symmetry

MDPI

Article

Automatic Repair of Semantic Defects Using Restraint Mechanisms

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

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

* Correspondence: dongyk@upc.edu.cn

Received: 1 September 2020; Accepted: 18 September 2020; Published: 22 September 2020

Abstract: Recently, software, especially CPS and Internet of Things (IoT), increasingly have high requirements for quality, while program defects exist inevitably duo to the high complexity. Program defect repair faces serious challenges in that such repairs require considerable manpower, and the existing automatic repair approaches have difficulty generating correct patches efficiently. This paper proposes an automatic method for repairing semantic defects in Java programs based on restricted sets which refer to the interval domains of related variables that can trigger program semantic defects. Our work introduces a repair mechanism symmetrically combining defect patterns and repair templates. First, the program semantic defects are summarized into defect patterns according to their grammar and semantic features. A repair template for each type of defect pattern is predefined based on a restricted-set. Then, for each specific defect, a patch statement is automatically synthesized according to the repair template, and the detected defect information is reported by the static detection tool (DTSJava). Next, the patch location is determined by the def-use chain of defect-related variables. Finally, we evaluate the patches generated by our method using DTSJava. We implemented the method in the defect automatic repair prototype tool DTSFix to verify the effect of repairing the semantic defects detected by DTSJava in 6 Java open-source projects. The experimental results showed that 109 of 129 program semantic defects were repaired.

Keywords: automatic program repair; program semantic defect; defect pattern; restricted-set; patch synthesis

1. Introduction

With the development of information technology, the Internet of Things (IoT) has made breakthrough progress in smart transportation, smart home, public safety, and so forth, and extended it to satellites, airplanes, submarines, and other areas. The number and complexity of IoT security defects have increased significantly. Program defects that threaten IoT security [1] may cause operational errors under certain conditions, producing abnormal results or behaviors, or even large irreparable losses in severe cases. Generally, developers are busy implementing algorithms and functions, which makes it easy to miss hidden semantic defects. These missed defects subsequently result in substantial workloads to find and repair the defects during the testing and maintenance stages.

Newly, static analysis to identify common program defects and automatic program repair (APR) becomes the main approach to strengthen the security of IoT applications [2]. APR is gradually becoming a hot spot in software engineering research due to its advantages in helping developers find and repair defects more efficiently. APR techniques, which reduce the onerous burden of debugging and preserve program quality, are of tremendous value. Depending on the targets of the repair, APR techniques can be classified into the following two families: functional defect repair [3–5] and

semantic defect repair [6–10]. Functional defect repair focuses on repairing defects that fail to meet the functional requirement specifications, while semantic defect repair is intended to repair defects that violate program security semantics. Functional defect repair usually relies on test cases or assertions, but test suites coverage issues and patch overfitting problems severely impede repair precision and efficiency. Qi et al. [11] performed manual inspections to repair to 105 real defects in GenProg [3] and AE [12] and found that only two of the GenProg repair results, and only three of the AE repair results were semantically correct. The program semantic defect repair not only guarantees the function-required implementation but also ensures the correctness of the program semantics. Defect repair based on program semantics can narrow the search space, ensure the correctness of the program semantics after a repair, and improve the repairs' success rate. However, existing program semantic defect repair methods do not combine defect information during the defect detection process. Thus, the repair is not targeted, and the success rate of repairs is still low.

By comparing the repair effect of different repair methods, we found that repair methods for specific defect patterns are more targeted and the success rate of repairs is higher. This paper proposes an automatic program repair method using restraint mechanisms for defect patterns. Our work introduces a repair mechanism symmetrically combining defect patterns and repair templates. With the help of the static analysis, the program semantic defects in the program are automatically repaired to improve the quality of the IoT program and security assurance. Similar to our prior work [13,14], we utilize the static detection tool DTSJava to obtain semantic defect information, including defect type, defect-related variable, and so forth. DTSJava is a static analysis tool based on defect patterns that can detect potential defects in programs, such as null pointer dereferences. Our work focuses on repairing variable-related defects that refer to a problem, error, or hidden defect caused by a variable value that damages the normal operation of the computer software or the program. In severe cases, the system may exit or crash abnormally. The current variable-related defects mainly include null pointer dereferences, out-of-bounds, illegal calculations, and so on.

First, we extract the defect information, such as the defect file name, defect pattern, defect location, to guide the repair process. Aiming at different defect patterns [13], we summarize six common defect repair templates and propose a unified repair method for semantic defects. Next, a corresponding predefined repair template is selected for each specific defect based on the defect pattern. And the patch condition is automatically synthesized by the defect-related variables based on a restricted-set denoting the defect semantic constraints. Then, the patch location is determined based on the principle of minimum program modification, and the patch statement is applied to the location. Finally, the generated patch is retested by DTSJava to ensure that the patch is both correct (i.e., the defect was repaired) and safe (i.e., no new defect was induced). This approach fully utilizes the defect information reported by DTSJava to synthesize the precise condition and determine the patch location, avoiding blind automatic program repairs and improving the repair precision and efficiency.

In summary, the main contributions of this paper are as follows:

- An automatic program repair method that utilizes defect information detected by the static analysis tool DTSJava.
- Two algorithms of conditional synthesis are proposed based on a restricted-set and patch location to achieve multi-point repair.
- An analysis of the repair results for 129 defects in a real-world program after applying the method proposed in this paper is provided.

