

Article

Automatic Repair Method for Null Pointer Dereferences Guided by Program Dependency Graph

Yukun Dong ^{*}, Yuxue Sun and Xun Wang 

College of Computer Science and Technology, China University of Petroleum, Qingdao 266580, China; z20070064@s.upc.edu.cn (Y.S.); wangsyun@upc.edu.cn (X.W.)

* Correspondence: dongyk@upc.edu.cn; Tel.: +(86)-176-8551-1137

Abstract: Automatic program repair (APR) is an effective technique for eliminating defects. The repair of null pointer dereferences, as the most common defects, requires accurate dependencies among statements to determine where to repair and how to repair. In order to precisely identify the data and control dependencies, the program dependency graph is adopted. Based on the symmetry among a large number of patches, we propose four repair mechanisms in this passage, namely the assignment mechanism, restraint mechanism, evading mechanism, and transfer mechanism, and employ the decision tree algorithm to match the best repair mechanism for defects. The four repair mechanisms locate the accurate repair position using the program dependency graph, and generate candidate patches by reassigning the null pointer with an appropriate value, making a judgment for the null value in advance, or throwing an exception. Our method was implemented in the repair tool DTSFix, which supports the automatic repair of null pointer dereference in Java programs. The experimental result on Defects4J shows that 73% of null pointer dereferences are successfully repaired by DTSFix, and that the generated candidate patches do not contain over-fitting patches.

Keywords: automatic program repair; null pointer dereference; program dependency graph; program patch; decision tree



Citation: Dong, Y.; Sun, Y.; Wang, X. Automatic Repair Method for Null Pointer Dereferences Guided by Program Dependency Graph. *Symmetry* **2022**, *14*, 1555. <https://doi.org/10.3390/sym14081555>

Academic Editors: Jeng-Shyang Pan and Sergei D. Odintsov

Received: 13 June 2022

Accepted: 25 July 2022

Published: 28 July 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (<https://creativecommons.org/licenses/by/4.0/>).

1. Introduction

The repair of defects is necessary during software development and maintenance, but it will consume numerous resources and result in a decrease in the developers' energy and effectiveness. Therefore, APR has become a research hotspot [1]. At present, the mainstream techniques of program repair are divided into three stages according to the repair process: defect localization, which finds the best location of the repair; patch generation, which generates candidate patches using static or dynamic analysis techniques; patch verification, which verifies whether the candidate patch is correct. Since different defects have different causes and code impact scopes, using generic repair strategies [2,3] often results in inaccurate repair locations and inappropriate patch generation strategies. Therefore, specific repair strategies aiming at a certain type of defect are required [4,5].

Null pointer dereference is one of the main types of program defects, and often leads to abnormal program termination or system crashes. The detection methods of null pointer dereferences [6] are relatively mature, but automatic repair methods and tools [7] are not ideal: some repair tools [8,9] only repair one or two of the 15 null pointer dereferences in Defects4J [10]. An analysis of the available studies suggests two main reasons for this situation. First of all, nearly 80% of variables in the object-oriented programs are the object types because objects can be assigned to each other, passed as parameters, etc. Secondly, data and control dependencies need to be accurately analyzed for selecting the repair location and generating the corresponding patches. VFix [11] was proposed to apply the value flow graph to analyze the program, which mainly focused on the transmission dependency of values in the analysis process. This method is also proposed to calculate

the repair location by the frequency of each statement node in the blocking paths, but it is often suitable for the repair of fewer null pointer dereferences. In view of this situation, we also put forward new ideas to improve the repair.

The repair of null pointer dereferences requires identifying the location where this null pointer is defined and tracing the source of the null pointer. In addition, it is important to understand the complex dependencies among statements and to find the program statements that are affected by a null pointer dereference. The program dependency graph [12] can accurately describe the dependencies within a program, including data dependencies and control dependencies, which are generated based on the analysis of variable dependencies and the post-dominance relation among statements in the control flow graph.

When summarizing the repair strategy for null pointer dereference, we mainly focus on the statement symmetry and program dependency graph symmetry among a large number of patches. Statement symmetry refers to the similarity of repair statement types in the patch, and the program dependency graph symmetry refers to the similarity of the affected program code range in the program dependency graph after the patch is repaired. According to the symmetry, we propose four repair mechanisms: assignment mechanism, restraint mechanism, evading mechanism, and transfer mechanism. The assignment mechanism assigns an appropriate value to the faulty pointer at the source of the defect whenever possible. In order to prevent null pointer dereferences in advance, the restraint mechanism sets up a simple check to bring an affected scope by the defect into the safe scope. In contrast with the restraint mechanism, the evading mechanism is more restrictive and changes the execution of the program to avoid null pointer dereferences. The transfer mechanism differs in that it throws null pointers to the external calling function that catches and handles the defect. In the selection of the defect repair mechanism, this paper used the trained decision tree classifier model to predict.

With the help of the program dependency graph, our repair mechanisms obtained the statement defining the defect variable and the subsequent statements affected by the defect variable. The repair process is as follows: firstly, the defect variable, the defect start line, and the defect end line are acquired by a defect detection tool; secondly, the control dependency graph and data dependency graph are obtained by employing static analysis, and these two are combined to generate the program dependency graph to receive the complex dependencies in a program; finally, different repair mechanisms are implemented to generate candidate patches for inter-procedural and intra-procedural null pointer dereferences.

The main contributions of this paper include: proposing four repair strategies, which are the assignment mechanism, restraint mechanism, evading mechanism, and transfer mechanism; presenting a method that applies a program dependency graph to find the befitting repair location; implementing a repair tool, DTSFix, to realize the repair of intra-procedural and inter-procedural null pointer dereferences; and demonstrating the effectiveness of DTSFix through experiments on Defects4J.

This paper is organized as follows: Section 1 introduces the progress of automatic program repair and related work on the repair for null pointer dereferences; Section 2 describes the framework of the repair for null pointer dereferences based on the program dependency graph; Section 3 analyzes how to build a program dependency graph or extended program dependency graph; then, the repair patterns of four repair mechanisms are presented in Sections 4, and Sections 5 presents the key processes of the automatic repair method; Section 6 carries out a series of experiments to evaluate the effectiveness and performance of DTSFix and discusses the limitations of DTSFix; Section 7 summarizes the work of this paper and looks forward to the next research work.

2. Related Work

2.1. Automatic Program Repair

According to the types of defects automatically repaired, the current automatic repair tools are divided into two main categories: general repair for multiple types and specific repair for a single type. The techniques adopted by APR tools can be classified into four repair techniques roughly: search-based, template-based, constraint-based, and statistical-analysis-based APRs. Search-based APR [13,14] uses manually defined heuristic rules to repair multiple types of defects, which expands the search space to find candidate patches using genetic algorithms, historical repair patches, and many other methods. Finally, it determines the correct patch by patch verification. In 2018, Yang et al. [15] proposed a genetic programming repair method based on the repair information of similar defects, which greatly reduced the cost of developers. Similarly, the emerging statistical-analysis-based APR [16,17] can also deal with different defects by applying deep learning methods to extract the features of codes, and trains patches of massive open-source programs to guide the repair. Huq et al. [18] trained a sequence-to-sequence model on a large number of code reviews and code changes in 2022, and introduced new processing methods. The results showed that the repair accuracy is greatly improved compared with other similar methods, and some suggestions are provided for stylistics and non-code errors.

In contrast, template-based APR [19] is more targeted, with the guidance of manually defined written templates, and specific repairs for a single type of defect can produce patches with higher quality. Sketchfix [20], a template-based automatic repair technology, was proposed by Hua et al. in 2018, and uses the auxiliary function to replace the codes to be repaired. The internal concrete implementation within the method is generated according to the predefined patch template. In the repair process, only the codes in the auxiliary function are modified each time, so the limited codes need to be updated and compiled each time, which saves the time spent compiling the whole program each time. Constraint-based APR [21] adopts symbolic execution and constraint solving algorithms to obtain the constraints of patches and transforms patch generation into constraint solving, which is suitable for general repair and specific repair. In 2021, Gao et al. [22] presented a repair method to extract symbolic constraints that automatically symbolized program variables related to security vulnerabilities to extract constraints in the process of sanitizers tracking violations. The results of implementing the method in ExtractFix show that it is very helpful in correctly repairing security vulnerabilities. However, it is not easy to apply a unified repair strategy to various defects due to the different sets of statements affected by the defect variables and the causes of defects.

