSFix, a program defect repair tool without side effects, is developed. The repair tool does not need to manually confirm defect report, and complete repair before the real defect triggers. For false positives, the tool can be successfully implemented to eliminate false positives without manual confirmation. The results show that the success rate of the tool for alarm repair is 58.13%, which can correct the real defects, reduce the risk of false positives, and realize the full automation of manual repair.

The second section of this paper introduces the research progress of automatic repair methods. describes program-semantic defect, Section 3 specifies the relevant states specifications of program points, and concretizes program point states in combination with the control flow diagram. Section 4 summarizes the impact of side effects [6,12] and proposes automatic program repair without side effects. Section 5 describes patch synthesis based on safe constraints. Section 6 uses repair methods without side effect to repair semantic defects in five projects, and evaluate the performance of the method for real defect and false-positive repair. Section 7 summarizes the article.

2. Related Work

In recent years, a lot of work has been done on the automated program repair. According to repair strategy, automated program repair can be divided into two categories, one is semantic-based automated program repair, and the other is search-based automated program repair.

2.1. Semantic-Based Program Automatic Repair

Semantic-based repair methods is generally through the symbolic execution and constraint solving directly synthetic patches. Nguyen et al. [1] proposed SemFix, using Tarantula technology for defect location, get suspicious statement list, derives repair constraint from a set of tests, the repair statement is generated by program synthesis, then the benchmark program is extracted from SIR [2] for verification, finally complete the repair. SemFix applies repairs on a smaller scale, and it is not big enough to repair multi-line code errors. Moreover, there are a lot of potentially error patches that identify the patches that pass tests suite as correct. Mechtaev et al. [3] proposed Angelix, which can handle larger programs. The test case set drives the controllable symbolic execution technology to collect path conditions, and the program specification is from the lightweight repair constraint Angel Forest. Compared with SemFix, the method of symbolic execution more lightweight. Bader et al. [4] proposes Getafix learning from the past human written submission repair templates, recommending

the top patch to developers. However, its disadvantage is that the bug categories that need to create context are far away from the bug code segments, resulting in many semantic changes for patches that are not the desired semantics, and the repaired program still has a large number of defects and false positives.

2.2. Search-Based Program Automatic Repair

Based on the search-based repair technology, the test set is directly used as the program specification to determine the patch quality in the candidate patch search space. The goal is to output the patches that pass all the test cases. Le et al. [5] proposed GenProg that using the extended form of genetic programming to develop program variant, encoding required functions by the existing tests suites. The experimental results showed that it repaired 55 of 105 defects. Qi et al. [6] verified the patches generated by GenProg manually, only two of the 55 patches were correct semantically and the success rate of repair was not high. Xiong Yingfei et al. [7] proposed ACS with more fine-grained determination of defect-related variables and predicates, which concentrated on condition synthesis to generate patches. The dependency of variables is used to determine defect-related variables and the API documentation is used to further filter variables. Mining technology was used to predicate sort, ACS makes statistics on the available open-source code. Discover the correlation information between variables and the operations applied to them to generate the correct patch. Ming et al. [8] proposed CapGen, a more fine-granularity context-aware patch generation technique based on Abstract Syntax Tree. Merger context information to distinguish the mutation operator priority. The experiment results show that the accuracy can be increased to 84%, the ability to correct patch sorting accuracy at 98.78%. Hua et al. [9] proposed SketchFix, still works on a fine-granularity at the AST node level, using Ochiai [13] spectrum-based for fault location, ordering of the suspicious statements with dubious value. EdSketch [14] is used to backtrace the search for candidate fixes to fill the holes, turning the buggy program into a fine-grained sketch with holes. Until a candidate patch is found that meets all tests. SketchFix prioritizes patterns that introduce minor changes to the original program. In general, search-based repair methods fail to produce high-quality patch candidates, partly because the search space may not contain the correct patch, or partly because the search space is so large that candidate patches cannot be found within the time-limited valve.

In view of the low quality, high false-positive rate and large number of suspected correct patches generated by the existing work, we propose a method without side effects for semantic defects, and generated patches through the safe constraint rules of relevant expressions. The main focus is on how to avoid programs migrating to wrong states, fixing defects and eliminating false positives successfully.

3. Program-Semantic Defect

This paper mainly studies the most typical and common semantic defect repairs related to program state, such as Null Pointer Dereference (NPD), Illegal Calculation (IAO), Resource Leak (RL), Uninitialized variable (UVF), Out of Bound (OOB) and so on.

There is an example of a null pointer dereference, it is taken from the BorderArrangement.java file of the open-source project JFreeChart.

Figure 1 is an example of a null pointer dereference. The L1 line declares that *content* may be empty after the L3 line, and it dereferenced directly on lines L5 and L6 causing NPD.

To fix defects and eliminate false positives more accurately, it is necessary to further deepen the understanding of program. Below we will describe the program symbolically. A program **P** can be represented by a six-tuple <V, S, L, $\rightarrow$, *entry*, *exit*>. V represents a set of program variables, S represents program statements, L represents program points, $\rightarrow$ represents program migration status, *entry* represents the entry of program, and *exit* represents the exit of program.

```
L1:   Size2D content = null;
L2:   if (h == LengthConstraint.NONE) {
L3:      content = arrange (container, g2);
L4:   }
L5:    return new Size2D(Width(content.getWidth()),
L6:         Height(content.getHeight()))
L7:    }
```

Figure 1. Example of a null pointer dereference false positive.

It is reliable to represent the migration of each state under the program-semantic system of abstract interpretation. The description of safe constraints and program state is given below.

Definition 1. ***(Control Flow Graph CFG):*** *It is a directed graph that reflects the logic control flow of a program. The possible flow of block execution CFG is a directed graph* $G = \{N, E, entry, exit\}$. *N represents the node sets (a basic block corresponds to a node in the graph), and the edge sets* $E = \{< n_i, n_j > | n_i, n_j \in N, n_i$ *may be executed immediately after* $n_j\}$, *entry and exit represent the program's unique entry and exit, respectively.*

Each statement in program represents a node in CFG, and the nodes represents a linear block codes with no jumps or jump targets. Program points are different from nodes. There are program points before and after nodes in control flow graph, and the edge from one node to another can be represented as a path.