We have implemented our approach as a Java program repair system, DTSFix, and evaluated DTSFix on six open-source Java projects. Our approach achieves an optimal balance in scalability (repairing the large-scale real-world projects), repairability (repairing more types of semantic defects detected by DTSJava), and repair quality (obtaining a functionally equivalent patch). And the approach we proposed can repair the 84.5% semantic defects, and achieve the functional-equivalence repair.

2. Related Work

At present, related researches on the automatic repair of program semantic defects are mainly divided into three main categories—program semantic defect repair based on constraint solving, program semantic defect repair based on program specifications, and template-based program semantic defect repair.

The program semantic defect repair method based on constraint solving refers to acquiring program runtime information via symbol execution, generating the constraint conditions required for solving and using conditional synthesis to complete the program repair. Hoang et al. [15] proposed an automatic repair method, called SemFix, which was based on symbolic execution, constraint solving, and program synthesis. Constrained by a given test suite, the constraint is solved by iteratively repairing the hierarchical space of the expression, and finally, the repair condition is synthesized. Specific to Java programs, Xuan et al. [16] proposed an automatic repair method for defect conditional statements, called Nopol, which uses the concept of angelic value pair positioning to determine the expected value of the condition during test execution and encodes the runtime trace collection variable and its actual value as an instance of satisfiability modulo theories (SMT). Finally, the solution is transformed into a code patch. Xuan-Bach et al. [17] inferred semantic constraints via Symbolic PathFinder, a well-known symbolic execution engine for Java programs, and utilized conditional synthesis to implement semantic repair of Java programs.

Program semantic defect repair based on program specifications refers to the use of a series of expected behaviors, namely, the program specifications, to achieve automatic program repair. This process is divided into incomplete-specification repair and complete-specification repair issues. Mechtaev et al. [18] introduced the idea of using a reference program to alleviate the overfitting problem and provided a new idea to alleviate the over-fitting problem in incomplete-specification repair issues. Gao Qing et al. [19] focused on a memory leak in C code, by summing up the program specifications that must be satisfied when no memory leak is present, and proposed a memory leak repair method based on these complete-specifications.

Template-based program semantic defect repair refers to a generic template summarized by the prior repair experience and patch data. Chen Liu et al. [20] combined historical fix templates, machine learning techniques, and semantic patch generation techniques to fix defects automatically. Benoit Cornu et al. [21] designed nine repair strategies by predefining two major types of repair templates and implemented runtime fixes for null pointer references in Java projects. Dongsun Kim et al. [22] studied existing human-written templates to obtain the 8 general repair templates, analyzed the synthetic repair conditions, and achieved automatic defect repair. Kui et al. [10] used a repair template for defective codes after static analysis as part of a patch generation effort to implement the AVATAR system. This system complemented other template-based repair methods. To address the inaccuracies of the existing methods, Xiong et al. [23] achieved the precise conditional synthesis of program repair by variable sorting, API document analysis and predicate mining; and the accuracy of this system reached 78.3%, which improved the patch repair rate.

3. Program Semantic Defects

3.1. Program Semantic Defect Patterns

Definition 1. *Semantic defect. During the programming process, the code is fully compliant with the specifications of the computer language, and no compile/link errors occur, but logic errors may exist. This defective code may cause a program exception at runtime, and could even cause the system to crash.*

The semantic defects related to variables are caused by the value of variables. During the static analysis process, for the current real domain, and the restricted domain of the variable γ_v^l, if $\theta_v^l \cap \gamma_v^l \neq \varnothing$, a program semantic defect may be generated.

Definition 2. *Defect pattern. A defect pattern refers to a class of semantic defects that have a commonality, where the same grammatical or semantic feature describes a program property, the root cause, or the same solution. This paper summarizes semantic defects into defect patterns.*

Definition 3. *Restricted-set. Variables may violate the sets of intervals of the legal value range in the program. The restricted interval domain is defined as a restricted-set.*

Let D be the interval domain of the variable, including $\varnothing$. The set of connectors is denoted as C = {$\wedge$,$\vee$}. If both two operands of the restricted operation violate the range of values, the connector $\wedge$ will be used; otherwise, it is $\vee$. The restricted rule defined by the restricted-set is a quadruple $R = <e, domain_1, c, domain_2>$, where $domain_1$ represents the restricted interval domain of the first relevant operational expression, e represents the defect expression, and $domain_2$ represents the restricted interval domain of the second relevant operation expression. $domain_1 \subseteq$ D, $domain_2 \subseteq$ D, $c \in$ C. This paper specifies that the operational expression is divided into a pre-expression operator (preoperator) and post-expression operator (postoperator). For example, in the expression $c = a/b$, the operator is the arithmetic division, and the preoperator and postoperator are a and b respectively. From the restricted rule of the division operator, the divisor cannot be zero, therefore the restricted set of $Defect_{var}(d)$ in the restricted-set of the b is [0,0] in this example.