2.2. The Repair of Null Pointer Dereferences

Null pointer dereferences in programs are more common and hidden than other defects, so manual repair is difficult to complete. In recent years, many general repair tools support automatic repair for null pointer dereferences. In 2013, Nguyen et al. [23] proposed SemFix to extract the correct constraints on patches semantically for the first time, and to generate correct patches under these constraints with the help of a code synthesis method. In 2018, Wen et al. [24] proposed an automatic repair tool, CapGen, to guide the repair by extracting and comparing the context from both historical patches and defect codes. To improve the repair efficiency, Ghanbari et al. [25] implemented PraPR in 2019, a JVM byte code-based repair technique that applies mutation operators to generate patches at the source code level of buggy programs. In 2020, Anil et al. [26] presented FixMiner, a method for mining repair patterns that uses a rich edit script to capture context at the changed AST level and mines more accurate repair patterns. Moreover, the proposed PARFixMiner, an automatic repair tool, has higher precision.

However, the effectiveness of automatic repair for null pointer dereferences is still limited, so Durieux et al. [7] proposed NPEfix specifically for the repair of null pointer dereferences in 2017. NPEfix injects all repairing policies while transcoding the repair program. The tool controls which repair behavior is enabled by executing failed test

cases dynamically, and finally obtains candidate patches that pass all test cases. In 2019, Xu et al. [11] achieved the repair tool VFix to achieve defect repair; VFix applies the chokepoints in the path of failed test cases on the value flow graph of a program to find repair locations.

3. The Repair Framework of DTSFix

To automatic repair null pointer dereferences, we developed a repair tool named DTSFix, whose repair framework is divided into three stages as shown in Figure 1: building the program dependency graph, matching repair mechanisms, and patch generation and verification. The repair takes a defective program and test cases as inputs and takes candidate patches as outputs.

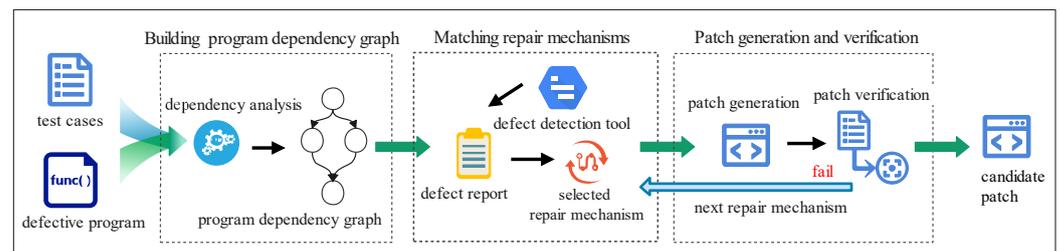


Figure 1. The repair framework of DTSFix.

Firstly, the application of the program dependency graph integrates the data dependency graph and the control dependency graph to obtain the overall dependency relationship. A data dependency graph is constructed by analyzing the define-use chains, which can obtain data dependencies. At the same time, by analyzing the post domination relationship of the statement nodes in the control flow graph, the control dependency graph can be determined in order to obtain the control dependency. The set of statements affected by a faulty variable v is called $ds(v)$, which is obtained from the program dependency graph. In addition, the defined location n_{def} of the faulty variable can be obtained from the data and control dependencies in the program dependency graph.

Then, according to the detection report and relevant information of the fault object obtained by the defect detection tool, four repair mechanisms were selected with the idea of changing the program execution path in the minimum range.

Later, the corresponding operation was applied to complete the patch. Finally, regression tests or test cases can be used for verification. If the verification is successful, the patch is considered as a candidate patch. Otherwise, the defect will be re-matched with other repair mechanisms for repair.

4. Program Dependency Graph

Notably, the precise analysis of the context at the defective point has much to do with correct repair, so it is necessary to analyze the dependencies precisely among objects and to identify the set of statements affected, which can contribute to choosing the repair location. The reason for applying the program dependency graph in the present paper is that it can identify the data and control dependencies at the same time. By further analyzing the process, data dependency can be captured by the define-use relation among variables in statements as defined in Definition 3. The post-dominance relation in Definition 2 describes which statement governs the execution of which statement. After that, in Definition 4, analyzing the relations between the statements in the control flow graph can identify the statements' control dependency. In the end, the program dependency graph in Definition 5 integrates the dependencies of two different types, the data dependency and the control dependency, which can accurately identify the context and the scope of the code affected by a defect. The relevant definitions are as follows:

Definition 1. *Post-dominance relationship.* Given a control flow graph $G = \langle N, E, \text{Entry}, \text{Exit} \rangle$, n_1 and n_2 are statement nodes; if n_2 appears on every path from n_1 to Exit , we can confirm that n_2 post-dominates n_1 .

Definition 2. *Data dependency. (DD)* Given a control flow graph $G = \langle N, E, \text{Entry}, \text{Exit} \rangle$, n_1 and n_2 are statement nodes, and there is a reachable path p from n_1 to n_2 . The variable v defined in n_1 is used in n_2 , and the path does not contain a redefinition of v ; we can determine that n_2 data depend on n_1 , denoted as $n_1 \xrightarrow{DD} n_2$.

Definition 3. *Control dependency. (CD)* Given a control flow graph $G = \langle N, E, \text{Entry}, \text{Exit} \rangle$, n_1 and n_2 are statement nodes, and there is a reachable path p from n_1 to n_2 . If all statement nodes except n_1 and n_2 in p are post-dominated by n_2 , and n_1 is not post-dominated by n_2 , we can determine that the n_2 control depends on n_1 , denoted as $n_1 \xrightarrow{CD} n_2$.

Definition 4. *Program dependence graph. (PDG)* A program dependence graph can be described as a graph $G = \langle N, E \rangle$, where $E = \{ \langle n_i, n_j \rangle \mid n_i \in N, n_j \in N, n_i \xrightarrow{CD} n_j \text{ or } n_i \xrightarrow{DD} n_j \}$ is the set of edges, and N is statement nodes.

The directed edges in a program dependency graph contain control dependency edges and data dependency edges. When n_i points to n_j , it means that n_j is data-dependent or control-dependent on n_i . Let $d_dep(n_j)$ represent the set of statements that is data-dependent on n_i , which contains all nodes pointed to by n_i in data dependency edges. In a similar way, the control-dependent set of statements $c_dep(n_j)$ can be computed. When a faulty statement node n_{err} appears, the error messages generated are likely to contaminate $d_dep(n_{err})$ and $c_dep(n_{err})$, so the set of statements $ds_ (n_{err})$ caused by the faulty statement can be obtained.

Figure 2 shows an example of the generation of the program dependency graph for a simple program. Observe that the new objects defined in statements L2 and L4 are used in other statements in Figure 2a, and the **if** condition in L5 controls the execution of L6 and L8. As depicted in Figure 2b, data dependencies between statements are determined by the define-use chains between variables. It can be seen that the set of data dependencies of L8 is {L2, L4}, so $\langle L2, L8 \rangle$ and $\langle L4, L8 \rangle$ are included in the data dependency graph. A data dependency graph (Figure 2c) is calculated and constructed through the statements' dependency analysis. The control dependency graph is constructed by identifying the post-dominance relation among statement nodes in the control flow graph (Figure 2d), as shown in Figure 2e. Finally, the program dependency graph (Figure 2f) is built by adding a data dependency edge to control the dependency graph.