Definition 2. ***(Path):*** *Paths can be represented by node-to-node edge sets in the CFG,*$< n_i, n_j >$ *represents the path from node* n_i *to node* n_j, $\bigcup < n_i, n_j >$ *represents the set of paths from node* n_i *to node* n_j.

For example, in Figure 1, L2 to L3 is a true branch path judged by **if**, which is expressed as $< n_2, n_3 >$. The path of the L5 to L6, expressed as $< n_5, n_6 >$. The set of paths from L2 to L6 is expressed as $\bigcup < n_2, n_6 >= \{< n_2, n_3 > || < n_2, n_4 > \cup < n_4, n_5 > \cup < n_5, n_6 >\}$.

Definition 3. ***(Program State):*** *program state* $\sigma : Var \to \mathbb{Z}$ *is a function from program variable to value. The set of active variables at program point l is* α_v^l, $l \in L$, β_v = <v,α_v^l> *represents the binary of active variables v and the variable value* α_v^l, σ_l:β_{v1}, β_{v2},$\cdots$ β_{vn} *represents the states of program point l , i.e., the set of all the active variables and their values at the program point.*

A program has one start state, one error state, and several intermediate states. σ_{entry} and while σ_{med} represents the intermediate state. Except for σ_{error}, the rest of the program states can be called security state σ_{safe}, i.e., the program in a safe state does not have defects. The start state of program $\sigma_{entry} \in \sigma_{safe}$. For example, the possible state of a program at the program point L1 in the example above is $\sigma_1 : \{\sigma_{entry} : content \in R\}, \sigma_l \in \sigma_{safe}$. The automatic detection tool reports an error (which is actually a false positive) at the point L7, the possible program state is $\sigma_7 : \{\sigma_{error} : content = null\}$.

Definition 4. ***(Safe Constraint):*** *We propose safe constraint with help of the definition of domain. Defines the safe range and constraints of a variable, function, or expression, that is, a limited set of constraints on them in a program. Variable values in the safe constraint means that the current value of the variable meaningless.* R_v^l *represents safe constraints at program point l active variables v.* $\alpha_v^l \subseteq V$ *if* $\exists\ v \in \alpha_v^l, \alpha_v^l \cap R\_v^l \neq \phi$, *i.e., if the set of values and the set of security constraints do not intersect, the program will not generate illegal memory access defects [15]. Conversely,* $\exists v \in \alpha_v^l, \alpha_v^l \cap R_v^l = \phi$, *i.e., the set of values and the set of security constraints do not intersect, the program will cause a defect at l.*

For example, in the above false-positive example, the safe constraint of the variable content in line L6 is $R_{content}^{l6}$ = {*content* | *content* = *null*} , and the value set is $\alpha_{content}^{l6}$ = {*content* | *content*$\neq$ *null* }. In line

L5, Add non-empty assertions to *content*, the set of value variable $\alpha^{16}_{content} \cap R^{16}_{content} = \phi$ in the L6 line, i.e., it is impossible to take the value to be null.

Definition 5. ***Definition 5(Program-Semantic Defect):*** *Program-semantic defect is the program state does not satisfy the requirements of the security attribute of the program, causing abnormal execution of the program, and even causing abnormal termination of the program. represents a program-semantic defect occurring at the program point l, if variable $v \in V, \alpha^l_v \cap R^l_v \neq \phi$, the program will migrate to σ_{error}, and the defect-detection tool will report a defect at l, i.e., if value of variable intersects with safe constraint of variable, defect will be caused. For example, NPD will triggered when the defect-related variable is null before dereferenced.*

4. Automatic Program Repair without Side Effects

The existence of defects will affect program quality seriously, but the existing program automatic repair methods and tools [16–27] cannot guarantee that the repaired program is correct. For example, the automatic repair tool GenProg [5] repairs the defect in Figure 2 mistakenly.

Figure 2a is a buggy program, and Figure 2b is side-effect repair from GenProg. In Figure 2a, when $getshort(icode, pc+1) >= String.length || getshort(icode, pc+1) < 0$, line 206 will report *array index out of bounds*. Genprog updated line 206 to $ars = ((Scriptable)ars).getDfalutValue(null)$. If *ars* is not defined, it will be assigned the default value of Scriptable class. However. The original program needs to get the corresponding subscript value of String array, so the modified semantics is inconsistent with the target semantics of the original program. This repair has side effects. Side effects not only fail to achieve goal of automatic repair, but also increase amount of manual confirmation of patches.

```
203  if(ars==MRK)
204      ars=null;
205  if(ars==underfined){
206      ars=strings[getshort(icode,pc+1)];
207  }
208  free(a);
```

(**a**) buggy program

```
203  if(ars==MRK)
204      ars=null;
205  if(ars==underfined){
206   +  ars=((Scriptable)ars).getDfalutValue(null);
207  }
208  free(a);
```

(**b**) side-effect repair from GenProg

Figure 2. GenProg repair defect instance.

We summarized four situation of side effects as below:

- A candidate patch can pass all test cases, but they are not similar to the developer-provided fix [12] semantically, (i.e., plausible but incorrect).
- introducing new bugs after repair.
- The fix patterns are too specific or too abstract cause false positives and omission [6] respectively.
- Patches are different semantics or syntax styles from the correct patch submitted by the developers.

The fix method uses conditional constraints extracted from test sets as program specification that determines the quality of the candidate patches, resulting in semantic bias and producing a large number of suspected correct patches. However, due to the lack of distinction in the program specification, the correct patch cannot be generated, and the potential error patches cannot be further

distinguished. Particularly complex repair operations are more likely to modify correct semantics of the original program, increasing the probability of introducing new bugs.