For example, the program shown in Figure 1 is the PolarPlot.java file of the Java open-source project **jfreechart**. The program initializes the variable *state* to null at line 1399, and *axes.size*() in line 1398 may return 0, the statement in the next for loop fails to execute and causing the code in line 1411 to be executed immediately. However, when the variable *state* is used in line 1417, its value is still null; therefore, a null pointer dereference occurs at that point. The defect pattern in Figure 1 is a null pointer dereference, and according to the repair template, a null check statement should be inserted. The restricted-set of the variable *state* is determined and the patch condition **state!=null** is synthesized. Then, the def-use chain analysis is used to determine the patch insertion location, which adds a null check before line 1417.

```
1398:   int axisCount = this.axes.size();
1399:    AxisState state = null;
1400:    for (int i = 0; i < axisCount; i++) {
              ......
1405:       AxisState s = this.drawAxis(axis, location, g2, dataArea);
1406:        if (i == 0) {
1407:           state = s;
1408:           }
1409:           }
            ......
1416:+     if(state!=null){
1417:      drawGridlines(g2, dataArea, this.angleTicks, state.getTicks());
1418:+            }
```

Figure 1. Example of defect program repair.

3.2. Semantic Defect Detection

In this paper, the static detection [13] tool for specific defect patterns, DTSJava, can be used to detect various defects through static analysis of program source code. The overall system architecture diagram of DTSJava is illustrated in Figure 2. The defect information includes the main details regarding the defect, such as the name of the defect file, the defect type, the defect location, and the

restricted-set of defects, and so forth. The specific defect information outputted in the defect report is illustrated by the defective program in Figure 1, as shown in Table 1.

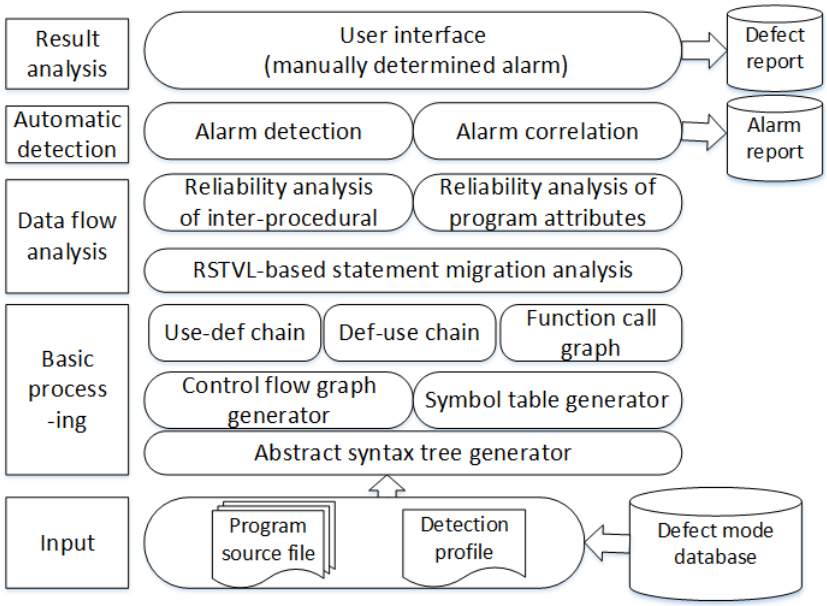

Figure 2. Basic framework of DTSJava.

Table 1. Example of defect report output.

Defect Feature	Defect Information	Example
Defect	defect type	The defect type in line 1417 is "fault"
Category	defect pattern	The defect pattern in line 1417 is "null pointer dereference"
Id	defect point ID	The unique number of the defect point in AST,52
File	location of defect file	The absolute path of the defect file, e.g., D:\testprogram \test.java
Variable	related variable	The variable related to the defect in line 1417: state
StartLine	start line	The declaration line for the state variable: line 1399
IPLine	defect line	The defect line: line 1417
IPLineCode	code in defect line	The code in the defect line: drawGridlines(g2,dataArea,this.angleTicks,state.getTicks());
Restrictset	restricted set	The interval set over which variables may violate the legal range of values in the program

The static detection tool DTSJava enables accurate analysis of Java projects, using field-sensitive and context-sensitive program analysis to obtain accurate defect information. The synthesis of patch statements depends on the information in the defect report, which helps make the repair more targeted and helps developers find the specific causes and defect information that leads to defects. DTSFix does not need to generate a large number of candidate patches, which reduces the overhead of generating and filtering invalid patches.

To facilitate the description of the program semantic defect repair method proposed in this paper, the following symbols are used to represent the various attributes of the defect record, where d refers to a specific defect:

$Defect_{pattern}(d)$:= defect pattern
$Defect_{id}(d)$:= The unique number of the corresponding node in the abstract syntax tree

$Defect_{file}(d)$:= defect file
$Defect_{op}(d)$:= operator in defect point
$Defect_{var}(d)$:= defect-related variable
$Defect_{line}(d)$:= defect line
$Defect_{rst}(d)$:= defect restricted-set

Various properties of the repair record:

$Patch_{syn}(d)$:= patch synthesis condition
$Patch_{startloc}(d)$:= patch start location
$Patch_{endloc}(d)$:= patch end location

4. Program Semantic Defect Repair

Upon receiving a defect report, we execute DTSFix, analyze the defect report, determine the defect pattern, and generate possible patches to repair the defect. First, a repair template is selected according to the defect pattern. Then, a patch statement is obtained based on the restricted-set. Next, the patch location is determined. Finally, the patch condition and the patch location are synthesized to generate the patch, achieving automatic repair of defective programs.