Suppose that there is a faulty variable a , and L2 is the position where a is defined. Significantly, the directed edges marked (1) and (2) point to L5 and L8, which are affected by L2, according to the data and control dependencies depicted in Figure 2f. Similarly, the directed edges marked (3) and (4) point to L6 and L8, respectively, which are influenced by the error statement node L5. Therefore, $ds(a)$ is {L2, L5, L6, L8}. When the repair of interprocedural defects involves more than one method, it requires a combination of multiple methods. In this case, it is necessary to further extend multiple program dependency graphs to an extended program dependency graph that relies on the call relations in the program. The extended rules are as follows:

- (i) The extended program dependency graph adds an entry node and an exit node to a called method, which are used to manipulate the input and output of parameters.
- (ii) The data of a formal parameter are dependent on the actual parameter, and the extended program dependency graph adds a data dependency edge between them.
- (iii) The received variable of the calling method depends on the return of the called method, and the extended program dependency graph adds a data dependency edge between the calling node of the calling method and the return node of the called method.

- (iv) If a called method is in a control relationship of a calling method, the extended program dependency graph adds a control dependency edge between the control node or the controlled node of the calling method and the entry node of the called method.

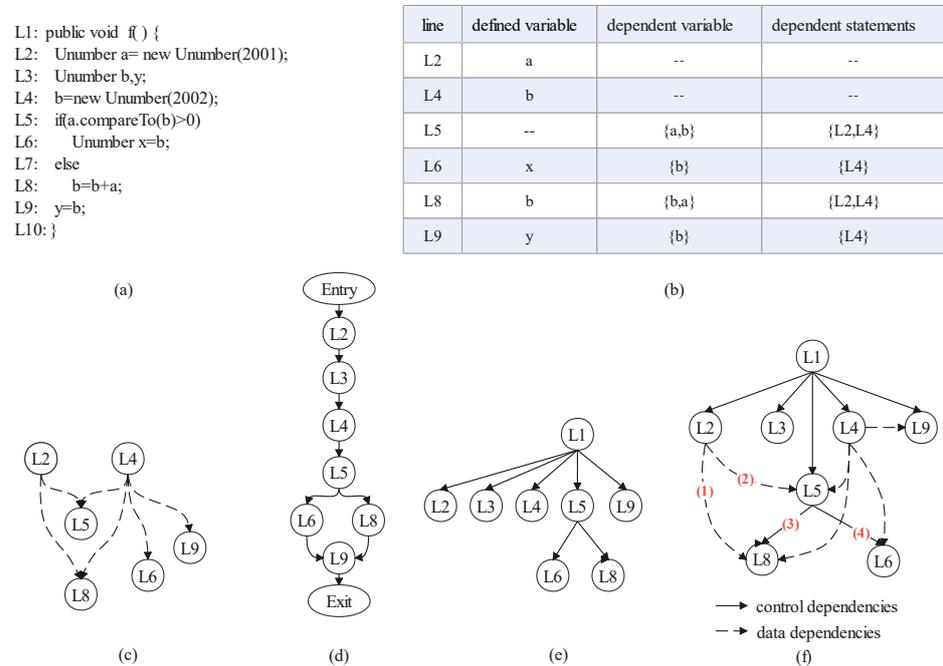


Figure 2. An example of program dependency graph. (a) code sample; (b) data dependency analysis among variables; (c) data dependency graph; (d) control flow graph; (e) control dependency graph; (f) program dependency graph.

5. Four Repair Mechanisms for Null Pointer Dereferences

According to the research analysis, the sources of null pointers can be divided into three types. The first is from internal declarations using internal instantiation; the second is from internal declarations calling other methods for assignment; the third is from external declarations using the parameter passing for assignment. To repair null pointer dereferences, an assignment mechanism, restraint mechanism, evading mechanism, and transfer mechanism are proposed in this paper based on the context of defects identified by the program dependency graph. This section illustrates the four repair mechanisms with repair examples.

5.1. Assignment Mechanism

Note that the source of null pointer dereferences caused by the internal instantiation is still located in the current method, so the primary goal of the assignment mechanism is to correctly initialize or assign a proper value to the target object at the source of this defect. This can be divided into two types: the first type is to find the location where the defective variable is defined and to reassign the variable; the second type is to perform a null check before the defect variable is used and to reassign the variable if it is null.

The equation used is a derivation rule, in which, the part on the horizontal line is the applicable condition and the symbol interpretation used, and the part below the horizontal line is the repair operation after the condition is matched. Equations (1) and (2) represent these two types of repair templates for the assignment mechanism, respectively, where l_{def} and l_{use} represent the defined and used statement of a variable v corresponding to n_{def} and

n_{use} in the nodes of the program dependency graph. In addition, l represents the statement to be added, and c indicates the judgment condition.

$$\frac{DU(l_{def}, l_{use}, v) \quad c : v == null \quad l : v = new_value;}{l_{use} \Rightarrow \text{if}(c)\{l\} \quad l_{use}} \quad (1)$$

$$\frac{DU(l_{def}, l_{use}, v) \quad l_{def} : v = _ ; \quad l : v = new_value;}{l_{def} \Rightarrow l} \quad (2)$$

When a null pointer dereference occurs using a method or property of an object, Equation (1) adds a judgment condition c before using v , and the value of v is reassigned through new_value if the judgment is true. Equation (2) replaces $v = _$ with $v = new_value$ to assign an appropriate value to v at l_{def} , where $_$ denotes a null or error value and new_value is the value reassigned to v .

Next, the four types of new_value are explained in detail. The first is the live variable at the repair location with the same type of defective variable; the second, if the defect type is a third-party API library, new_value is the instance object of the best-matched constructor between the live variables at the repair location and the formal parameters; regarding the third, if it is the custom-defined type without subclasses and there is only one constructor, then new_value is the instance object of the current constructor, whereas, if there are multiple constructors, then new_value is the instance object of the best-matched constructor that can be selected by matching the live variables in the repair location with the parameters in the constructors; regarding the fourth, if there is the custom-defined type with subclasses, new_value is the instance object of the subclass that best matches the name of the subclass and the name of the defective variable.

Shown in Figure 3a is an example with a null pointer dereference, where L6 is the defect location and the fault variable is $list$. Through the dependency analysis of the sample code, the program dependency graph shown in Figure 4 can be obtained. From the control-dependent edge marked (1) in Figure 4, it can be found that $list$ control depends on L5, where L5 is a loop structure. Therefore, when the assignment mechanism is applied, the assignment operation is set for $list$ before L5. By using the constructor to give a new value to $list$, the repaired program is shown in Figure 3b.

<pre> L1 public List<String> getData(BufferedReader br) throws Exception{ L2 List list; L3 if(br != null){ L4 String readLine = br.readLine(); L5 while(readLine != null) { L6 list.add(readLine); L7 readLine= br.readLine(); L8 } L9 } L10 return list; L11 }</pre>	<pre> L1 public List<String> getData(BufferedReader br) throws Exception{ L2 List list; L3 if(br != null){ L4 String readLine = br.readLine(); + list = new ArrayList<String>(); L5 while(readLine != null) { L6 list.add(readLine); L7 readLine= br.readLine(); L8 } L9 } L10 return list; L11 }</pre>
(a)	(b)

Figure 3. The sample program 1. (a) defective program 1; (b) repaired program 1.

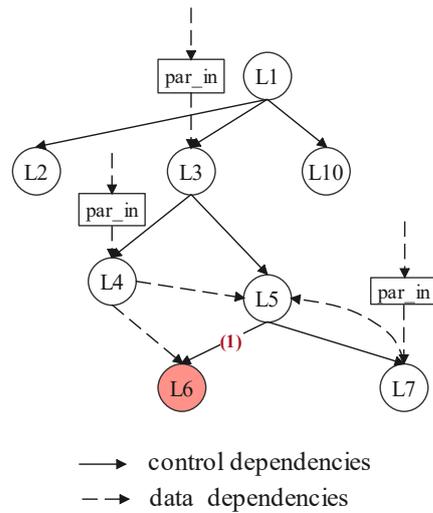


Figure 4. The extended program dependency graph of the sample program 1.

5.2. Restraint Mechanism

Null pointer dereferences are caused by external resources or services connection failure or a null return from an external method, etc., which may not be properly repaired using an assignment mechanism. In this case, the restraint mechanism can be adopted to avoid defects in advance, which can be repaired in two ways. The first way is to employ conditional judgment to constrain the set of statements influenced by the defect variable in advance, which can be executed only when the defect variables are not empty. The second way is to put the set of statements affected into the **try** block and to throw an exception whenever it occurs. Equations (3) and (4) represent these two cases, respectively, where $ds(v)$ is the set of all statements impacted by the faulty variable.