Figure 2a was repaired with no side effect method, and the results are shown in Figure 3. In Figure 3, the value space of *getshort(icode,pc+1)* is limited. The safe constraint of *getshort()* is $R^{206}_{getshort()} = \{getshort()|0 \leq getshort() < String.length\}$. Following the path <206,207> through the **if** statement, the state on line 206 is $\sigma_{206} : \{\sigma_{other} : 0 \leq getshort() < String.length\}$, which value is within the safe constraint, so $\sigma_{206} \in \sigma_{safe}$, Does not migrate to $\sigma_{error} : \{getshort() \geq String.length\}$. The program does not violate the semantics of the original program because it still gets the same value as the original program expected at line 208, so Figure 3 is correct repair. Moreover, the execution function before and after repair is consistent, and the state change is within the safe range of equal program semantics, which is a repair without side effect.

```
203   if(ars==MRK)
204       ars=null;
205   if(ars==underfined){
206   +   getshort(icode,pc+1)<string.length && getshort(icode,pc+1)>=0;
207       ars=strings[getshort(icode,pc+1)];
208   }
209   free(a);
```

Figure 3. Program repair without side effects.

The value space of $getshort(icode, pc + 1)$ is limited in Figure 3, The safe constraint of *getshort()* is $R^{206}_{getshort()} = \{getshort()|0 \leq getshort() < String.length\}$. Following the path <206,207> through the if statement, the state on line 206 is $\sigma_{206} : \{\sigma_{other} : 0 \leq getshort() < String.length\}$, which is within the safe constraint, so $\sigma_{206} \in \sigma_{safe}$. In order to avoid the program migrating to $\sigma_{error} : \{getshort() \geq String.length\}$, and the program does not violate the semantics of the original program because it still gets the same value as the original program expected at line 208, so Figure 3 is correct repair. Moreover, the execution function before and after the repair is consistent, and the state change is within the safe range, which is a repair without side effect.

We used test set-based test equivalence [28] to verify the repair effect. If two programs have the same result for the same test sets, they are test-equivalence relations based on the test sets.

Definition 6. ***(Equivalent Relations Test):*** $\prod$ *represents a set of procedures, T represents a set of tests,* $\forall p_1, p_2 \in \prod$ *if* $p_1 \overset{T}{\longleftrightarrow} p_2$ *, then* p_1*,* p_2 *either both pass or fail T, at this time, a set of equivalence relations about T (reflexive, symmetrical, transitive)* $\overset{T}{\longleftrightarrow} \prod \times \prod$ *is test-equivalence.*

Consider three different programs in Figure 4, and insert the highlighted statements into different locations in the program corresponding to Figure 4a,b,c. Our algorithm determines that Figure 4a is test-equivalence to Figure 4b. The test equivalent relationship is $\overset{T}{\longleftrightarrow}_{value}$. First, they only during test execution on the right side of the assignment. The highlighted statements take the same value during the test execution. Secondly, it can be determined that Figure 4b,c are also test-equivalence relations. Because they insert the same assignment at different program locations, both locations are executed by the test, because the variables *b* and *a* are not overwritten/modified during test execution. We call this kind of test equivalent relationship $\overset{T}{\longleftrightarrow}_{deps}$ Finally, merge the two analysis results, according to transitivity Figure 4a,c are test-equivalence relations.

a=0;
b=a;
if(dec) {
clear.bufs();
}
(**a**) before **if** statement
	a=0;
	b=0;
	if(dec) {
	clear.bufs();
	}
	(**b**) before **if** statement
		a=0;
		if(dec){
		b=a;
		clear.bufs();
		}
		(**c**) in **if** statement

Figure 4. Three different repairments to a program.

Test-equivalence relations can be divided into two categories: value-based test-equivalence $\overset{T}{\longleftrightarrow}_{value}$; dependency-based test-equivalence relation $\overset{T}{\longleftrightarrow}_{deps}$. We only care about the former in this paper.

Definition 7. ***(Value-Based Test-Equivalence Relation*** $\overset{T}{\longleftrightarrow}_{value}$***):*** *Two programs p and* p' *differ only in expressions, during the executions of p and* p' *with test* T *, the expressions e and* e' *are evaluated into the same values. The relationship between p and* p' *is value-based test-equivalence, denoted asp* $\overset{T}{\longleftrightarrow}_{value}$ p'*. We use definition function* $\downarrow_{value}$ *containing a set of expressions to represent the value-based test-equivalence relation:*

$$< p, \sigma_{in}, \{e, e^{'}\} > \downarrow_{value} < \sigma_{out}, \{e, e^{'}\} >$$

σ_{in} *represents input state and* σ_{out} *represents output state.*

To get a clearer understanding of value-based test-equivalence relation, the following examples are given. Consider a program p1 defined as:

$$\{\textbf{if} \quad (x > 0) \quad \textbf{then} \quad x = y;\}$$

A program p2 defined as:

$$\{\textbf{if} \quad (y = 2) \quad \textbf{then} \quad x = y;\}$$

And a test $t : x \to 1, y \to 2$, These programs are value-based test-equivalence relation for the test t, since they differ only in the *if*-condition and the following relation:

$$< p_1, \sigma_{in}, \{"x > 0", "y = 2"\} > \downarrow_{value} < \sigma_{out}, \{"x > 0", "y = 2"\} >$$

l_1 where the input state $\sigma_{in} : x = 1, y = 2$ and the output state $\sigma_{out} : x = 2, y = 2$.

Based on the above equivalence relation, we will verify effect of repair without side effects. Due to the different repair strategies for real defects and false positives. The following is a description of repair method without side effects.

- When real defect is repaired and any tests cannot trigger defects, if the same input has the same output, there is no difference in function. The specific description is:

$$p(input) \to p(output), p^{'}(input) \to p^{'}(output) \&\& p(output) = p'(output) \tag{1}$$

  p and $p^{'}$ are original program and patched program, respectively, i.e., the same input has the same output. Meanwhile, test-equivalence relation is satisfied, $< p \overset{T}{\longleftrightarrow} \downarrow_{value} p^{'} >$.
- When a real defect is repaired and a test triggers defects, the test case caused an error in the program before repair, and the error of executing the same test no longer occurs after repair.

$$p(test_i) \to is\ happen, p^{'}(test_i) \to is\ not\ happen \tag{2}$$

On the basis that defects do not occur again, $< p \overset{T}{\longleftrightarrow}\downarrow_{value} p^{'} >$ is satisfied.