4.1. Repair Template

According to the results of the data flow analysis, the defect pattern state machine iterates from the start state of each FSM state instance. The update of data flow information in the FSM state instance, triggers FSM condition; at that point, the defect state transitions from *start* to an interim state, and then further to *end* or *error*.

The automatic testing module of DTSJava can detect different defect patterns. First, DTSJava builds the defect state machine instance according to the defect feature described for each pattern to be tested. Next, it conducts the state transition operation for the defect state machine instance according to the corresponding semantic operation in the CFG. If the state can transfer to the *error* state, the corresponding defect pattern will be produced.

The goal of our repair strategy is to ensure that the state of the repaired program never transfers to the *error* state [24]. What we want to do is to make conditions that can transfer to *error* state unsatisfiable by adding check statements. The patch statements usually have the following forms:

$$\begin{aligned}\varphi_{ps} \overset{\text{def}}{=} & \text{ (if}(c) \text{ return } Ret; \text{)} \vee \text{ (if}(c) \text{ } \{s_1;\} \text{)} \\ & \vee \text{ (if}(c) \text{ } \{s_1; s_2; ...; s_n;\} \text{ } (n{>}1) \text{)} \\ & \vee \text{ (if}(c) \text{ throw } e \text{)}\end{aligned}$$

where c, Ret, s, and e represent the patch condition, expected return value, program statements, and object of exception class respectively. The value of n is determined by the patch location.

When the value of the variable is not within the permissible range, a variable-related defect will be generated. Before a variable is used, the value of that variable needs to be checked to ensure it is legal. Therefore, these semantic defects can be repaired by inserting a check statement or a statement that legalizes the value of the variable. For variable-related defects, the effects after repairing the defects are as follows: the string or object is initialized; the variable is not null when it is used; when the variable is declared, its value does not exceed the maximum or minimum range specified by the data type on the computer platform; array index does not exceed the initialized upper-bound.

To satisfy the above conditions and repair the semantic defects of Java programs, we summarize the common repair templates from experience gained during manual repairs and developer-patches. As shown in Table 2, a unified repair method is given for program semantic defect features, which improve the efficiency of program repair.

Table 2. Semantic defect repair template.

Defect Pattern	Repair Template
Null pointor dereference	Add null checker
Variable not initialization	Initializate the object
Out-of-bounds	Add array boundary check
Illegal calculation Integer overflow	Add a variable to the legal scope check in the computer

According to the repair templates, this paper implements a more accurate program repair method (automatic program repair method for program semantic defects based on restricted-set). When a new defect report is submitted, the method first reads the defect information and then selects the corresponding defect repair template based on the defect pattern. Then, the patch condition is synthesized using information such as $Defect_{var}(d)$. Subsequently, following the principle of minimum program modification, we can utilize the AST structure and def-use chain analysis to find the location of the defect node. The nearest-distance rule of the statement block states that beginning with the block where $Defect_{var}(d)$ is located, the first definition of $Defect_{var}(d)$ is found when checking each block from inside to outside. Then, the location of the patch statement is determined. Finally, a patch is generated, and the patch is retested by DTSJava. A patch both repaired the original defect and does not cause a new program defect, it is considered to be a correct patch. A patch verified as correct is inserted into the original program code, and a defect repair report is output, that includes the number of defect repairs, the repair template for the defect, and any conditional statements. Figure 3 shows the defect repair framework of this paper.

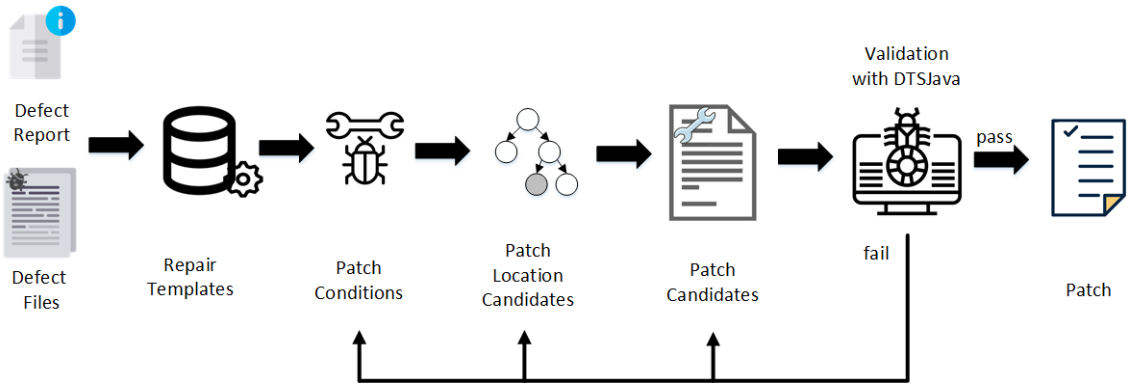

Figure 3. The Framework of DTSFix.

4.2. Patch Condition Synthesis Based on Restricted-Set

We can obtain the defect report output by DTSJava. If a defect is detected, $Defect_{pattern}(d)$ and $Defect_{op}(d)$ are first determined according to the defect report information, and $Defect_{rst}(d)$ is determined by the following constraint rule. The specific content of the restricted rules and the corresponding $Patch_{syn}(d)$ are shown in Table 3.