Equation (3) adds a non-full judgment for v before $ds(v)$, and the statements in $ds(v)$ can be executed only if v passes the judgment. Because l_{use} is the used statement of v , it must be included in $ds(v)$. Therefore, we replace l_{use} with $ds(v)$ included in the non-null judgment to avoid the trigger of null pointer dereferences. Equation (4) puts $ds(v)$ into the code block of **try**; the execution will be aborted once an exception occurs in the code block of **try**. After that, the method enters the code block of **catch** to warn the developer.

$$\frac{DU(l_{def}, l_{use}, v) \ c : v \neq \text{null} \ ds(v) = \{l_1, l_2, \dots, l_n\}}{l_{use} \Rightarrow \text{if}(c) \{ds(v)\}} \quad (3)$$

$$\frac{DU(l_{def}, l_{use}, v) \ ds(v) = \{l_1, l_2 \dots l_n\}}{l : \text{try}\{ds(v)\} \ \text{catch} \ (\text{NullPointerException } e) \ \{...\}} \quad (4)$$

The program fragment is shown in Figure 5a, and there is a null pointer dereference in the use of *parser* in L5. At the time, it is difficult to obtain a correct patch using the assignment mechanism, so the trigger of *parser* is avoided as much as possible. Given the program dependency analysis in Figure 6, {L6, L7, L9} depends on the faulty node L5 according to the directed edges (1), (2), and (3). The directed edges (4) and (5) indicate that both L7 and L9 are controlled by L6, so $ds(\text{parser})$ is {L5, L6, L7, L9}. The restraint mechanism is adopted to perform the null check before $ds(\text{parser})$ and $ds(\text{parser})$ is skipped if *parser* is null.

```

L1   protected void analyzeAction () {
L2       Point sta = pointState.getPoint();
L3       if (sta != null) {
L4           Parser parser = sta.getParser();
L5           Action action = parser.findAction(); +
L6           if (action != null)
L7               action.add();
L8           else
L9               action.delete();
L10      }
L11  }
    
```

(a)

```

L1   protected void analyzeAction () {
L2       Point sta = pointState.getPoint();
L3       if (sta != null) {
L4           Parser parser =sta.getParser();
L5           if (parser != null) {
L6               Action action = parser.findAction();
L7               if (action != null)
L8                   action.add();
L9               else
L10                  action.delete();
L11          }
L12      }
    
```

(b)

Figure 5. The sample program 2. (a) defective program 2; (b) repaired program 2.

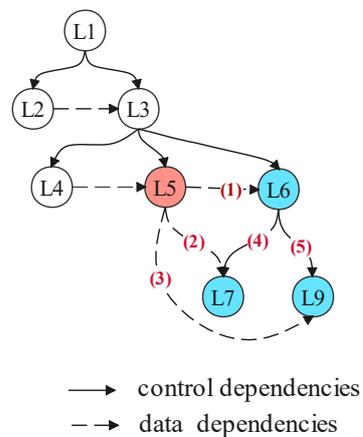


Figure 6. Program dependency graph of the sample program 2.

5.3. Evading Mechanism

The evading mechanism adds the appropriate statements to divert the execution of the method to complete the repair, and the repair can be roughly divided into three situations. First, a **return** statement appears in the set of statements impacted by the defect, and a new **return** statement should be added to avoid the defect of no return value when using conditional judgment repair. Second, when a **return** statement appears in the set of statements influenced and is put into the code block of **try** for repair, a new **return** statement needs to be added to the code block of **catch**. Third, if a null pointer occurs during one of the loops, add **continue** to skip this execution and go to the next loop. Equations (5)–(7) represent these three cases, respectively, where $\text{type}(ds(v)) \in \text{return} = \text{true}$ represents the statement set containing the **return** statement, $d_dep(v)$ represents the set of statements that the variable v data depend on, and $\text{type}(d_dep(v)) \in \text{loop} = \text{true}$ denotes the statement in $d_dep(v)$ that contains the loop statement.

The condition in Equation (5) is satisfied if $\text{type}(ds(v)) \in \text{return}$ is true, and l in the scope of the non-empty condition is executed. In the same case, Equation (6) replaces l_{use} with l , where l puts $ds(v)$ into the code block of **try** and adds $\text{return } value_{return}$ into the **catch** to avoid introducing a new defect. There are three cases of the value of $value_{return}$: firstly, if there is a live variable of the same type as the return value, then the value of $value_{return}$ is equal to the live variable; secondly, if the return value of a method belongs to primitive types, the default value of the primitive type is assigned to $value_{return}$; thirdly, if the return value of a method belongs to reference types, $value_{return}$ is obtained by new_value . If a null pointer dereference appears in the loop and $\text{type}(ds(v)) \in \text{loop}$ is satisfied, Equation (7)

sets a null judgment for the defective object and adds **continue** into the scope of the null judgment to skip this loop.

$$\frac{DU(l_{\text{def}}, l_{\text{use}}, v) \ c : v == \text{null} \ ds(v) = \{l_1, l_2, \dots, l_n\} \ l : \text{return } \text{value}_{\text{return}};}{l_{\text{use}} \xrightarrow{\text{type}(ds(v)) \in \text{return}} \text{if}(c) \{l\} \ l_{\text{use}}} \quad (5)$$

$$\frac{DU(l_{\text{def}}, l_{\text{use}}, v) \ ds(v) = \{l_1, l_2 \dots l_n\} \ l : \text{try}\{ds(v)\} \ \text{catch} \ (\text{NullPointerException } e) \{ \text{return } \text{value}_{\text{return}}; \}}{l_{\text{use}} \xrightarrow{\text{type}(ds(v)) \in \text{return}} l} \quad (6)$$

$$\frac{DU(l_{\text{def}}, l_{\text{use}}, v) \ c : v == \text{null}}{l_{\text{use}} \xrightarrow{\text{type}(d_{\text{dep}}(v)) \in \text{loop}} \text{if} \ (c) \{ \text{continue}; \} \ l_{\text{use}}} \quad (7)$$

As shown in Figure 7a, the faulty variable is *people[i]*, and the defective location is L3. The value of *people[i]* depends on the condition of the **for** and is null in one iteration. Given the program dependency graph in Figure 8, the L3 data and control depend on node 2.2, and the statement that 2.2 is located belongs to the loop. The above situation satisfies $\text{type}(ds(v)) \in \text{loop}$ in Equation (7). Therefore, the wrong variable is judged by using the null check. In addition, **continue** is executed to end this iteration and enter the next iteration if the variable is null. The defect is repaired within the minimum scope in Figure 7b.

<pre> L1 public void staff (People[] people) { L2 for (int i=0; i<3; i++) { L3 people[i].addwork(); L4 } L5 }</pre>	<pre> L1 public void staff (People[] people) { L2 for (int i=0; i<3; i++) { L3 if (people[i] == null) L4 continue; L3 people[i].addwork(); L4 } L5 }</pre>
(a)	(b)

Figure 7. The sample program 3. (a) defective program 3; (b) repaired program 3.

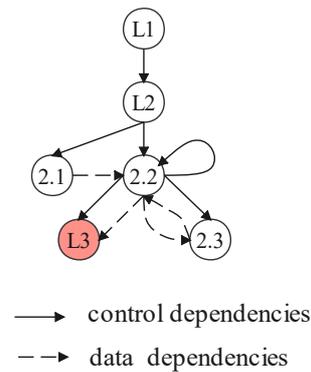


Figure 8. Program dependency graph of the sample program 3.