- When eliminate false positives, tools report defects before repair, and they are no longer considered to be defects by after repairing.

First, The elimination of false positives first satisfies Equation (1) and equivalence relation of test, secondly it satisfies semantic consistency. Semantic consistency in this paper means that state of the corresponding program point is same before and after repair. If two programs are test-equivalence and semantic consistency, this paper is considered equivalent as a program equivalence. The statement changes are recorded. After repair, program point closest to defect line that does not change is monitoring point $a^{'}$, and the corresponding position of this point is a; the program point closest to defect line that does not change after statement is monitoring point $b^{'}$. Similarly, the position of b is determined. σ represents states of before repair, and $\sigma^{'}$ represents states after repair, comparing states of two monitoring points, if $\sigma_i = \sigma_{i'}^{'}$ && $\sigma_j = \sigma_{j'}^{'}$, after patched, does not change monitoring point state. Program records the execution path from σ_{entry} along data flow $< n_i, n_j >, 0 \leq i < j \leq p.length$, analysis point state. On the basis that the semantics of the monitoring point have not changed, if $\bigcup < n_a, n_b >= \bigcup < n_{a'}, n_{b'} >$, there is no semantic difference. At the same time, $< p \overset{\prod}{\longleftrightarrow}\downarrow_{value} p^{'} >$, we call this repair no side effect of false positives.

Algorithm 1 describes the functional equivalent verification algorithm for false positives elimination.it first verifies whether functions are consistent before and after repair. If not, it does not satisfy functional equivalence and program equivalence. Otherwise, the control flow graph continues to be generated. Executing along the path of control flow from start state. Each execution updates state of relevant program points until end of program. Ensure that next node of path execution and program state at that node is consistent. If σ_{exit} is consistent and the function is equivalent before and after repair, it is an automated program repair without side effects.

Algorithm 1. The equivalent verification algorithm of false positives elimination function without side effects

Input: Programs before and after repair
Output: Program equivalent or not
1 $p(input) \rightarrow p(output), p^{'}(input) \rightarrow p^{'}(output)$;
2 **if** (p(output)! = $p^{'}$(output)) **then**
3 **return**;
4 Generate the corresponding control flow graph
5 **if** $< p \overset{\prod}{\longleftrightarrow}\downarrow_{value} p^{'} >$
6 **while** (n! = exit) **do**
7 **if** (σ_i == σ_a && σ_{i+1}! = σ_a) **then**
8 k = a;
9 **if** (σ_i!=σ_b && σ_{i+1} == σ_b) **then**
10 m = b;
11 **while** (n! = exit) **do**
12 **if** ($\sigma_m = \sigma_{m'}^{'}$ && $\sigma_k = \sigma_{k'}^{'}$) **then**
13 **if** $\bigcup < n_a, n_b >= \bigcup < n_{a'}, n_{b'} >$ **then**
14 flag = true;
15 flag = false;
16 $\sigma_j, \sigma_j^{'} \leftarrow$ update along the data stream
17 **if** (flag == true) **then**
18 **return** true;
19 **return** false;

5. Defect Repair and False-Positive Elimination Based on Safe Constraints

To achieve the goal of automated program repair without side effects, we propose automatic repair based on safe constraints. In this section, we formalize defect. On this basis, we combine safe constraints to synthetic repair conditions and add security transfer statements to repair program.

In this paper, automated program repair method based on safe constraints is proposed. An instance of a defect state machine is created according to the defect characteristics, and then a state migration operation is performed on the defect state machine. Our fix goal is to add statements so that program state is never migrated to σ_{error}. $\sigma_{safe} = \neg\sigma_{error}$, $\alpha_v^l \cap R_v^l = \phi$. If value of variable does not intersect with the restricted interval, it will not cause bugs, defect-detection tool also does not detect an alarm. Repair methods based on safe constraints are more granular and can avoid many suspected correct patches. The resulting patches are more targeted.

When patch is synthesized, defect reports of static detection tool is read to determine the defect-related variables. The synthesis conditions are determined according to safe constraint rules in Table 1. Then, safe patches are generated according to synthesis conditions, as shown in Algorithm 2.

Table 1. Security Restricted rule.

Defect	Exp	Related Operation	Safe Constraint	Synthesis Condition
NPD	e_1	e_1.f()	$R_{e_1}^l = \{e_1 \| e_1 = null\}$	e_1! = null
NPD	e_1, e_2	/,/=, %, div(), fmod()	$R_{e_2}^l = \{e_2 \| e_2 = 0\}$	e_2! = 0
IAO	e_1	log(),log10(),sqrt(), _logb(),_y0()	$R_{e_1}^l = \{e_1 \| e_1 < null\}$	e_1 >= 0
IAO	e_1	asin(),acos()	$R_{e_1}^l = \{e_1 \| [Min, 0] \| \| [1, Max]\}$	$e_1 \geq 0 \| \| e_1 \leq 1$
IAO	e_2	atan2()	$R_{e_2}^l = \{e_2 \| e_2 = 0\}$	e_2! = 0
OOB	e_1	array[e_1]	$R_{e_1}^l = \{e_1 \| e_1 \geq array.length()\}$	e_1 < array.length()
UVF	e_1	variables uninitialization before use	$R_{e_1}^l = \{e_1 \| e_1 = null \| \| e_1 = 0\}$	e_1 = new classname of e_1()
RL	e_1	resource unclosed after use	$R_{e_1}^l = \{e_1 \| !e_1.close()\}$	e_1.close()

Algorithm 2. Patch Synthesis Algorithm Based on Safe Constraints

Input: Defect file,Defect report
Output: Patch for defect program
1 **ResultSet** ← Defect Information from Defect Report order by File
2 Sec_{con} ← ResultSet// D_{file}: Defect file; D_{report}: Defect report; Sec_{con}: Safe constraint.
3 **while** (D_{file}.*hasDefect()*) **do**{
4 D_{var} ← D_{report};//D_{var}: Defect-related variables;
5 Sec_{rule} ← D_{var};// Sec_{rule} : Safe constraint rules;
6 Pat_{con} ← Sec_{rule};//Pat_{con}: Patch synthesis conditions
7 }
8 **return** patch;