Table 3. Restricted rule.

Defect Pattern	Related Expression	Operator	Restricted Sets	Synthesis Condition
Null pointer dereference	e_1	e_1.f()	<e_1,null,∨,∅>	e_1!=null
Illegal calculation	e_1,e_2	/, /=, %, div(), ldiv(), fmod()	<e_2,∅,∨, [0,0]>	e_2!=0
	e_1	log(), log10(), sqrt(), _logb(),_y0(), _y1()	<e_1, [Min,0],∨,∅>	e_1>=0
	e_1	asin(), acos()	<e_1, [Min,0]∪[1,Max],∨,∅>	e_1>=0&&e$_1$<=1
	e_2	atan2()	<e_2,∅,∨,[0,0] >	e_2!=0
	e_1,e_2	pow()	<e_1∪e$_2$,[0,0],∧, [Min,0]>	e_1!=0&&e$_2$>=0
	e_1,e_2	_ jn(), _ yn()	<e_1∪e_2, [Min,0],∧, [Min,0]>	e_1>0&&e_2>=0
Out-of-bounds	e_1	array[e$_1$]	<e_1,[array.length(),Max],∨,∅>	e_1<array.length()
Variable uninitialization	e_1	Variable uninitialization before use	<e_1,[0,0]∪null,∨,∅>	e_1=new classnameofe_1()

For some common program semantic defects, the proposed method can select a predefined repair template based on the defect pattern, $Defect_{pattern}(d)$, and then use the defect information to analyze the variable-related defect to obtain the restricted-set of the defect, $Defect_{rst}(d)$. Finally, the patch synthesis condition, $Patch_{syn}(d)$, is synthesized using $Defect_{var}(d)$ as the synthesis ingredients of the patch condition according to $Defect_{rst}(d)$.

Based on the restricted-set, an algorithm is proposed to obtain the synthesized patch condition $Patch_{syn}(d)$. Another defect file performs the synthesis of the patch statement successively according to Algorithm 1 until all the defect file information has been read.

Algorithm 1: Patch synthesis condition in DTSFix

Input: A Specific Defect, *d*
Ouput: $Patch_{syn}(d)$
1 ResultSet←Defect Information from Defect Report order by File
// Read the defect information into the result set in the order of $Defect_{file}(d)$
2 $Defect_{pattern}(d)$← Category in ResultSet
3 Apply the repair-template for $Defect_{pattern}(d)$
4 Rebuild the AST of $Defect_{file}(d)$
5 **while** ($Defect_{file}(d)$.hasDefect()) // Defects still exist in a defect file
6 defectNode← ASTraverse ($Defect_{id}(d)$)
7 $Defect_{op}(d)$←operator on the currentNode
8 $Defect_{rst}(d)$←the rational variable restricted set by $Defect_{op}(d)$
9 $Patch_{syn}(d)$←synthesis condition by restricted set
10 **return** $Patch_{syn}(d)$

Algorithm 1 implements a patch condition solver based on the restricted-set. First, the process reads the defect report, sorts each row of the defect report according to $Defect_{file}(d)$, and obtains $Defect_{var}(d)$. Then, the Abstract Syntax Tree (AST) of $Defect_{file}(d)$ is reconstructed. According to the unique $Defect_{id}(d)$ of the defect point on the AST, the corresponding tree node on the AST is returned by the breadth-first traversal of the AST. Thus, $Defect_{op}(d)$ will be determined. $Defect_{rst}(d)$ is obtained

from $Defect_{op}(d)$ according to the restricted rules, and finally, $Patch_{syn}(d)$ of the patch is obtained. For instance, in the program code shown in Figure 1, line 1417 is detected as NPD. The first template selected involves adding a null check, and $Defect_{op}(NPD)$ is a reference to a variable that may be null. Then, the restricted set of $Defect_{var}(NPD)$ is obtained (here the variable *state* is determined to be null), and its composition condition is determined **state !=null**.

4.3. Patch Location Based on the Def-Use Chain

Given a program variable v and statement s, if v is used as input in s, it becomes a use point of v, denoted as $U(s, v)$. If v is assigned in s, it becomes a definition point of v, denoted as $D(s, v)$. The def-use chain is introduced to represent the def-use relationship, where U and D are the use point and definition point of the same variable v respectively, and the def-use chain U-D is used to denote that D is the last assignment of v before it is used at U. The assignment of the definition point determines the value of V at the use point U.

To increase the readability of the program, the patch insertion location is set according to the principle of minimum program modification, and the same defect patterns in the same statement block are preferentially analyzed. If the defect pattern appears in a statement block and $Defect_{var}(d)$ are the same, and there is no redefinition or a statement that affects it, the location where the statement block starts is determined as the insert location of patch statement. In other cases, the defect location is considered the insertion location.

For example, if there are multiple defects in the **for** loop of line 1400 in Figure 1, the null pointer dereference exception caused by *state* is null, and *state* is not redefined and no subsequent statement affects it, then it will still be null, and the check statement is placed before the **for** loop.

The defect files in the Java project are sequentially repaired. Algorithm 2 determines the patch location of all the defects in a file. We can obtain the patch synthesis conditions and the defect patch location, and uniformly repair the defects in the file. During the repair, the information in the defect report is read, then breadth-first traversal of AST is performed, and the information on the ASTNode is marked on the AST based on $Defect_{pattern}(d)$ and $Defect_{var}(d)$. Algorithm 2 is the algorithm that determines the specific location for all the defect repairs in a file.