5.4. Transfer Mechanism

In the worst case, if all of the previous three repair mechanisms fail, the transfer mechanism is adopted to throw the exception to an external calling method that catches and handles the exception. Equation (8) sets a null judgment before using an object; then, *l* is executed if the result of the judgment is true. Further, the external calling method captures the exception to processing.

$$\frac{DU(l_{\text{def}}, l_{\text{use}}, v) \ c : v == \text{null} \ l : \text{throw new NullPointerException}(\dots);}{l_{\text{use}} \Rightarrow \text{if}(c) \{l\} \ l_{\text{use}}} \quad (8)$$

Suppose that the passed argument *axis* is null in Figure 9a. The method *getDomainAxisIndex(...)* still calls an external method with *axis* as a parameter. Although the set of statements affected by *axis* contains a **return** statement, the defect is not properly repaired using the evading mechanism. In this case, Equation (8) conducts a null judgement on the incorrect variable and adds *l* under the condition of the null. The repair is shown in Figure 9b.

<pre>L1 public int getDomainAxisIndex(CategoryAxis axis) { L2 return this.domainAxes.indexOf(axis); L3 }</pre>	<pre>L1 public int getDomainAxisIndex (CategoryAxis axis) { + if (axis == null) { + throw new NullPointerException ("Null'axis'argument."); } L2 return this.domainAxes.indexOf(axis); L3 }</pre>
(a)	(b)

Figure 9. The sample program 4. (a) defective program 4; (b) repaired program 4.

5.5. The Location of the Null Check

In order to further explain the adding position of the null check of the four repair mechanisms, we discuss the position from three aspects: sequential structure, loop structure, and branch structure.

First, when the null pointer dereference appears in the sequential structure, we need to find all of the statement nodes affected by the fault variable, and then set the null check before the fault statement and the affected statement set, which avoid the error information caused by the null pointer, as shown in Figure 10a.

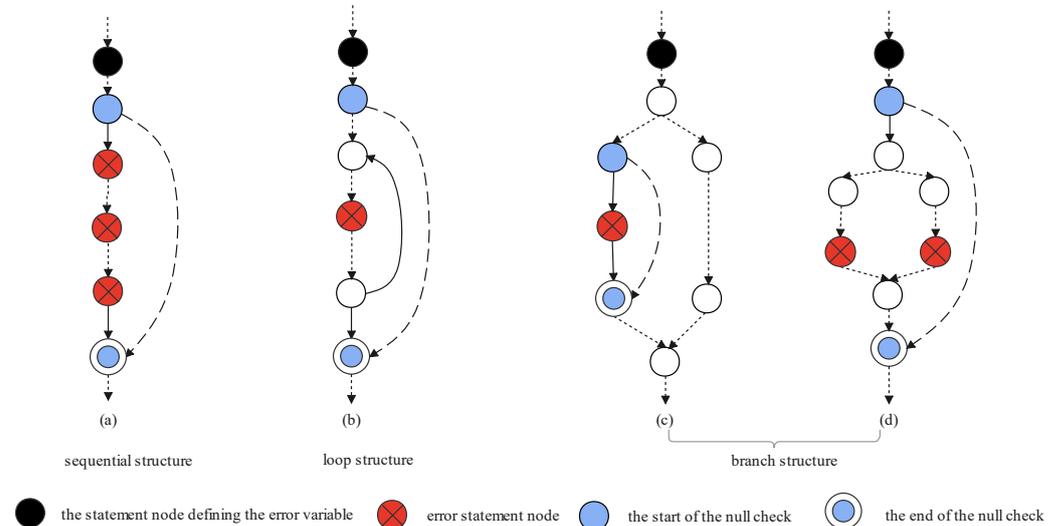


Figure 10. The location of the null check.

Second, when the definition of the defect variable is outside the loop structure, its use in the loop structure fails, which indicates that the function executed by the loop structure does not allow for the occurrence of a null pointer or null value. Therefore, according to this situation, we take the null check before the start of the loop structure, as shown in Figure 10b.

Third, when the definition of the defect variable is outside the branch structure and its use in one branch or more branches fails, if the number of wrong branches exceeds half of the total number of branches, the position of the null check is placed before the beginning of the branch structure, as shown in Figure 10c. However, if the number of wrong branches does not exceed half of the total number of branches, the null check is added before the

fault statement and the affected statement set in the wrong branches, respectively, as shown in Figure 10d.

6. Automatic Repair Algorithm for Null Pointer Dereferences

In the automatic repair for null pointer dereferences, the generation of a program dependency graph and the matching of repair strategies are the key processes. Algorithm 1 describes the process for generating the program dependency graph or the extended program dependency graph. In Step 3, the filter condition for nodes in the executable path are defined based on the post-dominance relation. After that, nodes that satisfy the condition are added to the control dependencies in Step 4. Steps 7 and 8 identify defined and used nodes of variables applying the define-use chain, and then add them to the data dependencies. Finally, the program dependency graph is generated by adding data dependency edges to the control dependency graph in Step 10.

Algorithm 1: Generating a program dependency graph or an extended program dependency graph.

Input: $ps(source, sink, mid-node)$: the executable paths in the control flow graph.

$source$: the source node of an executable path;

$sink$: the sink node of an executable path;

$mid-node$: the intermediate nodes of an executable path;

$def-use$: the define-use relationships between statement nodes;

Output: G : program dependency graph.

- $data_dep$: data dependencies between statement nodes
- $control_dep$: control dependencies between statement nodes
- $post_dominance(a,b)$: b post-dominates a

Begin

1: $data_dep \leftarrow \emptyset, control_dep \leftarrow \emptyset$

2: **for** $p \in ps$ **do**

3: **if** $\neg post_dominance(source, sink) \ \&\& \ post_dominance(mid-node, sink)$ **then**

4: $control_dep \leftarrow add(source, sink)$

5: **end if**

6: **end for**

7: **for** $du \in def-use$ **do**

8: $data_dep \leftarrow add(def, use)$

9: **end for**

10: $G \leftarrow merger(control_dep, data_dep)$

11: **return** G

END

In the process of selecting the four repair mechanisms, we give priority to the successful repair of defects that have the least impact on other functions of the program, precisely because it reduces the side effects on the program as much as possible. The assignment mechanism is used to assign a correct value to the fault variable at the source of the defect as far as possible, which can ensure that the wrong program is fundamentally repaired and the modified program code range is the smallest. The restraint mechanism and evading mechanism are applied to different contexts, respectively. The restraint mechanism adds null checking to bring the code set affected by the defect into the security scope of execution. The evading mechanism changes the execution path of the program by adding appropriate statements to avoid null pointer dereferences. Compared with the assignment mechanism, the scope of code changes in these two mechanisms is larger. Therefore, when matching the repair strategy, first judge the context that conforms to the assignment mechanism, and then judge whether it conforms to the restraint mechanism and evading mechanism. Finally, the transfer mechanism throws the null pointer dereference to the external calling program,

which is captured and processed by the external. Its repair scope is further expanded and depends on external methods, so it is the last repair mechanism to be matched.

In the matching process of repair mechanisms, the decision tree is used in this paper to predict which repair mechanism can be selected to obtain candidate patches for defects. The applicable features of the repair mechanism summarized in Section 4 are learned in the summarized training samples, and the new defects are predicted in the learned classification model, where the classification rules are shown in Figure 11. Each branch of the decision tree prediction model will select a repair mechanism. After that, in the process of applying the repair mechanism, the process ends as soon as a candidate patch is acquired.

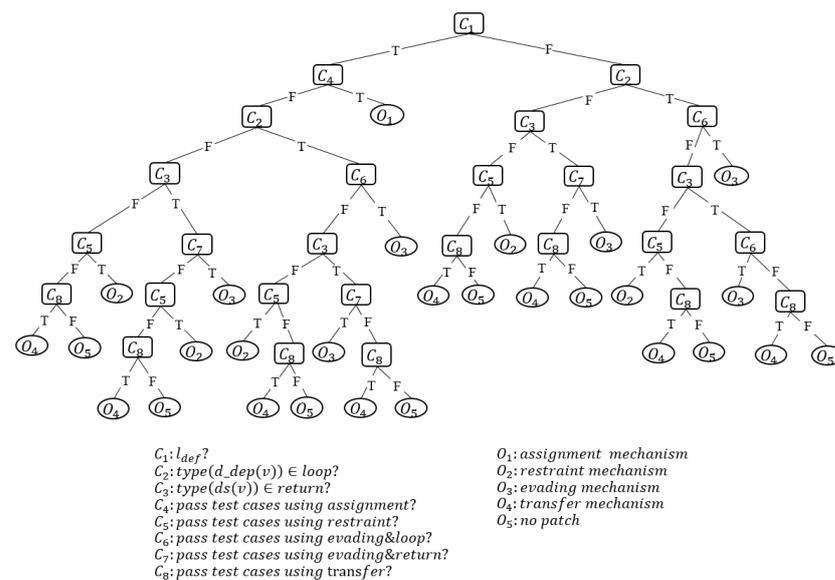


Figure 11. The decision tree corresponding to the choice of repair mechanisms.