The repair for NPD is shown in Table 2. We determine the state at row where defect occurs, Defect occurs in *Class c = (Class) stream.readObject()*, the state is σ_l:{ $\sigma_{may-null}$:*c = null || c! = null* }. It is found that the point state is $\sigma_{may-null}$. If the reference is dereferenced at this point, it will migrate to σ_{error}. To avoid the state migration to σ_{error}:{*content = null*}, we need to add a safe transition condition. Then we determine the defect correlation variable by defect report, it is *c*. The defect correlation operation is a reference to a potentially empty variable, *c* corresponds to e_1 in the security constraint rule in Table 1.$R_{e_1}^l = \{e_1 | e_1 = null\}$. The repair synthesis condition is $e_1! = null$, then safe constraint of *c* is $R_c^l = c|c = null$, its synthetic condition is $c! = null$. Adding repair statement $if(c! = null)$ before the migration condition is satisfied, so that it does not satisfy conditions for migrating to σ_{error}.

Table 2. Comparison before and after repair for NPD.

Before	After

Class c = stream.readObject().getClass();
if (c.equals(Alpha.class)) {
System.out.println(c.getName());
}
	Class c = stream.readObject().getClass();
	+ if(c!=null) {
	if(c.equals(Alpha.class)) {
	System.out.println(c.getName());
	}
	+ }

6. Experiment

To verify whether the method in this paper can realize defect repair and false-positive elimination without side effect, DTSFix is implemented based on static detection tool DTS (Defect Test System) [29,30], DTSFix detect locations of suspicious statements by DTS, and produce defect reports [31]. Partial defect reports of JFreeChart detected by DTS are shown in Table 3.

The defect report produced by DTS contains defect type, defect ID, the file where defect is located, variables associated with defects, start line of defect, occurrence line of defect, and description of defect.

Table 3. Partial defect report of JFreeChart project detected by DTS.

Defect	Type	Id	File	Var	IPLine	Description
fault	NPD	89	D:\workspacejava\jfreechart\src\main\java\org\jfree\chart\block\BorderArrangement.java	con	174	Variable 'con' declared at line 136 may cause a null pointer exception at line 174.
fault	IAO	37	D:\workspacejava\jfreechart\src\main\java\org\jfree\chart\axis\Arithmic.java	Math.log	312	There is an illegal operation in the expression 'Math.log' on line 312.
fault	RL	31	D:\workspacejava\jfreechart\src\main\java\org\jfree\chart\ChartUtils.java	FileOut Stream	497	The resources allocated on line 497 may be leaked on line 497.
fault	NPD	22	D:\workspacejava\jfreechart\src\main\java\org\jfree\chart\renderer\category\BarRenderer.java	calcuBar	868	The array'calcuBar' appears in line '868' and returns NULL.

Figure 5 illustrates an overview of the DTSFix approach. First, defect reports and buggy program are input. Then, according to safe constraint rules, patch conditions are synthesized and inserted into candidate patch locations. Finally, candidate patches are generated, and candidate patches are retested by DTSFix. If a patch does not produce new bugs and the original defect has been repaired, it is considered to be a correct patch. Otherwise skip and validate next patch.

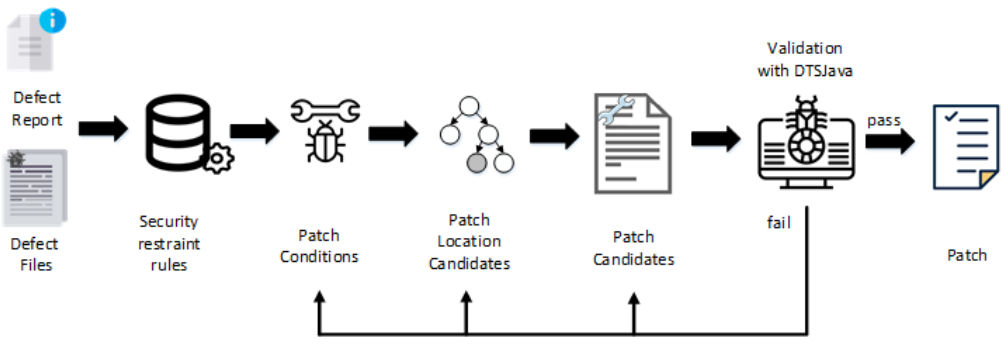

Figure 5. Repair framework of DTSFix.

Our evaluation aims to answer the following research questions:

- **Question 1: How effectively does DTSFix fix alarms?**
- **Question 2: How effective are the patches produced by DTSFix?**

To answer the two RQs, we use Defects4J dataset which is widely used as a benchmark for Java-targeted APR research. The dataset contains 393 bugs and their corresponding developer fixes. DTSFix attempts to fix 5 projects JFreeChart, commons-lang, Log4j, commons-math, commons-rhino and artificial confirmed the number of real defects, false positives and successful repair. The defect report of JFreeChart detected by DTS is shown in Table 3. Results showed that a total of 855 alarms were taken as experimental objects by the five project, and detailed data such as false alarm rate were also given in Table 4. Among them, the number of alarms was the sum of defects and false positives detected by DTS, and the number of successful fixes included repair of false positives and real defects.

Table 4. Comparison of experimental results.

Project	KLOC	Alarms	FP	RD	FPA	NS	Time
JFreeChart	96	68	17	51	25.00%	40	2.1 h
Log4j	27.8	156	93	63	59.62%	93	3.4 h
commons-math	121.2	280	178	102	63.57%	154	5.3 h
commons-lang	54	136	57	79	41.91%	72	2.8 h
commons-rhino	51	215	117	98	54.42%	138	4.6 h
Total	350	855	462	393	54.04%	497	17.0 h

Alarms is the number of alarms detected by DTS, including real defects and false positives. **FP** and **RD** are the number of false positives and real defects, respectively. **FPA** is the percentage of false positives in defect reports. **NS** is the number of successful repairs. DTSFix repaired 497 of 855 alarms correctly, successful repair rate is 58.13%. Manual validation time is 17.0 h. It can be seen that manual validation alarms take a lot of time, increasing workload of developers greatly and delaying progress of project seriously.