When a variable triggers only one defect, we apply the signal-point repair, that is, one patch statement repairs only one defect. But when a variable causes multiple identical defect patterns, we choose the multi-point analysis method. Algorithm 2 shows the idea of identifying the patch location following the principle of minimum program modification. The implementation of Algorithm 2 is based on the def-use chains and the patch synthesis condition obtained from Algorithm 1. Algorithm 2 determines the real patch location. First, the defect information is read. According to the unique $Defect_{id}(d)$ of the defect point on the AST, all the defect nodes are pushed into the stack by traversing the AST. The patch start and the end locations are initialized to the location of the top element of the stack. The top elements of the stack are compared with $Defect_{pattern}(d)$ and $Defect_{var}(d)$ and with other remaining elements sequentially until the bottom of the stack is reached. If $Defect_{pattern}(d)$ and $Defect_{var}(d)$ are the same, the definition point of $Defect_{var}(d)$ of the currently accessed element needs to be checked by the def-use chain. If the definition point of the top element is prior or the same as the location of the current element $Defect_{var}(d)$, we update the patch start location(the start location of the block statement where the current element is located).

The end location of the patch remains unchanged, and the current element access update flag is set; otherwise, both start and end locations of the patch remain unchanged. The bottom element of the stack is accessed and a repair triplet is returned <$Patch_{syn}(d)$, $Patch_{startloc}(d)$, $Patch_{endloc}(d)$>. Then, the top element of the stack is popped.

We need to check whether an updated access token exists. If there is one, the algorithm continues to pop items; otherwise, repeats the above steps, and the updated access token is no longer compared until the stack is empty. The algorithm uses the list to record the repair triple <$Patch_{syn}(d)$, $Patch_{startloc}(d)$, $Patch_{endloc}(d)$> for all the defects in a defect file.

Algorithm 2: Repair Location in DTSFix

Input: AST of $Defect_{file}(d)$, Defect Information of $Defect_{file}(d)$
Ouput: list[<$Patch_{syn}(d)$, $Patch_{startloc}(d)$, $Patch_{endloc}(d)$>]
1 Update Defect Information of ASTNode in AST
// ASTNode represents a node on the abstract syntax tree
2 **while** (ASTNode != null)
3 defectNode ← ASTraverse ($Defect_{id}(d)$)
4 push defectNode to the stack
5 set flag of each Element in the stack to false
6 $Patch_{startloc}(d)$, $Patch_{endloc}(d)$ ← topE.defloc()
// topE represents the top element of the stack, defloc represents the
//definition point of the defect-related variable
7 **while** (!stack.isempty())
8 **while** (stack.nextElement())
9 **if** (flag == true)
10 pop topEm
11 **else if** (topEm. $Defect_{pattern}(d)$ == nextEm.Defect$_{pattern}(d)$
&&(topEm. $Defect_{var}(d)$ == nextEm.Defect$_{var}(d)$)
// nextEm represents the next element in the stack
12 analyze the ASTNode by def-use chain
13 **if** (topEm.defloc() <nextEm.loc())
// loc indicates the location of the defect
14 $Patch_{startloc}(d)$ ← nextEm.loc()
15 $Patch_{endloc}(d)$ ← topEm.defloc()
16 set the flag of the nextEm as true
17 **else** $Patch_{startloc}(d)$, $Patch_{endloc}(d)$ ← topEm.defloc()
18 get the $Defect_{syn}(d)$ of topEm
19 pop topEm
20 **return** list[<$Patch_{syn}(d)$, $Patch_{startloc}(d)$, $Patch_{endloc}(d)$>]

4.4. Verification of Functional Equivalence of Programs

Automatic program repair can modify bug code, but functional test cases may not be sufficient to ensure that the function of the program after the repair is consistent with the original. Although program semantic defect repair can repair bugs, it may lead to functional differences with the original program. Thus, it is necessary to verify whether the repaired program with the original one is functionally equivalent.

Definition 4. *Functional equivalence. If the same outputs are generated before and after repair for any inputs, then it is called functional equivalence.*

$$\frac{Exec(BRP)(input) = output, Exec(ARP)(input) = output}{BRP^f = ARP^f} \quad (1)$$

where the BRP, ARP, and f indicate the before-repair program, after-repair program and function of the original program respectively.

The program function can be regarded as a set of program paths. Therefore, if we can ensure that each path produces the same output for the same input before and after the repair, we can consider the path equivalent. Given a program S with semantic defects, in which exists program statement s and defective statement s', the control flow graph is obtained by program S, and the corresponding defective statement is marked as unsafe node γ on control flow graph. Traversing control flow graph,

the program path set *P*<*input*, *path*, *output*>is obtained, where *input* records program input and *path* records program execution path. The path containing γ is called unsafe path *nSP*, and the path without γ is called safe path *SP*. According to the output result of the same input of the defective program and the repaired program, it verifies whether the program change with semantics preservation and the program automatic repair with functional equivalence are realized.

In order to avoid triggering γ, security constraints are imposed on *nSP* by adding constraints (synthesis conditions) entering *nSP*. Semantic defects of *nSP* are repaired by code changes, such as adding statements, deleting statements, modifying statements.