Whether the definition position of the fault variable l_{def} can be located should be judged first. If it is found, the test cases are used to verify the repair of the assignment mechanism. If it is not found or the patch cannot pass the test cases, whether the set of statements $d_def(v)$ that are data-dependent by the faulty variable contains the loop statement should be judged next. If $\text{type}(d_def(v)) \in \text{loop}$ is true, the test cases are used to judge whether the evading mechanism can obtain candidate patches. However, if $\text{type}(d_def(v)) \in \text{loop}$ is false or the test cases fail to pass the verification, the set $ds(v)$ of statements affected by the fault variable needs to be judged regarding whether the **return** statement is included in the set. Then, if $\text{type}(ds(v)) \in \text{return}$ is true, the test cases that verify the evading mechanism continue to be used. If it is false, then the repair of the restraint mechanism is verified by using the test cases. Finally, if the candidate patch cannot be obtained through the evading mechanism or restraint mechanism, an exception is thrown to **try** to repair through the transfer mechanism.

7. Evaluation

Based on the defect detection in DTSJava [27], this paper carried out further research to implement DTSFix for the automatic repair of null pointer dereferences. The analysis procedure of DTSFix described by the UML activity diagram is shown in Figure 12. This section focuses on an experimental evaluation of DTSFix, with the main purpose of assessing DTSFix's ability in practice. Experiment 1 compares the repair of 15 null pointer dereferences in Defects4J by DTSFix with other existing tools to verify the superiority of DTSFix. Further, experiment 2 verifies the effectiveness of DTSFix on various datasets and compares the repair capabilities of the four mechanisms by working on the defect detection tool DTSC [28], the public dataset BugSwarm [29], and Bugs.jar [30].

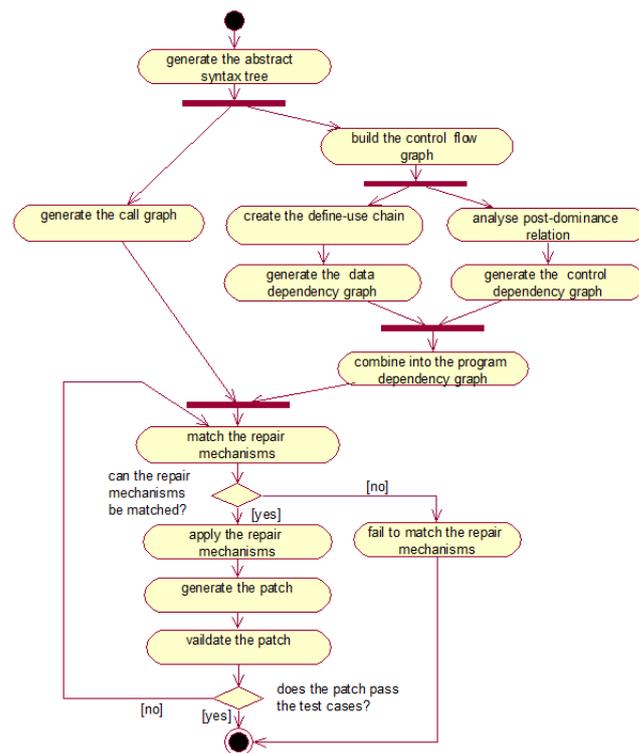


Figure 12. The analysis procedure of DTSFix.

7.1. Comparison of Repair Results between DTSFix and Existing Repair Tools

Experiment 1 takes 15 null pointer dereferences from Defects4J summarized by VFix as the dataset. Table 1 lists the details of the null pointer dereferences in the four projects of Defects4J, including seven in Chart, five in Lang, and three in Math. To better compare DTSFix with other tools, we chose the same dataset that they used. As for evaluation metrics, recall and precision were applied, and they are defined as Equations (9) and (10). The higher the recall, the stronger the patch generation capability of the automatic repair tool, and the higher the precision, the higher the quality of the patches generated by the repair tool.

$$RE(\text{recall}) = \frac{COP(\text{correct patches})}{DEN(\text{all defects})} \quad (9)$$

$$PR(\text{precision}) = \frac{COP(\text{correct patches})}{CAP(\text{candidate patches})} \quad (10)$$

Table 1. Null pointer dereferences in Defects4J.

Project	Defect ID	Defects
Chart	2,4,14,15,16,25,26	7
Lang	20,33,39,47,57	5
Math	4,70,79	3
Time	-	0
Total	-	15

After selecting the appropriate dataset, this paper compared DTSFix with nine typical APR tools. jGenProg [9], JKali [3], Nopol [2], ACS [31], CapGen, HDRepair [8], and SimFix [32] are general automatic repair tools, whereas NPEFix [7] and VFix are automatic repair tools specifically for null pointer dereferences. Figure 13 shows a comparison of the experimental results, where the repair results of both the general repair tools and VFix are taken from their papers. We can see that SimFix is the best among the eight general repair

tools, with a recall of 27% (4/15) and a precision of 80% (4/5). In contrast, DTSTFix has a recall of 73% (11/15), which is 46% higher than SimFix, and a precision of 100%, which is 20% higher than SimFix. The experimental results show that DTSTFix is more effective than the existing general repair tool for null pointer dereferences.

	Chart							Lang					Math		
	2	4	14	15	16	25	26	20	33	39	47	57	4	70	79
SimFix				×					√	√				√	√
jGenProg				×		×								√	
JKali				×		×	×								
ACS			√							×			√		
Nopol		×				×	×						×		
CapGen												√		√	
HDRRepair														√	
NPEFix		×	×	×	×	×	×	×	√			√	×	×	×
VFix		√	√	√	√	√	√	√	√	√	√	√	√	×	
DTSTFix		√	√	√		√	√		√	√	√	√	√	√	

correct patch
 overfitting patch
 incorrect patch

Figure 13. The comparison results with other automatic repair tools.

A comparison among DTSTFix, NPEFix, and VFix is shown in the last three rows of Figure 13, which are specialized in repairing null pointer dereferences. NPEFix generates 12 candidate patches, of which, only 2 are correct, while DTSTFix repairs 11 defects successfully. Although VFix repairs 13 defects, it contains 1 over-fitting patch, whereas all the candidate patches generated by DTSTFix are correct. As is known to all, over-fitting patches do not completely repair the defects and may damage some of the functions in the program. Thus, manual verification is required to ensure the quality of patches, which, however, will increase human work and violate the original intention of automatic repair. Therefore, reducing over-fitting patches is of great importance, which is the strength of DTSTFix. In Math 70, the defect appears in the statement $fmin = f.value(min)$ in the method $solve()$, where f is null. However, f is declared as an interface `UnivariaterrealFunction`, which contains multiple classes implementing the interface of `SinFunction` and `QuinticFunction`. When the defect variable is declared as an interface or a parent class with subclasses, the repair needs to select the appropriate class implementing the interface or subclass to create a new object using their constructor. The value flow graph in VFix mainly focuses on the change and transmission of variables, and cannot obtain the types of active variables at the program point, so it is impossible to find the class that implements the interface or the subclass being used to assign the value to the defect variable. However, DTSTFix can obtain the types of active variables in program points through context, and can create appropriate objects according to the specific class implementing the interface or the subclass types currently used. In the main calling method $solve()$, there is a definition of the same type as f : `UnivariaterrealFunction f = new SinFunction()`. DTSTFix applies the extended program dependency graph to find the definition, and adds its same class implementing the interface $f = new SinFunction()$ before using the defect variable. In this way, the defect can be successfully repaired.