6.1. Performance in Repairing Alarms

To answer RQ1, we verify the correctness of the candidate patches for each bug. A patched program executes use cases that failed before, if it passed, regression testing is performed. If it also passed, we do not think the patch introduces any new bugs. Otherwise, skip and validate the next patch. DTSFix continues to generate and validate candidate patches over a specified period of time. After all patches have been validated, we collect plausible patches that can make the buggy program pass all test cases successfully, and then manually analyze patches to see if their semantics are equivalent to those submitted by developer. If semantics are same, the patch candidate will be considered to be a correct patch.

ID in Table 5 are obtained from defect reports generated by DTS, each of them is unique. DTSFix generated plausible patches for 213 bugs in total among all the 393 real bugs, successful repair of real bugs accounted for 52.16% of the total number of alarms. The details of some bugs are listed in Table 5, the total number of patches generated is affected by the size of error source files, and in some cases DTSFix generates multiple correct patches, for example, Math79 generates two correct patches and it needs to insert a statement to repair. However, both patches are the same as those submitted by developers, because the required statements can be inserted in two adjacent locations. **Time(s)** shows that the time required for DTSFix to generate correct patches is between 2.88 and 32.59 s. Table 5 shows that DTSFix generates three reasonable repairs for two defects.

Table 5. The performance of real defects repaired by DTSFix.

Project	ID	Correct Patch	Unreasonable Patch	Time(s)
JFreeChart	76	1	10	26.21
JFreeChart	89	1	16	28.36
Log4j	43	1	3	6.41
Log4j	56	1	8	15.69
commons-math	37	1	2	2.88
commons-math	79	2	9	8.24
commons-lang	62	1	8	6.59
commons-lang	61	3	6	3.85
commons-rhino	18	3	18	32.59
commons-rhino	52	1	4	6.72

Before repair in Figure 6 refers to the number of real bugs, and *after repair* is the number of real bugs confirmed manually by DTSFix, The number of real bugs before repair in the 5 projects is 393, and DTSFix is unable to repair the real defects of 47.84% (188/393), because 1/3 of the remaining 188 defects are defects with too complex calling relationship, such as pointer referring to the released complex data, and 2/3 are infinite loop.

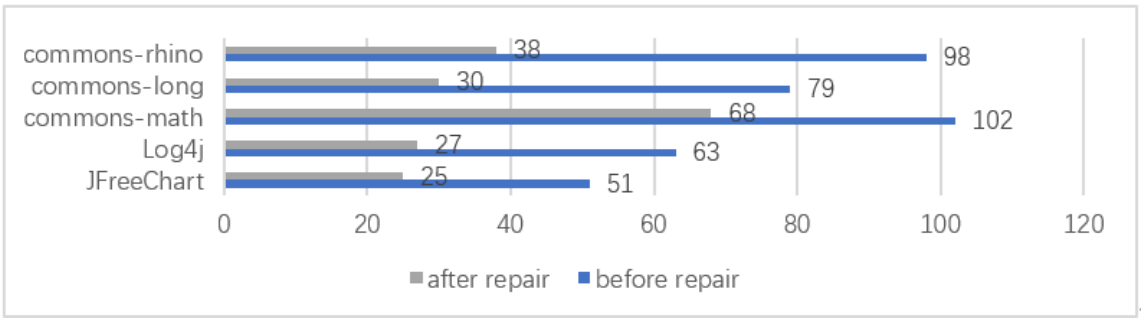

Figure 6. Comparison of the actual number of defects before and after repair.

To study the ability of DTSFix to repair false positives, we manually confirmed the effect of the repair. *before repair* in Figure 7 is the number of false positives before fixing, *after repair* is the number of false positives after fixing. DTSFix eliminates 381 false positives, there are 81 false positives not eliminated. This is because some false positives are beyond the scope of DTSFix , such as a pointer to complex data type reference is released. Successful false positives have been eliminated and manually checked, there are no side effects.

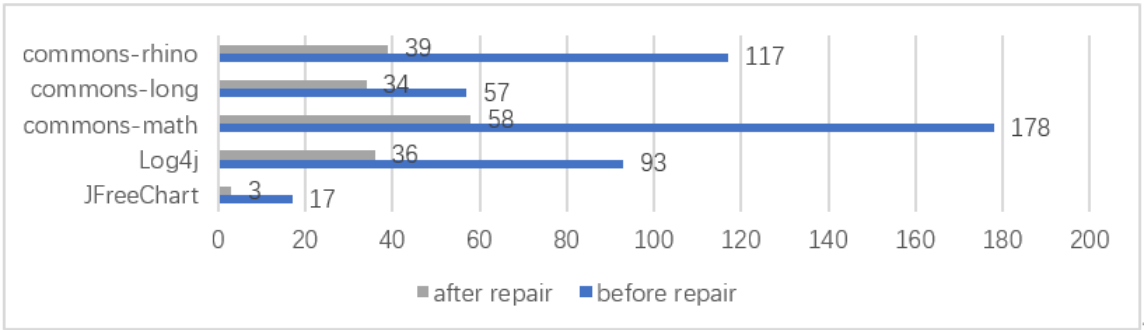

Figure 7. Comparison of the number of false positives before and after repair.

Table 6 is DTSFix false-positive elimination instance, defect report shows that $row * column$ may have illegal calculation on $used = 1/(row * column)$, whereas line 215 $if((row * column) > 0)$ determines that the expression cannot have illegal calculation on line 241. When DTSFix repairs,

the security constraint of $row * column$ on line 241 is $R^{l241}_{row*column} = \{row * column | row * column = 0\}$, adding repair statement $if((row * column)! = 0)$. The regression test cases are executed, this alert was not appear again in the defect report and no new bug was introduced, i.e., the repair is successful. Monitor point $L_a = L_{240}$, $L^{'}_a = L_{240}$, $L_b = L_{242}$, $L^{'}_b = L_{243}$.