5. Experimental Evaluation

To analyze the repair effect of our program repair method for semantic defects, we conducted a program repair comparison experiment using the defect repair algorithm and the program semantic defect automatic repair method based on the restricted-set proposed in this paper. We scanned 6 large Java open-source projects and verified the effect of repairs using DTSJava. Aiming at the defects detected by DTSJava, we compared with the state-of-the-art approach based on the repair template (PAR [22]). The patch generation results are shown in Table 4.

Statistics suggest that the density of program semantic defects is 3–5/kloc. Thus, improving the repair efficiency of program semantic defects has high value for software development. These experimental results demonstrate that we can repair most of variable-related semantic defects, and effectively improve the program quality.

The number of repairs required for different defect patterns is different. In this paper, the repair consequences of various types of defects in the experiment are counted to verify the repair effect of the proposed method on different types of defect patterns. The statistical results are listed in Table 5.

Because NPD generation is closely related to the value of the variable, we can obtain the $Defect_{var}(d)$ detected by the static defect detection tool DTSJava. Thus, by relying on DTSJava, DTSFix can repair all the NPD defects. For example, a null pointer dereference exception in the ChartPanel.java file under the **jfreechart** project was repaired as shown in Figure 4.

Table 4. Comparison of experimental result.

Subject	#Size(Line)	#Bugs	#Bugs Repaired by PAR	#Bugs Repaired by DTSFix
rhino	51,001	24	9	21
log4j	27,855	24	5	20
math	121,168	34	6	28
lang	54,537	22	3	19
collections	48,049	9	1	8
jfreechart	130,300	16	13	8
Total	432,910	129	33	109

Table 5. Repair effect of different defect patterns.

Defect Pattern	Number of Defects	Number of Repaired Defects
Null pointer dereference	75	75
Illegal calculation	34	32
Variable uninitialization	0	0
Out-of-bounds	3	2

```
if (option == JFileChooser.APPROVE_OPTION) {
    String filename = fileChooser.getSelectedFile().getPath();
    if (isEnforceFileExtensions()) {
+       if(filename!=null){
            if (!filename.endsWith(".png")) {
                filename = filename + ".png";
            }
+       }
    }
}
```

Figure 4. Code comparison before and after null pointer dereference repair.

We can repair over half the illegal calculation, even achieve the multi-point repair, such as the defect shown in Figure 5 (an illegal calculation exception in the FastCosineTransformer.java file under the **commons-math** project). However, in some cases, we cannot repair these defects, for example, when the returned value of the function involved in the illegal calculation.

```
for (int i = 1; i < (n >> 1); i++) {
    final double a = 0.5 * (f[i] + f[n - i]);
+   if(n!=0){
        final double b = FastMath.sin(i * FastMath.PI / n) * (f[i] - f[n - i]);
        final double c = FastMath.cos(i * FastMath.PI / n) * (f[i] - f[n - i]);
+   }
    ......
}
```

Figure 5. Code comparison before and after illegal calculation.

Most of the out-of-bounds defects can be repaired by DTSFix in a manner similar to the repair to TokenStream.java in the **rhino-mirror** project shown in Figure 6. DTSFix determines the defect location accurately based on the defect static analysis; therefore, the repair effect is more dependable.

```
if(ungetCursor!=0){
    cursor++;
+   if(0<ungetCursor&&ungetCursor<3){
        return ungetBuffer[--ungetCursor];
+   }
}
```

Figure 6. Code comparison before and after arrays out of bound repair.

The experimental data show that our tools have the following advantages for the repairing of program semantic defects:

1. We repaired 6 Java open-source programs and achieved a high repair rate especially for NPD defects.
2. Because the method proposed in this paper can be combined with the defect detection process, the repair is targeted, and does not need to generate large numbers of candidate patches; thus, it substantially reduces the repair time.
3. Compared with other approaches which assume that the repair is successful if the patch program can pass all of the test cases (but where the program may actually still contain program semantic defects), our method utilizes DTSJava to assure the semantic correctness of the repaired program, which avoids invalid repairs.

Although the method we proposed possesses the advantages listed above, the following threats to validity still exist:

1. The proposed method can repair variable-related defects, but it does not currently the repair non-variable related defects, such as multi-thread related or concurrent correlation defects.
2. The proposed method mainly repairs the defects based on prior defect detection information generated by DTSJava. When defects cannot be detected by DTSJava, the proposed method lacks the necessary reference information. In such cases, the versatility of the repair method is limited.

6. Conclusions

This paper analyzed the advantages and disadvantages of current automatic program repair methods and proposed a semantic defect repair method for the Java program. Compared with existing other semantic defect repair methods, our approach avoids the blind repairs and does not generate any invalid patches, which reduces the development costs. The repair effect achieved on the semantic defects of six large Java open-source projects showed that the proposed method is highly targeted and yields a high repair rate.

In future work, we plan to formulate additional general repair templates from manual defect repair experiences and from data mining of repair reports and add the associated repairable defect patterns. We also plan to apply DTSFix to more Java open source programs to verify the effectiveness of the proposed method. Finally, we also plan to use machine learning and other data mining algorithms to learn the relationship between defect patterns and repair strategies, train a generalizable repair model, and improve the repair efficiency.