7.2. The Repair Ability of DTSTFix on Large Projects

To verify the effectiveness of the DTSTFix, we needed to collect a large number of null pointer dereferences from different datasets. DTSC, BugSwarm, and Bugs.jar were selected as the datasets in experiment 2. DTSC, which contains 522 java files with a total of

approximately 160,000 lines of code, employs a static analysis approach to detect defects. BugSwarm and Bugs.jar are both public datasets. After the defect detection on the three datasets, it was manually confirmed that DTSC, BugSwarm, and Bugs.jar contain 69, 25, and 32 null pointer dereferences, respectively. Table 2 lists the number of candidate patches (CAP), correct patches (COP), and defects (DEN) in DTSC, BugSwarm, and Bugs.jar.

Table 2. The repair result of experiment 2.

Projects	Defects	Repair Mechanisms (CAP/COP/DEN)				Total (CAP/COP/DEN)	Evaluation (RE/PR)
		Assignment	Restraint	Evading	Transfer		
DTSC	69	7/8/10	22/28/29	10/14/16	9/12/14	48/62/69	69.5%/77.4%
BugSwarm	25	3/4/4	9/11/13	2/3/4	2/4/4	16/22/25	64.0%/72.7%
Bugs.jar	32	2/3/5	13/16/17	1/2/4	4/5/6	20/26/32	62.5%/76.9%

As shown in Table 2, the recall of all three tools exceeds 60%, with an average recall of 65.3%, and the precision of candidate patches exceeds 70%, with an average precision of 75.7% for all three tools. To some extent, these two metrics are constrained by each other, because a higher recall means that the repair tool generates more patches and more overfitting patches are involved as well, leading to a decrease in precision. The experimental results show that DTSFix exhibits a relatively high precision of candidate patches on a wide scope of datasets while ensuring a stable recall.

To accurately evaluate the repair capability of different repair mechanisms in DTSFix, Figure 14 compares the precision of the candidate patches generated by the four repair mechanisms on the three datasets, and the fold line shows the average precision for the four mechanisms. It is widely observed in Figure 14 that the precision of candidate patches from the assignment mechanism and restraint mechanism are significantly higher than that of the other two, and the repair ability of the evading and transfer mechanisms are not very stable. This is mainly because the evading and transfer mechanisms have a greater impact on the program semantics and the repairs are associated with the program code. Figure 15 compares the number of correct patches for the four repair mechanisms of DTSFix on the three data sets. It can be seen that the restraint mechanism produces more correct patches and candidate patches than the other three repair mechanisms. The reason is to that these three mechanisms adapt to fewer situations than the restraint mechanism, and the added repair statements in the three mechanisms may have side effects on the program semantics, while the restraint mechanism only sets the null judgment in front of the codes affected by defects and completes the repair within a small change.

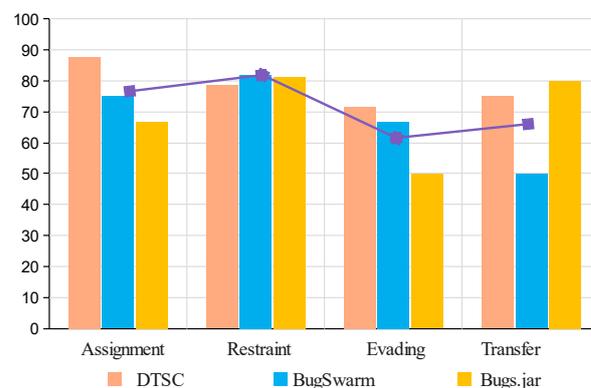


Figure 14. The precision of candidate patches for four repair mechanisms(%).

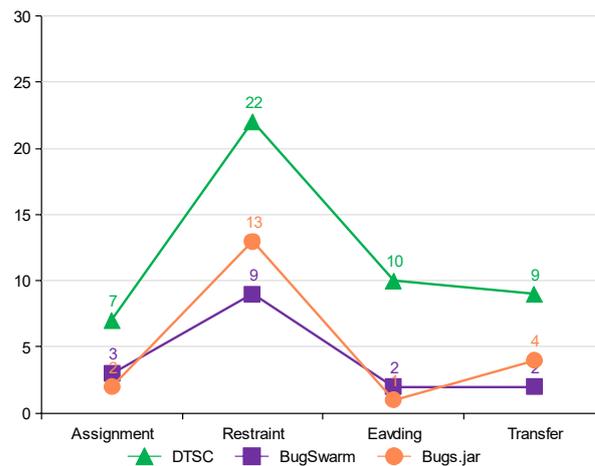


Figure 15. The number of correct patches for four repair mechanisms.

7.3. Discussion and Limitations

According to the existing automatic repair research, the appropriate repair location is a key factor to determine the candidate patch. By analyzing the program dependency graph of the whole program or the extended program dependency graph, accurately grasping the data and control dependencies related to the error variables can help us to analyze the context and match the repair template to select the appropriate repair location. In order to further improve the ratio of correct patches, we trained a large number of defective programs to build a decision tree to predict the repair methods of defective programs. Through screening each attribute, we eliminated inappropriate repair mechanisms, which reduces the generation of incorrect patches and improves the repair efficiency. Moreover, since the null pointer dereference is a semantic defect, the defect detection tool using static analysis technologies in this paper can carry out regression testing, which plays a good role in the patch verification stage. The best way to eliminate software defects is to set preventive measures in advance to avoid damage caused by defects. The processing method is also being studied in blockchain and microservices, which has made good progress [33–35].

Recently, annotation technology can be used to check null pointers in programming languages. Adding a non-null annotation to a particular field will be equivalent to setting a null check judgment in advance in this field. In the stage of program design and implementation, the appearance of a non-null annotation will reduce a large amount of repetitive work of verification and relieve code redundancy. According to the particular requirements of some fields in the design stage, the non-null annotation of custom-defined can help the program to select the processing method and request. Non-null annotation is used to check whether the field is empty during program operation, but DTSFix applies static analysis technologies to analyze and repair the program.

However, in the repair process, our method still has some shortcomings. When the program structure of a program is very complex, it is difficult to summarize it into a repair template to repair. This is also a common problem in the automatic repair of programs based on the repair template. Faced with such a program, DTSFix will give up the repair after the repair time out or the candidate patch cannot be obtained.

8. Conclusions

In this paper, we implemented the automatic repair for null pointer dereferences by using the program dependency graph to obtain program dependencies, proposing four repair mechanisms and eight repair templates. DTSFix proposed in this paper first analyzes program dependencies of the defective program to select repair mechanisms by matching the context and the conditions of the repair mechanism. It is proven that DTSFix can accurately judge the repair location and generate appropriate patches. The experiment evaluated DTSFix with 15 null pointer dereferences in the Defects4J, and the result shows

that the automatic program repair guided by the program dependency graph can generate candidate patches for 11 defects, and that all of them are correct. Additionally, the experiments on a large project and two public datasets further verify the effectiveness of DTSFix in practical projects. In the future, we plan to further improve the precision of DTSFix and summarize more repair mechanisms for null pointer dereferences based on the program dependency graph.