The states of monitor point before repair:

σ_{240}:{ $\sigma_{other} : row * column > 0\}$, $\sigma_{242} : \{\sigma_{med} : used = space * Axis.getLowerMargin | 1/(row * column)\}$.

The states of monitor points after repair:

$\sigma^{'}_{240}$:{ $\sigma_{other} : row * column > 0\}$, $\sigma^{'}_{243} : \{\sigma_{med} : used = space * Axis.getLowerMargin | 1/(row * column)\}$.

Compare the states before and after repair, $\sigma_{240} = \sigma^{'}_{240}$ && $\sigma_{242} = \sigma^{'}_{243}$, so the fix does not change program semantics. We observed before and after repair program execution paths, it is not hard to see $\bigcup < n_{212}, n_{242} >= \bigcup < n_{212}, n_{243} >$. From what has been discussed above, the elimination of false positives is no side effects.

Table 6. Example of false-positive elimination.

Before	After
212:int column = data.Column ();	212:int column = data.Column ();
213:int row = s.Count() >= 0?s.Count():data.Row();	213:int row = s.Count() >= 0?s.Count():data.Row();
214:double used=space*Axis.getLowerMargin();	214:double used = space*Axis.getLowerMargin();
215:if ((row*column) > 0){	215:if((row*column)>0){
. . .	. . .
240: System.out.println(s.content);	240: System.out.println(s.content);
241: used = 1/(row*column);	241:+ if((row*column)! = 0)
242: }	242: used = 1/(row*column);
	243:}

6.2. Validity of Patches Generated by DTSFix

We evaluate whether the patches produced by DTSFix are valid for automatic repair, and once DTS produces a list of suspect statements, the DTSFix repairs each suspect statement according to the security constraint rules. After the candidate patch is generated, it is applied to the buggy program. If it can pass all test cases successfully, the candidate patch will be considered to be a plausible patch. Furthermore, DTSFix stops looking for other candidate patches for the bug, otherwise the patch generation steps are repeated until all defect reports are processed. Specifically, we only evaluate the correctness of the first generated reasonable patch for each error. For all plausible patches, We further manually check these differences with the patches provided in Defects4J. If they have the same semantics as those provided by the developer, we called them correct patches, otherwise remaining plausible.

The results of DTSFix were compared to the seven latest APR tools, and the repair performance was evaluated with the Defects4J benchmark. The parts with a gray background color represent the results obtained by our method. Table 7 shows the results of the ratio of plausible and correct patches. Overall, we found that DTSFix generates the correct patches for the 26 bugs in the Defects4J benchmark successfully. We provide *x/y* number: x is the number of bugs fixed correctly; y is the number of bugs for a plausible patch. Accuracy (P) represents the accuracy of bug fixes. For example, 76.1% of DTSFix plausible patches are actually correct, while 63.8% and 62.9% of respectively ACS and SimFix plausible patches are correct. The data of GenProg and Nopol were extracted from the experimental results of Martinez et al. [32], and the results of the remaining tools were all obtained from their papers SimFix [1], ACS [7], CapGen [8] and SketchFix [9]. In fact, only CapGen has achieved similar performance with high probability (84%) patches, and CapGen high performance confirms their intuition, the security constraints we provided without side effects are critical to improve the correctness of patch.

Table 7. Number of bugs fixed by different APR tools.

Pro	DTSFix	GenProg	Nopol	ACS	SketchFix	CapGen	SimFix
JfreeChart	**18/26**	4/24	6/22	10/19	19/26	18/20	14/26
Log4j	**30/36**	9/29	16/46	29/37	27/38	31/35	41/58
commons-math	**24/34**	5/37	6/39	21/36	22/32	28/36	29/70
commons-lang	**36/49**	16/50	10/48	28/45	28/42	30/37	36/49
commons-rhino	**51/60**	21/65	13/43	27/43	36/60	31/49	48/64
Total	**156/205**	55/205	51/198	115/180	132/198	138/177	168/267
P(%)	**76.1**	26.8	25.8	63.8	66.7	**78.0**	62.9

7. Conclusions

Aiming at problem of low correct rate and high false-positive rate, this paper proposes an automated program repair without side effects. Mainly repair defects such as Null Pointer Dereference, Out of Bounds, Uninitialized Variable, Resource Leak, etc. In contrast to previous repair methods, our method uses safe constraint mechanism of variables, methods and functions to synthetic repair condition, and finally combines functional verification to ensure that repaired program is in safe states. It can greatly improve the rate of successful repair and reduce false-positive rate; the quality of patch was verified in 5 different projects. Results show that our tool can repair programs effectively. The rate of successful repair is 52.16%.

In future work, we hope to further improve repair rate, refine and expand the safe constraint rules, and enhance reusability of safe constraint rules. At the same time, we consider adding defect types, such as uninitialized variables related to complex data types, freeing local pointers for later use, and so on.

Author Contributions: Conceptualization, Y.D. and M.W. (Mengying Wu); methodology, Y.D. and M.W. (Mengying Wu); software, W.Y.; validation, W.Y.; formal analysis, L.Z.; investigation, L.Z. and M.W. (Meng Wu); resources, S.P.; data curation, L.Z. and H.L.; writing—original draft preparation, Y.D. and M.W. (Mengying Wu); writing—review and editing, Y.D. and M.W. (Mengying Wu); visualization, M.W. (Meng Wu); supervision, M.W. (Mengying Wu); project administration, S.P.; 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).

Acknowledgments: In this section you can acknowledge any support given which is not covered by the author contribution or funding sections. This may include administrative and technical support, or donations in kind (e.g., materials used for experiments).