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

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. Xu, G.; Zhao, Y.; Jiao, L.; Feng, M.; Ji, Z.; Panaousis, E.; Chen, S.; Zheng, X. TT-SVD: An Efficient Sparse Decision Making Model with Two-way Trust Recommendation in the AI Enabled IoT Systems. *IEEE Internet Things J.* **2020**. [CrossRef]
2. Xu, G.; Wang, W.; Jiao, L.; Li, X.; Liang, K.; Zheng, X.; Lian, W.; Xian, H.; Gao, H. SoProtector: Safeguard Privacy for Native SO Files in Evolving Mobile IoT Applications. *Internet Things J.* **2020**, *7*, 2539–2552. [CrossRef]
3. Goues, C.L.; Nguyen, T.; Forrest, S.; Weimer, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [CrossRef]
4. Goues, C.L.; 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; pp. 3–13.
5. Le, X.D.; Le, Q.L.; Lo, D.; Goues, C.L. Enhancing Automated Program Repair with Deductive Verification. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Raleigh, NC, USA, 2–7 October 2017; pp. 429–432.
6. Li, B.; He, Y.; Ma, H. Automatic program repair: Key problems and technologies. *Ruan Jian Xue Bao J. Softw.* **2019**, *30*, 244–265.
7. Xiong, Y.; Liu, X.; Zeng, M.; Zhang, L.; Huang, G. Identifying patch correctness in test-based program repair. In Proceedings of the 40th International Conference on Software Engineering, Gothenburg, Sweden, 27 May–3 June 2018; pp. 789–799.

8. Durieux, T.; Monperrus, M. DynaMoth: Dynamic code synthesis for automatic program repair. In Proceedings of the International Workshop on Automation of Software Test, Austin, TX, USA, 14–15 May 2016; pp. 85–91.
9. Le, X.D.; Chu, D.; Lo, D.; Goues, C.L.; Visser, W. S3: Syntax- and semantic-guided repair synthesis via programming by examples. In Proceedings of the Joint Meeting on Foundations of Software Engineering, Uppsala, Sweden, 22–29 April 2017; pp. 593–604.
10. Liu, K.; Koyuncu, N.; Kim, O.; Bissyande, E.F. Avatar: Fixing semantic bugs with fix patterns of static analysis violations. In Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 1–12.
11. 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 & Analysis, Baltimore, MD, USA, 12–17 July 2015; pp. 702–713.
12. Weimer, W.; Fry, Z.P.; Forrest, S. Leveraging program equivalence for adaptive program repair: Models and first results. In Proceedings of the Automated Software Engineering, Silicon Valley, CA, USA, 11–15 November 2013; pp. 356–366.
13. Dong, Y. Illegal computing defect detection by static analysis for C program. *Comput. Eng. Appl.* **2016**, *52*, 31–36.
14. Dong, Y.; Gong, Y.; Jin, D. Null Pointer Dereference Defect Detected Based on Region-Based Memory Model. *Acta Electron. Sin.* **2014**, *42*, 1744–1752.
15. Nguyen, H.D.T.; Qi, D.; Roychoudhury, A.; Chandra, S. SemFix: Program Repair via Semantic Analysis. In Proceedings of the 2013 35th International Conference on Software Engineering(ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 772–781.
16. Xuan, J.; Martinez, M.; DeMarco, F.; Clement, M.; Marcote, S.L.; Durieux, T.; Berre, D.L.; Monperrus, M. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Trans. Softw. Eng.* **2017**, *43*, 34–52. [CrossRef]
17. Le, X.D.; Chu, D.; Lo, D.; Goues, C.L.; Visser, W. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In Proceedings of the International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, 10–14 July 2017; pp. 376–379.
18. Mechtaev, S.; Roychoudhury, A. Semantic program repair using a reference implementation. In Proceedings of the 40th International Conference, Gothenburg, Sweden, 27 May–3 June 2018; pp. 129–139.
19. Gao, Q.; Xiong, Y.; Mi, Y.; Zhang, L.; Yang, W.; Zhou, Z.; Xie, B.; Mei, H. Safe Memory-Leak Fixing for C Programs. In Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), Florence, Italy, 16–24 May 2015; pp. 459–470.
20. Liu, C.; Yang, J.; Tan, L.; Hafiz, M. R2Fix: Automatically Generating Bug Fixes from Bug Reports. In Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation(ICST), Luxembourg, 18–22 March 2013; pp. 282–291.
21. Durieux, T.; Cornu, B.; Seinturier, L.; Monperrus, M. Dynamic patch generation for null pointer exceptions using metaprogramming. In Proceedings of the 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER), Klagenfurt, Austria, 20–24 February 2017; pp. 349–358.
22. Kim, D.; Nam, J.; Song, J.; Kim, S. Automatic patch generation learned from human-written patches. In Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE), San Francisco, CA, USA, 18–26 May 2013; pp. 802–811.
23. Xiong, Y.; Wang, J.; Yan, R.; Zhang, J.; Han, S.; Huang, G.; Zhang, L. Precise condition synthesis for program repair. In Proceedings of the IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 416–426.
24. Zhang, D.; Jin, D.; Gong, Y.; Wang, Q.; Dong, Y.; Zhang, H. Optimizing static analysis based on defect correlations. *Ruan Jian Xue Bao J. Softw.* **2014**, *25*, 386–399.