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

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

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

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

References

1. Vu, D.L.; Pashchenko, I.; Massacci, F. Please hold on: More time = more patches? Automated program repair as anytime algorithms. In Proceedings of the 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), Madrid, Spain, 1 June 2021; pp. 9–10. [\[CrossRef\]](#)
2. Martinez, M.; Durieux, T.; Sommerard, R.; Xuan, J.F.; Monperus, M. Automatic Repair of Real Bugs in Java: A Large-Scale Experiment on the Defects4J dataset. *Empir. Softw. Eng.* **2017**, *22*, 1936–1964. [\[CrossRef\]](#)
3. Qi, Z.; Long, F.; Achour, S.; Rinard, M. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA 2015), New York, NY, USA, 12–17 July 2015; pp. 24–36. [\[CrossRef\]](#)
4. Hong, S.; Lee, J.; Lee, J.; Oh, H. SAVER: Scalable, precise, and safe memory-error repair. In Proceedings of the ICSE'20: 42nd International Conference on Software Engineering, New York, NY, USA, 27 June 2020; pp. 271–283. [\[CrossRef\]](#)
5. Klieber, W.; Martins, R.; Steele, R.; Churilla, M.; Svoboda, D. Automated Code Repair to Ensure Spatial Memory Safety. In Proceedings of the 2021 IEEE/ACM International Workshop on Automated Program Repair (APR), Madrid, Spain, 1 June 2021; pp. 23–30.
6. Aslanyan, H.; Arutunian, M.; Keropyan, G.; Kurmangaleev, S.; Vardanyan, V. BinSide: Static Analysis Framework for Defects Detection in Binary Code. In Proceedings of the 2020 Ivannikov Memorial Workshop (IVMEM), Orel, Russia, 25–26 September 2020; pp. 3–8. [\[CrossRef\]](#)
7. 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.
8. Le, X.B.D.; Lo, D.; Le Goues, C.L. History driven program repair. In Proceedings of the 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), Osaka, Japan, 15 March 2016; pp. 213–224. [\[CrossRef\]](#)
9. Le Goues, C.; Nguyen, T.; Forrest, S.; Weimer, W. GenProg: A Generic Method for Automatic Software Repair. *IEEE Trans. Softw. Eng.* **2012**, *38*, 54–72. [\[CrossRef\]](#)
10. René, J.; Darioush, J.; Michael, D.E. A database of existing faults to enable controlled testing studies for Java programs. In Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA 2014), New York, NY, USA, 21 July 2014; pp. 437–440. [\[CrossRef\]](#)
11. Xu, X.Z.; Sui, Y.L.; Yan, H.; Xue, J.L. VFix: Value-Flow-Guided Precise Program Repair for Null Pointer Dereferences. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 512–523.
12. Noda, K.; Yokoyama, H.; Kikuchi, S. Sirius: Static program repair with dependence graph-based systematic edit patterns. In Proceedings of the 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), Luxembourg, 27 September–1 October 2021; pp. 437–447. [\[CrossRef\]](#)
13. Villanueva, O.M.; Trujillo, L.; Hernandez, D.E. Novelty search for automatic bug repair. In Proceedings of the 2020 Genetic and Evolutionary Computation Conference (GECCO'20), New York, NY, USA, 8–9 July 2020; pp. 1021–1028. [\[CrossRef\]](#)

14. Dantas, A.; de Souza, E.F.; Souza, J.; Camilo, C.G. Code Naturalness to Assist Search Space Exploration in Search-Based Program Repair Methods. In Proceedings of the 11th International Symposium on Search-Based Software Engineering (SSBSE), Tallinn, Estonia, 31 August–1 September 2019; Volume 11664, pp.164–170. [[CrossRef](#)]
15. Yang, G.; Jeong, Y.; Min, K.; Lee, J.W.; Lee, B. Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair. *Symmetry* **2018**, *10*, 92. [[CrossRef](#)]
16. White, M.; Tufano, M.; Martinez, M.; Monperrus, M.; Poshyvanyk, D. Sorting and transforming program repair ingredients via deep learning code similarities. In Proceeding of the 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), Hangzhou, China, 24–27 February 2019; pp. 479–490. [[CrossRef](#)]
17. Gulwani, S.; Radiček, I.; Zuleger, F. Automated clustering and program repair for introductory programming assignments. In Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Philadelphia, PA, USA, 18–22 June 2018; pp. 465–480. [[CrossRef](#)]
18. Huq, F.; Hasan, M.; Haque, M.M.A. Review4Repair: Code review aided automatic program repairing. *Inf. Softw. Technol.* **2022**, *143*, 106765. [[CrossRef](#)]
19. Liu, K.; Koyuncu, A.; Kim, D.; Bissyandé, T.F. TBar: Revisiting template-based automated program repair. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, Beijing, China, 15–19 July 2019; pp. 31–42. [[CrossRef](#)]
20. Hua, J.R.; Zhang, M.S.; Wang, K.Y.; Khurshid, S. Towards practical program repair with on-demand candidate generation. In Proceedings of the 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 12–23. [[CrossRef](#)]
21. Afsson, A.; Manish, M.; Kathryn, S.; Yuriy, B.; Claire, L.G. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Trans. Softw. Eng.* **2019**, *47*, 2162–2181. [[CrossRef](#)]
22. Gao, X.; Wang, B.; Duck, G.J.; Ji, R.Y.; Xiong, Y.F. Roychoudhury A. Beyond Tests: Program Vulnerability Repair via Crash Constraint Extraction. *ACM Trans. Softw. Eng. Methodol.* **2021**, *30*, 1–27. [[CrossRef](#)]
23. Nguyen, H.D.T.; Qi, D.W.; Rocychoudhury, A.; Chandra, S. SemFix: Program Repair via Semantic Analysis. In Proceedings of the 35th International Conference on Software Engineering (ICSE 2013), San Francisco, CA, USA, 18–26 May 2013; pp. 772–781.
24. Wen, M.; Chen, J.J.; Wu, R.X.; Hao, D.; Cheung, S.C. Context-aware patch generation for better automated program repair. In Proceedings of the 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), Gothenburg, Sweden, 27 May–3 June 2018; pp. 1–11. [[CrossRef](#)]
25. Ghanbari, A.; Benton, S.; Zhang, L.M. Practical program repair via bytecode mutation. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA), Beijing, China, 15–19 June 2019; pp. 19–30. [[CrossRef](#)]
26. Koyuncu, A.; Liu, K.; Bissyandé, T.F.; Kim, D.; Klein, J.; Monperrus, M.; Le Traon, Y. Fixminer: Mining relevant fix patterns for automated program repair. *Empir. Softw. Eng.* **2020**, *25*, 1980–2024. [[CrossRef](#)]
27. Zhou, H.B.; Jin, D.H.; Gong, Y.Z. Application of Interval Arithmetic in Software Testing Based on Field-Sensitive Point-to Analysis. *J. Comput. Res. Dev.* **2012**, *49*, 1852–1862.
28. Wang, S.D.; Yin, W.J.; Dong, Y.K.; Zhang, L.; Liu, H. Data flow analysis for sequential storage structures. *J. Softw.* **2020**, *31*, 1276–1293.
29. Tomassi, D.A.; Dmeiri, N.; Wang, Y.C.; Bhowmick, A.; Liu, Y.C.; Devanbu, P.T.; Vasilescu, B.; Rubio-González, C. Bugswarm: Mining and continuously growing a dataset of reproducible failures and fixes. In Proceedings of the 41st International Conference on Software Engineering (ICSE), Montreal, QC, Canada, 25–31 May 2019; pp. 339–349. [[CrossRef](#)]
30. Saha, R.; Lyu, Y.J.; Lam, W.; Yoshida, H.; Prasad, M.R. Bugs.jar: A large-scale, diverse dataset of real-world java bugs. In Proceedings of the 15th International Conference on Mining Software Repositories (MSR), Gothenburg, Sweden, 28–29 May 2018; pp. 10–13.
31. Xiong, Y.F.; Wang, J.; Yan, R.F.; Zhang, J.C.; Han, S.; Huang, G.; Zhang, L. Precise Condition Synthesis for Program Repair. In Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE), Buenos Aires, Argentina, 20–28 May 2017; pp. 416–426. [[CrossRef](#)]
32. Jiang, J.J.; Xiong, Y.F.; Zhang, H.Y.; Gao, Q.; Chen, X.Q. Shaping Program Repair Space with Existing Patches and Similar Code. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2018), Amsterdam, The Netherlands, 15–21 July 2018; pp. 298–309. [[CrossRef](#)]
33. Six, N.; Herbaut, N.; Salinesi, C. Blockchain software patterns for the design of decentralized applications: A systematic literature review. *Blockchain Res. Appl.* **2022**, 100061. [[CrossRef](#)]
34. Gorski, T. Reconfigurable Smart Contracts for Renewable Energy Exchange with Re-Use of Verification Rules. *Appl. Sci.* **2022**, *12*, 5339. [[CrossRef](#)]
35. Waseem, M.; Liang, P.; Shahin, M.; Di Salle, A.; Marquez, G. Design, monitoring, and testing of microservices systems: The practitioners’ perspective. *J. Syst. Softw.* **2021**, *182*, 111061. [[CrossRef](#)]