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. Nguyen, T. Semfix: Program repair via semantic analysis. In Proceedings of the 35th IEEE International Conference on Software Engineering, San Francisco, CA, USA, 18–26 May 2013; pp. 772–781.
2. Do, H.; Elbaum, S.; Rothermel, G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *ESE* **2005**, *10*, 405–435.
3. Mechtaev, S.; Yi, J.; Roychoudhury, A. Angelix: Scalable multiline program patch synthesis via symbolic analysis. In Proceedings of the 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 691–701.
4. Bader, J.; Scott, A.; Pradel, M.; Chandra, S. Getafix: Learning to fix bugs automatically. In Proceedings of the ACM on Programming Languages OOPSLA, Athens, Greece, 20–25 October 2019; Volume 3, pp. 1–27. [CrossRef]
5. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. Genprog: A generic method for automatic software repair. *IEEE Trans. Softw. Eng.* **2011**, *38*, 54–72. [CrossRef]

6. 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 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 12–17 July 2015; pp. 24–36. [CrossRef]
7. Xiong, Y.T. Precise condition synthesis for program repair. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering, Buenos Aires, Argentina, 20–28 May 2017; pp. 416–426.
8. Wen, M.; Chen, J.; Wu, R.; Hao, D.; Cheung, S.C. Context-aware patch generation for better automated program repair. In Proceedings of the IEEE/ACM 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–11.
9. Hua, J.; Zhang, M.; Wang, K.; Khurshid, S. Sketchfix: A tool for automated program repair approach using lazy candidate generation. In Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Lake Buena Vista, FL, USA, 4–9 November 2018; pp. 888–891. [CrossRef]
10. Xiao, Q.T. Research on key technologies to improve the accuracy of static defect detection[D]. Beijing University of Posts and Telecommunications. 2012. (In Chinese with English Abstract)
11. Hua, J.T. EdSketch: Execution-driven sketching for Java. *Int. J. Softw. Tools Technol. Transf.* **2019**, *21*, 249–265. [CrossRef]
12. Suzuki, R.; Suzuki, R.; Suzuki, R.; Polozov, O.; Gulwani, S.; Gheyi, R.; Hartmann, B. Learning syntactic program transformations from examples. In Proceedings of the 39th Int'l Conference on Software Engineering, Buenos Aires, Argentina, 20–28 May 2017; pp. 404–415.
13. Abreu, R.; Zoeteweij, P.; Van Gemund, A.J. On the accuracy of spectrum-based fault localization. In Proceedings of the IEEE Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION, Windsor, UK, 10–14 September 2007; pp. 89–98.
14. Tufano, M.; Watson, C.; Bavota, G.; Penta, M.D.; White, M.; Poshyvanyk, D. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.* **2019**, *28*, 1–29. [CrossRef]
15. Dong, Y. Fully Detection of Illegal Memory Access Defects. In Proceedings of the International Conference on Complex, Fukuoka, Japan, 6–8 July 2016.
16. Mechtaev, S.T.; Roychoudhury, A. Directfix: Looking for simple program repairs. In Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 448–458.
17. Le, X.B.D.; Lo, D.; Le Goues, C. History driven program repair. In Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, Suita, Japan, 14–18 March 2016; Volume 1, pp. 213–224.
18. Yuan, Y.; Banzhaf, W. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Trans. Softw. Eng. Methodol.* **2020**, *29*, 1–53. [CrossRef]
19. Le Goues, C.; Forrest, S.; Weimer, W. Current challenges in automatic software repair. *Softw. Qual. J.* **2013**,*21*, 421–443. [CrossRef]
20. Le, X.B.D.; Chu, D.H.; Lo, D.; Le Goues, C.; Visser, W. S3: Syntax-and semantic-guided repair synthesis via programming by examples. In Proceedings of the 11th Joint Meeting on Foundations of Software Engineering, Uppsala, Sweden, 22–29 April 2017; pp. 593–604.
21. Long, F.; Rinard, M. Staged program repair with condition synthesis. In Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, Bergamo, Italy, 30 August–4 September 2015; pp. 166–178.
22. Long, F.; Rinard, M. An analysis of the search spaces for generate and validate patch generation systems. In Proceedings of the IEEE/ACM 38th International Conference on Software Engineering, Austin, TX, USA, 14–22 May 2016; pp. 702–713.
23. Long, F.; Rinard, M. Automatic patch generation by learning correct code. In Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, FL, USA, 20–22 January 2016; pp. 298–312.
24. Koyuncu, A.; Liu, K.; Bissyé, T.F.; Kim, D.; Klein, J.; Monperrus, M.; Le Traon, Y. Fixminer: Mining relevant fix patterns for automated program repair. *ESE* **2020**, *25*, 1–45. [CrossRef]
25. Kim J, T. The effectiveness of context-based change application on automatic program repair. *ESE* **2020**, *25*, 719–754. [CrossRef]

26. Lutellier, T.; Pham, H.V.; Pang, L.; Li, Y.; Wei, M.; Tan, L. CoCoNuT: Combining context-aware neural translation models using ensemble for program repair. In Proceedings of the 29th ACM SIGSOFT Internatinal Symposium on Software Testing and Analysis, 18–22 July 2020; pp. 101–114. Available online: https://dl.acm.org/doi/10.1145/3395363.3397369 (accessed on 22 November 2020).
27. Tan, S.H.; Roychoudhury, A. Relifix: Automated repair of software regressions. In Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering, Florence, Italy, 16–24 May 2015; Volume 1, pp. 471–482.
28. Mechtaev, S.; Gao, X.; Tan, S.H.; Roychoudhury, A. Test-equivalence analysis for automatic patch generation. *ACM Trans. Softw. Eng. Methodol.* **2018**, *27*, 1–37. [CrossRef]
29. Wang, S. Data Flow Analysis for Sequential Storage Structures. *J. Softw.* **2020**, *31*, 1276–1293. (In Chinese with English Abstract)
30. Xiao, Q.; Gong, Y.; Yang, Z.H.; Jin, D.; Wang, Y. A path-sensitive static defect detection method. *J. Abbr.* **2010**, *21*, 209–217. (In Chinese with English Abstract)
31. Dong Y.T. Automatic Repair of Semantic Defects Using Restraint Mechanisms. *Symmetry* **2020**, *12*, 1563. [CrossRef]
32. 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.* **2017**, *22*, 1936–1964. [CrossRef]